Mechanically Verifying Concurrent Programs

David Moshe Goldschlag

Technical Report 71 May, 1992

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work, originally presented as a Ph.D. thesis at the University of Texas at Austin, was sponsored in part at Computational Logic, Inc. by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency, or the U.S. Government.

DEDICATION *To my parents*

ועשה חסד לאלפים לאהבי ולשמרי מצותי
שמות כ:ו

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

Acknowledgements

I would like to thank my committee members, Bob Boyer, J Moore, Leslie Lamport, Jay Misra, and Al Mok, for their thoughtful help throughout my graduate work. Special thanks are due to Matt Kaufmann who was a constant source of good and friendly advice (and of Pc-Nqthm).

This work was begun in collaboration with Jimi Crawford in 1986. The project has greatly evolved since then, but the general scheme of defining an interpreter for concurrent programs, and only using proved proof rules for program verification was decided jointly at that early date. That year of collaboration was hard work and great fun.

The writing of this thesis was completed after I began working for the National Security Agency. However, the vast majority of the research was supported by Don Good's group, first at the University of Texas at Austin's Institute for Computer Science and Computer Applications, and later at Computational Logic, Inc. Don has succeeded in building a talented and friendly research group supported by excellent administrative personnel and equipment. I appreciate the friendship and help of Dianne King Akers, Sandy Olmstead, Ron Olphie, and Laura Tice over these years. Mike Smith deserves thanks for keeping Scribe's dissertation style format current, and for allocating me time to do my research.

I would also like to thank my friends Bill Bevier, Eliezer Levy, Boaz Super, Warren Hunt, and Bill Young.

This thesis is dedicated to my parents. They imbued in me the curiosity to learn, gave me the opportunity to study, and taught me the value of hard work. Most of all they gave me their love.

And finally, אחרון אחרון חביב, I thank my wife Barbara for being my best friend.

Abstract

This thesis develops a sound, mechanically verified proof system suitable for the mechanical verification of both the safety and liveness properties of concurrent programs. Mechanical verification increases the trustworthiness of a proof. The properties proved may be (non-existentially quantified) first order, and the programs may be parameterized. The proof system provides a mechanized framework for reasoning about concurrent programs under a variety of fairness assumptions. It is demonstrated by the mechanically verified proofs of four concurrent programs. This thesis presents several significant results, including the first extensive mechanized use of proof rules which are theorems of an operational semantics. This research also prompted the development of both the functional variables [13] and Defn-Sk [53] extensions to the Boyer-Moore logic. Thesis chapters include:

- A formalization, in the Boyer-Moore Logic [11], of an interpreter for concurrent programs with non-deterministic statements. This operational semantics is the foundation for the development of a sound proof system similar to Unity [17].
- The presentation of Unity's major proof rules (except for superposition) as theorems about this interpreter. This includes the infinite disjunction and substitution axioms. Unity's theorems remain true for programs with non-deterministic statements.
- The extension of the Unity logic to include the literature's notions of weak and strong fairness [61], and deadlock freedom. The latter two notions guarantee freedom from starvation and absence of deadlock, respectively.
- A new proof rule for reasoning about properties that are eventually invariant.
- A proof of an n-node mutual exclusion algorithm.
- A proof of a minimum tree value algorithm. This proof demonstrates reasoning about programs with multiple instances of several statements and a complicated data structure.
- A proof of an n-node dining philosopher's program under the assumptions of strong fairness and deadlock freedom.
- A proof of an n-node delay insensitive FIFO queue, where statements correspond to Martin's production rules [64]. Both the safety and liveness properties proved are non-propositional theorems.

This thesis both contributes to the science of mechanized program verification and lays the groundwork for future research.

Chapter 1

INTRODUCTION

Since the semantics of a programming language can be precisely specified, programs are mathematical objects whose correctness can be assured by formal proof. Mechanical verification, which uses a computer program to validate a formal proof, greatly increases one's confidence in the correctness of the proved theorem. This thesis develops a mechanically verified proof system for concurrent programs, and demonstrates the use of this proof system by the mechanically verified correctness proofs of four parameterized multiprocess programs.

The proof system presented here differs from most other proof systems because its proof rules are theorems. All the proof rules are either theorems of an operational semantics of concurrency (and have been mechanically verified) or are a conservative extension of that operational semantics. In the first case, the proof system is (relatively) complete, since all properties may ultimately be derived directly from the operational semantics. In the second case, completeness depends upon whether certain proof rules in the literature are sufficient. However, the proof system is sound, since the operational semantics justifies the new proof rules. The methodology of justifying proof rules by an operational semantics and the mechanized use of those proof rules (without appealing to the operational semantics) in proofs are used extensively for the first time in this thesis. Using these proof rules facilitates the development of mechanized proofs which closely resemble the structure of hand proofs, but are necessarily longer since all concepts are defined from first principles. The methodology of proving proof rules [28] makes a clear distinction between the theorems in the proof system and the logical inference rules and syntax which define the logic. Since the underlying logic is sound, the proof system is sound, and since the underlying logic has been mechanized, the proof system has been mechanized as well.

This proof system is based on the Unity logic [17] and has been encoded in the Boyer-Moore logic on the Boyer-Moore prover [9, 11]. These are described in the next sections. Unity's major proof rules (except for superposition) are theorems in this proof system and are presented in this thesis. In addition, Unity is extended in several ways: Program statements may be non-deterministic [58]; Unity's proof rules are still sound. Programs may be reasoned about under the assumptions of weak and strong fairness, in addition to Unity's unconditional fairness [23]. Also, a predicate for specifying properties that are eventually invariant is presented, along with several supporting proof rules.

This thesis is organized in the following way: This chapter presents the necessary background material and related work. Chapter 2 presents an operational semantics of concurrency which underlies the proof rules presented in chapter 3. Chapter 3 also discusses several fairness notions and describes how the proof rules are used in practice. Chapter 4 presents the specification, algorithm, and correctness proof of a solution to the mutual exclusion problem. Chapters 5, 6, and 7 do the same for a concurrent solution to finding the minimum node value in a tree, for a solution to the dining philosophers problem under the (unusual) assumptions of strong fairness and absence of deadlock, and for an n-bit delay insensitive FIFO queue [64]. Chapter 8 discusses both the limitations and possibilities of this formalization and compares this work to related mechanizations.

1.1 Unity

Unity, developed by Chandy and Misra and described in [17], defines both a notation for writing concurrent programs, and a logic for reasoning about computations (executions) of those programs. Unity has two important characteristics:

- Unity provides predicates for specifications and proof rules to derive specifications directly from the program text. This type of proof strategy is often clearer and more succinct than an argument about a program's operational behavior.
- Unity separates the concerns of algorithm and architecture. It defines a general semantics for concurrent programs and encourages the refinement of architecture independent programs to architecture specific ones.

The mechanization of Unity presented here focuses on the first characteristic. The latter is a refinement methodology and the necessary proof rules have not been formalized.

As an example of the Unity notation, consider the following program which sorts an array of **N** elements **X[1], ..., X[N]** into non-decreasing order:

```
<∀ I, J : 0 < I < J <= N : X[I], X[J] := X[J], X[I]
                                             IF X[J]<X[I] >
```

This program contains **N(N-1)/2** statements, each of which swaps an out-of-order pair of array elements. The execution of this program is as follows: Some statement is chosen. The condition following the **IF** is evaluated. If the condition is false, the statement's execution is equivalent to a **SKIP** statement. If the condition is true, the statement is executed, and the out-of-order pair is swapped. Another statement is then chosen, and the process is repeated. The only restriction on the scheduling of statements is a fairness restriction, which requires that every statement be scheduled infinitely often. Although execution never terminates, a fixed point may be reached when all statements are equivalent to **SKIP**'s.

To demonstrate the correctness of this program, we must first present the specification. We will do this somewhat informally. The specification is broken down into two parts. The first states that the final array is a permutation of the original array. This is a consequence of the following property: the bag of values that fills the array **X** is unchanged throughout the computation. This sort of property is an invariant: **INVARIANT BAG(X) = K**. This means, roughly, that if the bag of values in array **X** is equal to **K** before executing any statement in the program, then the bag of values in that array is unchanged subsequent to executing any statement in the program. Since every statement at most swaps array values, no values are ever lost, and this invariant is true.

The second part of the specification states that the array will eventually become sorted. This is a liveness (or a progress) property. In Unity, this is stated by **TRUE LEADS-TO SORTED(X)**. The value **TRUE** simply states that there are no preconditions on this property. To prove this liveness property, we use the following measure: Imagine a lexicographic less than relation on **N** elements. If the array is not sorted, then every statement in the program either modifies the array to one with a smaller lexicographic order, or does not change the array at all. Furthermore, if the array is not sorted, some statement modifies the array. By fairness, the array's lexicographic order will eventually decrease. The fact that this lexicographic order decreases may be repeatedly applied by induction, to conclude that eventually the array becomes sorted.

A small note: Unity would not recommend that a sorting program be *implemented* in this way. This algorithm, however, is a starting point for other, more efficient, sorting routines based on the swapping of out-of-order pairs.

The Unity logic differs from other temporal logic proof systems for reasoning about concurrency because it is really only a subset of temporal logic. Unity's specification predicates provide a simple and powerful vocabulary to specify and reason about the behavior of concurrent programs. They permit the specification of many temporal properties without introducing all of temporal logic. However, the specification predicates **INVARIANT** and **LEADS-TO** are operators that take predicates as arguments, and are not quantifiers like ∀, ∃, or temporal logic's **ALWAYS** or **EVENTUALLY**. This means that these operators may not be nested, and that Unity is less expressive than full first order temporal logic. Unity provides proof rules for taking large formal proof steps, and is (relatively) complete, even though it contains fewer proof rules than other temporal logics. This suggests that Unity is simpler and that proofs are more straightforward, since one has fewer necessary tools to work with.

The soundness and completeness of Unity is discussed in [76, 54, 26, 50].

1.2 The Boyer-Moore Logic and Prover

In a mechanically verified proof, all proof steps are validated by a computer program called a theorem prover. Hence, whether a mechanically verified proof is correct is really a question of whether the theorem prover is sound. This question, which may be difficult to answer, need be answered only once for all proofs validated by the theorem prover. The theorem prover used in this work is an extension of the Boyer-Moore prover [9, 11]. This prover has been carefully coded and extensively tested. The Boyer-Moore logic, which is mechanized by the Boyer-Moore prover, has been proved sound [51, 10]. All of the definitions and theorems presented in this thesis have been validated by the extension of the Boyer-Moore prover described here.

Understanding the rest of this thesis requires some familiarity with the Boyer-Moore logic and its theorem prover. The following sections informally describe the logic and the various enhancements to the logic and prover that are used in this work.

Interaction with the theorem prover is through a sequence of events, the most important of which are *definitions* and *lemmas*. A definition defines a new function symbol and is accepted if the prover can prove that the recursion (if any) terminates. Adding a definition is one way of obtaining a conservative extension to the logic. A lemma is accepted if the prover can prove it using the logic's inference rules, from axioms, definitions and previously proved lemmas. By the judicious choice of lemmas and the order of their presentation, a user may guide the theorem prover through the verification of complicated theorems.

1.2.1 The Boyer-Moore Logic

This proof system is specified in an extension of the Nqthm version of the Boyer-Moore logic [11, 12]. Nqthm is a quantifier-free first order logic with equality that permits recursive definitions. All variables are implicitly universally quantified. Nqthm also defines an interpreter function for the quotation of terms in the logic. Nqthm uses a prefix syntax similar to pure Lisp. This notation is completely unambiguous, easy to parse, and easy to read after some practice. Informal definitions of many of the functions used in this thesis follow:

- **`T`** is an abbreviation for **`(TRUE)`** which is not equal to **`F`** which is an abbreviation for **`(FALSE)`**.
- **`(EQUAL A B)`** is **`T`** if **`A=B`**, **`F`** otherwise.
- The value of the term **`(AND X Y)`** is **`T`** if both **`X`** and **`Y`** are not **`F`**, **`F`** otherwise. **`OR`**, **`IMPLIES`**, **`NOT`**, and **`IFF`** are defined appropriately, also treating any non **`F`** argument as **`T`**.
- The value of the term **`(IF A B C)`** is **`C`** if **`A=F`**, **`B`** otherwise.
- **`(NUMBERP A)`** tests whether **`A`** is a natural number.
- **`(ZEROP A)`** is **`T`** if **`A=0`** or **`(NOT (NUMBERP A))`**.
- **`(FIX I)`** returns **`I`** if **`(NUMBERP I)`** is true, **`0`** otherwise.
- **`(ADD1 A)`** equals the successor of **`A`** (i.e., **`A+1`**). If **`(NUMBERP A)`** is **`F`** then **`(ADD1 A)`** is **`1`**.
- **`(SUB1 A)`** is the predecessor of **`A`** (i.e., **`A-1`**). If **`(ZEROP A)`** is **`T`**, then **`(SUB1 A)`** is **`0`**.
- **`(PLUS A B)`** is **`A+B`**, and is defined recursively using **`ADD1`**.

- **(LESSP A B)** is **A<B**, and is defined recursively using **SUB1**.
- Literals are quoted. For example, **'ABC** is a literal. **NIL** is an abbreviation for **'NIL**.
- **(CONS A B)** represents a pair. **(CAR (CONS A B))** is **A**, and **(CDR (CONS A B))** is **B**. Compositions of car's and cdr's can be abbreviated: **(CADR A)** is read as **(CAR (CDR A))**.
- **(LISTP A)** is **T** if **A** is a pair.
- **(LIST A)** is an abbreviation for **(CONS A NIL)**. **LIST** can take an arbitrary number of arguments: **(LIST A B C)** is read as **(CONS A (CONS B (CONS C NIL)))**.
- **'(A)** is an abbreviation for **(LIST 'A)**. Similarly, **'(A B C)** is an abbreviation for **(LIST 'A 'B 'C)**.[1]
- **(LENGTH L)** is the length of the list **L**, or **0** if the argument is not a list.
- **(MEMBER X L)** tests whether **X** is an element of the list **L**.
- **APPLY$** uses a straightforward mapping between quoted literals and functions. **(APPLY$ FUNC ARGS)** is the result of applying the function named **FUNC** to the arguments **ARGS**.[2] For example, **(APPLY$ 'PLUS (LIST 1 2))** is **(PLUS 1 2)** which is **3**.

Recursive definitions are permitted, provided termination can be proved. For example, the function **APPEND**, which appends two lists, is defined as:
Definition: Append

```
(APPEND X Y)
   =
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y)
```

This function terminates because the measure **(LENGTH X)** decreases in each recursive call.

Since the Boyer-Moore logic can be used like a functional programming language, it is often natural to say that a term *returns some value*, instead of the equivalent *is equal to some value*.

1.2.2 Eval$

EVAL$** is an interpreter for partial recursive functions in Nqthm.[3] Informally, the term **(EVAL$ T TERM ALIST) represents the value obtained by applying the outermost function symbol in **TERM** to the **EVAL$** of the arguments in **TERM**. If **TERM** is a literal atom, then **(EVAL$ T TERM ALIST)** is the second component (the **CDR**) of the first value in **ALIST** whose first component (the **CAR**) is **TERM**.

For example, **(EVAL$ T '(PLUS X Y) (LIST (CONS 'X 5) (CONS 'Y 6)))** is **(PLUS 5 6)** which is **11**. **(EVAL$ T (LIST 'QUOTE TERM) ALIST)** is simply **TERM**, since **EVAL$** does not evaluate arguments to **QUOTE**. **QUOTE** can be used to introduce what look like free variables into an expression. For instance, **(EVAL$ T (LIST 'PLUS 'X (LIST 'QUOTE Y)) (LIST (CONS 'X 5)))** is **(PLUS 5 Y)**. Unfortunately, **(EVAL$ T (LIST 'PLUS 'X (LIST 'QUOTE Y)) (LIST (CONS 'X 5)))** is somewhat difficult to read.

---

[1]Actually, this quote mechanism is a facility of the Lisp reader [85].

[2]This simple definition is only true for total functions but is sufficient for this paper [12].

[3]And like **APPLY$**, **EVAL$** is only used here to interpret total functions.

The Lisp backquote syntax [85] can be used to write an equivalent expression.[4] Backquote (**‘**) is similar to quote (**’**) except that under backquote, terms preceded by a comma are not evaluated. Therefore, the terms **‘(PLUS X (QUOTE ,Y))** and **(LIST ’PLUS ’X (LIST ’QUOTE Y))** are identical to the Lisp reader. So, **(EVAL$ T (LIST ’PLUS ’X (LIST ’QUOTE Y)) (LIST (CONS ’X 5)))** can be rewritten as **(EVAL$ T ‘(PLUS X (QUOTE ,Y)) (LIST (CONS ’X 5)))**.

The meaning of backquote presented here is only one of several legitimate interpretations of the definition of backquote presented in [85] and is called the Nqthm interpretation of backquote. Boyer and Moore’s Common Lisp code defining this interpretation is presented in Appendix A, page 91. Informally, that code interprets backquoted expressions in the following way:

- Backquote of a list is the list of the backquote of each of the elements in the list: **‘(A B C D)** equals **(LIST ‘A ‘B ‘C ‘D)**.
- Backquote of a term preceded by a comma is that term: **‘,(ADD1 1)** equals **(ADD1 1)** and **‘,A** equals **A**.
- Backquote of a literal atom is the quote of that atom: **‘A** equals **’A**.

1.2.3 Functional Instantiation

Sometimes it is useful to describe classes of functions and prove theorems about the whole class. One way to do this is to define a typical function in the class and prove the desired theorems without appealing to the extraneous properties of this function. However, it might be difficult to demonstrate that only the representative properties were used. In earlier versions of the Boyer-Moore logic, one would therefore add axioms about an undefined function symbol. For example, if one wanted to prove theorems about all numbers, a new, undefined function symbol would be introduced (say the zero-ary function **FOO**), and the single axiom **(NUMBERP (FOO))** would be added to the prover’s database. There is a risk, however, since adding random axioms introduces potential inconsistencies. For this hypothetical function **FOO**, we are not terribly concerned, since it is obvious that some zero-ary function satisfies the axiom (e.g., any function equal to **0**). However, for more complicated axiom sets, the risk becomes more pronounced.

One solution is to follow a methodology where axioms are only added about undefined function symbols if they have been proved as theorems about existing function symbols. This alternative is adequate and was used effectively in [21]. However, this methodology is inconvenient, and does not truly take advantage of the possibilities of partially defined function symbols.

In response to this need, the Boyer-Moore logic was extended with the notion of functional variables [13]. A class of functions is represented by a partially constrained function: a function symbol possessing only the class’s shared properties, which are called constraints. Theorems about such a function are theorems for a whole class of functions, and may be instantiated by any function in the class. This extension both formalizes the methodology discussed above, and permits the instantiation of function symbols, in a manner not unlike the instantiation of variables in a first-order system. In contrast to the automatic instantiation of many variables in rewrite rules in the Boyer-Moore logic, however, this functional instantiation is not done automatically.

Partially constrained function symbols are defined by the *constrain* event which introduces new function symbols and their constraints. To ensure the consistency of the constraints, one must demonstrate that they are satisfiable. Therefore, the constrain event also requires the presentation of one old function symbol as a model for each new function symbol; the constraints, with each new symbol substituted by its model, must be provable [13]. There is no logical connection between the new symbols and their models, however; providing the models is simply a soundness guarantee.

[4]Thanks to Matt Kaufmann for showing me how backquote would be useful in this context. Interestingly, Matt objects to my use of backquote in publications because the resulting expressions are often difficult to parse. I agree with this attitude for most syntactic sugar and Matt’s objection is certainly valid. However, un-backquoted versions of the quoted expressions used in this thesis are quite impossible to read, and, perhaps more importantly, to type in correctly.

1.2.4 Definitions with Quantifiers

Although the Boyer-Moore logic does not define an existential quantifier, the combination of recursion and the implicit universal quantification permits the specification of many interesting formulas. One might argue that these restrictions do not limit what may be reasoned about within computer science, since computer scientists are usually reasoning about bounded domains. In fact, an early version of this project was successfully completed without existential quantifiers [21], even though the domain in question involves both executions that are infinitely long, and the specification of properties that imply the existence of infinitely many solutions. However, working within this restricted framework has two deficiencies: the appropriate formulas are often odd looking; and, certain convenient and helpful theorems may not be stated at all.

Therefore, in this work, a facility permitting fully quantified definitions in the Boyer-Moore logic is used. A more primitive version of this extension was first discussed in [27]; the present facility was developed and tested by Matt Kaufmann [53] and is called the Defn-Sk definitional principle. The ability to define functions with full quantification in the body of the function permits the statement of any first order formula.

It is interesting to note, and important to point out, that a solution very similar to the solution to the mutual exclusion problem presented and verified here in chapter 4 was also verified in [21] in a proof system without quantifiers. The proof was almost identical, with a nearly one to one correspondence between theorems in each proof script. Therefore, although it is indisputable that quantifiers permit the formalization of concepts that cannot be stated in an unquantified logic, it is not clear that full quantifiers facilitate the proof of theorems that could have been stated in an unquantified logic. This experience suggests that the proof effort will be very similar.

To include quantifiers in the body of a definition in the Boyer-Moore logic, the quantifiers are removed by a technique called Skolemization. If the definition is not recursive, adding the Skolemized definition preserves the theory's consistency [53].

For example, suppose we wish to define:
Definition.

$$\begin{array}{l}\mathbf{(P\ X_1\ X_2\ \ldots\ X_N)}\\ \quad\Leftrightarrow\\ \mathbf{BODY}\end{array}$$

where **`P`** is a new function symbol of arity **`N`**, and **`BODY`** is a quantified term mentioning only free variables in the set $\{X_1, \ldots, X_N\}$ and only old function symbols. Furthermore, the outermost operator in **`BODY`** is **`FORALL`**, **`EXISTS`**, or some other logical connective, and within **`BODY`**, **`FORALL`** and **`EXISTS`** are only arguments to **`FORALL`**, **`EXISTS`**, or some other logical connective.

We may consider this definition to be the conjunction of two formulas:

$$\begin{array}{l}\mathbf{((P\ X_1\ X_2\ \ldots\ X_N)}\\ \quad\Rightarrow\\ \ \mathbf{BODY)}\\ \wedge\\ \mathbf{((P\ X_1\ X_2\ \ldots\ X_N)}\\ \quad\Leftarrow\\ \ \mathbf{BODY)}\end{array}$$

We then Skolemize (positive Skolemization—to preserve consistency) both conjuncts by substituting for each existential a Skolem function. The resulting formula is quantifier-free and can be added as an axiom. Consistency is preserved since such a definition is truly an abbreviation (there is no explicit recursion and no interpreter axioms are added). Finally, the meaning of the Skolemized formula is the same as the original definition due to the correctness of Skolemization. One may have to use the function symbols introduced by Skolemization in the proof (and statement) of theorems about the newly defined function; this analysis is often tedious.

As a convenience, one may abbreviate nested `FORALL`'s by putting all consecutive universally quantified variables in a list. Therefore, `(FORALL X (FORALL Y (EQUAL X Y)))` may be abbreviated to `(FORALL (X Y) (EQUAL X Y))`. Nested `EXISTS`'s may be shortened similarly.

1.2.5 The Kaufmann Proof Checker

The Boyer-Moore prover automatically proves a lemma by heuristically applying sound inference rules to simplify it to a value other than `F`. Sometimes, it is easier to direct the proof process at a lower level. The Kaufmann Proof Checker [52] is an interactive enhancement to the Boyer-Moore prover. It allows the user to manipulate a formula (the original goal) using sound operations, perhaps creating additional goals; once all goals have been proved, the original formula has been proved. The prover will then accept the new theorem, which will be used as if it were proved automatically.

All new function symbols extending the Boyer-Moore logic presented in this thesis were added using either the definitional principle, the Defn-Sk principle, or the constrain mechanism. Furthermore, the admissibility of these definitions and constraints was mechanically checked using the extended Boyer-Moore prover. In a similar way, all theorems were validated by the extended Boyer-Moore prover. This guarantees that the resulting logic is a conservative extension of the Boyer-Moore logic, and is therefore sound (at least as sound as the prover).

The Boyer-Moore logic and prover have been used to specify and verify numerous difficult problems including Goedel's Incompleteness Theorem and the Church-Rosser Theorem [81, 80], a microprocessor [45], an assembler [70], and a compiler [87]. The latter three proofs have been formally integrated into a stack of machines [71, 8].

1.3 Related Work

The research directly related to this thesis is focused on the mechanical verification of concurrent programs. Although automatic verification of propositional temporal specifications of finite state machines has been explored for some time [22, 19, 14, 18], most other projects investigating the semi-automatic verification of non-propositional specifications of concurrent systems are relatively recent and were started after this research began [16, 77, 4, 72]. The Gypsy system [34] formalized mechanized proof rules for verifying invariants of message passing systems over ten years ago, however. These will be discussed in the next sections.

This thesis is supported by the broader field of program verification, specifically various logics and proof systems for verifying concurrent programs. Aside from Unity, several such logics are [61, 58, 44, 74]. These are discussed first.

1.3.1 Communicating Sequential Processes

CSP [44, 43], developed by C.A.R. Hoare in 1978, is a programming language for describing synchronous processes. In this paradigm, processes are described by their effect on input and output channels. Communication is synchronous; if one process is ready before the other, the early process waits until communication is possible. Asynchronous communication may be modeled by introducing an unbounded buffer process between the communicating processes [48]; the buffer process is an abstraction of the delay associated with a channel for message passing. CSP has been used to specify a wide variety of systems of varying complexity, including a real communications protocol and delay insensitive circuit components. The development of CSP was influenced by Milner's Calculus of Communicating Systems [66].

In CSP, processes are described by means of the traces they will accept. Two processes are equivalent if the traces they accept are equal. Equivalence may be generalized by stating that the traces are equal if they are restricted to the same alphabet. CSP systems are specified by simpler CSP processes.

CSP also defines a set of operators for combining CSP processes, such as parallel composition, non-deterministic choice, etc. These operators permit processes to be combined easily and encourage the development of large systems from small building block components. The operators are defined axiomatically by laws: for example, parallel composition is both associative and commutative. These laws are the basis for reasoning in CSP: when proving an equality, one would apply the appropriate laws to each side of the equals and unfold certain definitions to transform both sides to the same goal. Some of the laws are basic axioms. Others may be proved just like any CSP equality. This is an algebraic proof technique, so called because of the lack of quantifiers.

CSP has been implemented by the language OCCAM [46], a programming language designed to run on INMOS's Transputer [47]. The Transputer was designed to implement CSP processes by easily reflecting the synchronous communication. Each Transputer is a CSP process and communication occurs over wires connecting the Transputer chips.

1.3.2 An Axiomatic Approach

An early technique for verifying concurrent programs was based on the types of axioms and style that Hoare introduced in his axioms for verifying sequential programs [40] and parallel programs [42]. First introduced by Susan Owicki in her thesis in 1975 [73], it was extended and used in many ways [74, 75, 37, 55, 59].

This approach assumes that statements at some level are atomic, and provides proof rules for proving the correctness of annotated programs. Both safety (partial correctness) and liveness (total correctness, termination) may be proved.

Owicki's original proof system requires the formal treatment of auxiliary variables. Sometimes, is it necessary to add variables to a program to keep track of events so a desired correctness property may be stated and/or proved. Such variables are called auxiliary because they are not needed for the correct operation of the program, only for the proof of the correctness property. This may be due to a weakness in the logic's proof rules or in the specification language (to use history variables, for example).

Owicki's work also emphasized the importance of limiting shared variables and resources in parallel programs, since such sharing introduces complexity and much longer proofs.

It is interesting to note that [74] mentions in conclusion that "although formal proofs are generally too long to be reasonably done by hand, the axiomatic method would be well suited for an interactive program verifier, in which the programmer provides the resource invariants and some of the pre and post assertions, and the program verifier checks that these satisfy the axioms." Unfortunately, fifteen years later, with simpler and better logics and specification languages, such a simple division of labor is still not possible.

1.3.3 Temporal Logic

Aside from CSP, the dominant specification and proof technique for verifying concurrent programs is temporal logic. Introduced by Manna and Pnueli [60, 61], the important observation is that the correctness properties of concurrent programs are easily specified by stating when and how often certain predicates hold over possibly infinitely long program executions.

In the temporal logic approach, concurrent programs are sets of subprograms each of which are sequences of atomic statements. The executions of these subprograms are interleaved in some fair manner. There are many different forms of fairness [23]; Manna and Pnueli provide proof rules for reasoning about strong and weak fairness.

Typical properties proved are invariants, which are specified by the temporal quantifier **`ALWAYS`**, meaning that the subsequent formula holds throughout the execution. This is proved by demonstrating that the desired invariant, or some stronger invariant, holds initially and is preserved by every statement in the program. Liveness properties are proved by demonstrating that some well founded measure exists that is never increased, and is decreased by some program statement. Whether or not that helper statement must be continuously enabled, or simply enabled often enough, depends upon the fairness notions assumed. Liveness properties are stated using the quantifiers **ALWAYS EVENTUALLY**, meaning that the subsequent formula will hold infinitely often throughout the execution. Other properties, such as **UNTIL** and **PRECEDES**, may also be specified and proved.

Temporal logic maps a computational paradigm, the interleaved model of concurrency, to a well understood and very expressive logical framework. Although most properties are specified in a small subset of temporal logic, the soundness and completeness of the logic can be analyzed independently of the sorts of programs being verified.

1.3.4 Temporal Logic of Actions

A new system for verifying concurrent programs is Lamport's Temporal Logic of Actions (TLA) [58]. TLA is a subset of temporal logic and permits the specification of invariance and progress properties. In contrast to Unity, TLA does not build a specific fairness notion into the logic. Instead, several progress proof rules are available and the choice of one or the other indicates which fairness notion is used. Furthermore, in TLA, program statements are logical formulas not unlike those in [39, 49]. Statements are expressions relating the new value of variables (indicated by a prime) to the old value of variables. Although this is often longer than simple multiple assignments, these expressions permit non-deterministic and other incompletely specified statements. Because program statements are logical formulas, no translation is necessary between a programming language and a logical notation. Another advantage is that it is possible to prove that one program implements another via refinement mappings [1].

TLA includes quantification over state variables and is sufficiently expressive to support Lamport's hierarchical method of designing concurrent programs [57].

TLA is being mechanized in LP [24], the theorem prover implementing the Larch shared specification language [38], and mechanical proofs of several programs are in progress. The mechanization of TLA differs from the mechanization of Unity presented here for two reasons: The TLA proof system, although sound, was not encoded onto the prover in a way which guarantees its soundness. Also, proofs are not carried out to first principles. For example, certain facts about integers are assumed. These differences reflect Lamport's priorities as a researcher, since fully verified proofs are costly to develop. However, the reader should keep in mind that cost comparisons between mechanically verified proofs should compare similar goals.

A subset of TLA has also been mechanized in HOL [36] in [86]. This preliminary work focuses on specifying TLA actions and invariant properties, and the proof of invariant properties. The project intends to prove properties about TLA programs, but not prove properties about TLA itself. Work continues on formalizing TLA's liveness properties and refinement mappings in HOL.

1.3.5 Gypsy

Gypsy is a programming and specification language [34]. Hoare type proof rules are defined for every construct in the language, so it is possible to automatically generate the verification conditions necessary to prove that a program satisfies its specification. The Gypsy Verification Environment (GVE) is a computer program that, among other facilities, provides tools for creating these verification conditions, mechanically checking proofs, and managing the proof tree.

Gypsy has been used in many contexts, from design verification, where theorems are proved about a high level design, to code verification, where executable Gypsy code is proved to satisfy a higher level specification. Gypsy was one of the first systems to provide mechanized support for the verification of concurrent programs.

Gypsy uses message passing as its sole form of interprocess communication; this limits the possible interactions of processes and simplifies proofs, as noted in [74]. However, channels may be shared by several senders and receivers. The concurrent operator is **COBEGIN** which initiates a fork, where execution subsequently is fair. Gypsy proof rules may be used to prove invariants of concurrent programs.

The largest distributed system proved in Gypsy controlled an interface between a host possessing sensitive data and a public network [35]. The interface's requirements are that all data sent from the host should be encrypted and all data sent to the host be decrypted. The interface is compatible with the Arpanet and has been demonstrated there. This application contained 4211 lines of executable Gypsy code. The aforementioned requirements were mechanically proved, although two lemmas were assumed. This is a substantial achievement, even by today's standards.

1.3.6 Mechanized Temporal Logic

Russinoff has encoded on the Boyer-Moore prover a subset of Manna and Pnueli's temporal formalism for verifying concurrent programs [77]. This system has been used to mechanically verify invariance (**`ALWAYS`**) and progress (**`ALWAYS EVENTUALLY`**) properties of several non-parameterized concurrent algorithms, including a program that computes binomial coefficients [77] and Ben-Ari's incremental garbage collection algorithm [6, 78], the latter a well known and complicated algorithm.

Russinoff's approach to verification is similar to the one taken here. He defines an arbitrary (weakly) fair execution of a concurrent program and proves theorems about that trace which may be used instead of Manna and Pnueli's proof rules, including the well-founded liveness proof rule [60]. Also, an interpreter is defined which executes Manna and Pnueli type subprograms, updating the subprogram's program counter within the global state and noting the other effects of the atomic execution. This interpreter characterizes input values, which are constant, and program variables, which may be changed.

Russinoff introduces a very clever single axiom which, although sound, is not provably sound in this logic. This axiom introduces a function **`N`** which permits one to prove progress properties without defining a clock function that computes when the property will eventually hold in the arbitrary fair execution. **`N`** is axiomatized to represent that point, if it exists.

This system also defines a front end that permits an easily read specification style. Although the formulas are still in prefix form (Lisp like), certain combinations of temporal operators may be specified; these formulas are translated into a pure Boyer-Moore logic formula before being proved. The front end also allows the user to specify proof hints to the prover in a sensible fashion, including which proof rules ought to be used, with what measures, etc. The drawback is that it is easy to forget both that the printed formula is not what the theorem prover deals with, and what the actual translation is. This is especially important if a proof is not completed automatically, since one may have to interact with the prover during a proof.

1.3.7 CSP in HOL

Parts of CSP were encoded in HOL [36] in [16]. HOL is a theorem prover for Higher Order Logic, a fully quantified and typed logic that permits quantification over functions. This logic is more expressive than the Boyer-Moore logic. The HOL prover, however, has fewer built in heuristics for automatically attempting to prove a theorem. Instead, users define tactics which define a generally useful reasoning process. This approach produces proofs which look more like formal proofs; the corresponding claim in the Boyer-Moore prover is that if the prover certifies a proof, then a formal proof exists. HOL has been used to check numerous proofs, perhaps the most significant is the verification of parts of the VIPER microprocessor [20].

Camilleri's approach to respecifying CSP takes advantage of CSP trace-based semantics. Traces are sequences of elements from a finite alphabet. Therefore, it is possible to define CSP's operators, like restriction and interleaving, as functions of traces. Instead of using CSP's basic axioms as the definitions of the operators, these laws are proved as theorems about the defined operators. This careful approach combines the soundness obtained by relying upon a model, with the ease-of-use obtained from proof rules.

This encoding also successfully investigated a difficult problem, the formalization of CSP's fixed point semantics. It provides a proof system for a subset of CSP. Unfortunately, however, this mechanical support was not demonstrated on any CSP programs.

1.3.8 Unity in HOL

Fleming Andersen has encoded most of Unity proofs rules in HOL in an interesting manner [4]. The methodological details are sketchy, however, since the report is still being written.

In any case, this formalization in HOL is not supported by an execution-based semantics. Rather, Unity's **LEADS-TO** relation was characterized directly in terms of the three defining axioms in [17]. Another model was used to demonstrate soundness. Using these three axioms, Unity's proof rules were proved again, using structural induction.

The most difficult proof, apparently, was to demonstrate that the two versions of Unity's structural induction principle are equivalent. These versions differ in their second axiom:

```
P LEADS-TO Q
Q LEADS-TO R
------------
P LEADS-TO R
```

The alternative is:

```
P ENSURES Q
Q LEADS-TO R
------------
P LEADS-TO R
```

Since **LEADS-TO** is a consequence of **ENSURES** in both versions of Unity's structural induction principle, it is obvious that any proof using the alternative structural induction may be translated into a proof using the original structural induction. The opposite is more complicated but was also completed.

Andersen's work focuses on the soundness and completeness of Unity but does not provide a facility for verifying real Unity programs.

1.3.9 Mutual Exclusion on the Boyer-Moore Prover

Nagayama and Talcott [72] have successfully verified the invariance property of an algorithm that was posed by Amir Pnueli as a challenge for mechanical verification [62]. The algorithm, when followed by each of **N** indexed processes, prevents more than one process from being in its critical section at a time. The invariant property proved, therefore, is that at most one process is critical at a time. There are several notable facts about the correctness proof: This program is parameterized because it specifies an algorithm for an arbitrary number of processes. Two variants of the same algorithm were proved, each assuming a different level of atomicity. In order to prove the desired correctness property, a stronger invariant was proved first. Finally, this proof was accomplished within the unextended Boyer-Moore logic, as described in [9].[5]

Since only an invariance property was being considered, no notion of execution was defined. Rather, the authors demonstrated that the stronger invariant was preserved by every possible transition in the program. The difficulty was mainly one of stating each lemma accurately, and coaxing the proofs through the prover.

[5] This work did not use the interpreter functions or the bounded quantification of Nqthm.

Chapter 2
AN OPERATIONAL SEMANTICS

The operational semantics of concurrency used here is based on the transition system model [61, 17]. A *transition system* is a set of statements each of which effects transitions (changes) on the system state. A *computation* is the sequence of states generated by the composition of an infinite sequence of transitions on an initial state. *Fairness* notions are restrictions of the scheduling of statements in the computation. For example, if every program statement is a total function, then *unconditional fairness* requires that each statement be responsible for an infinite number of transitions in the computation (every statement is scheduled infinitely often). Other fairness notions introduce the concept of *enabled* transitions, where a statement can only effect a transition if it is enabled (the statement can produce a successor state). These notions will be formalized in section 3.3. Stronger fairness notions restrict the set of computations that a program may generate; hence a program's behavior may be correct under one fairness notion and not under another.

The next sections present an operational characterization of an arbitrary computation.

2.1 A Concurrent Program

A program is a list of statements. Each statement is a relation from previous states to next states. We define the function **`N`** so the term **`(N OLD NEW E)`** is true if and only if **`NEW`** is a possible successor state to **`OLD`** under the transition specified by statement **`E`**. The actual definition of **`N`** is not important until one considers a particular program. For completeness, however, the definition of **`N`** is:

Definition:

```
(N OLD NEW E)
   =
(APPLY$ (CAR E) (APPEND (LIST OLD NEW) (CDR E)))
```

`N` applies the **`CAR`** of the statement to the previous and next states, along with any other arguments encoded into the **`CDR`** of the statement. A state can be any data structure. Intuitively, a statement is a list with the first component being a function name, and the remainder of the list being other arguments. These arguments may instantiate a function representing a generic statement to a specific program statement. This encoding provides a convenient way to specify programs containing many similar statements which differ only by an index or some other parameter.

A statement **`E`** is *enabled* in state **`OLD`** if there exists some state **`NEW`** such that **`(N OLD NEW E)`** is true. That is, a statement is enabled if it can produce a successor state. We call such transitions *effective*. If a statement cannot effect any effective transitions from state **`OLD`** then it is *disabled* for that state. The *enabling* condition for a statement is the weakest precondition guaranteeing an effective transition. A statement's effective transition may be the identity transition, however (e.g., the **`SKIP`** statement).

2.2 A Computation

We now characterize a function, named **S**, representing an arbitrary, but fixed computation. The execution of a concurrent program is an interleaving of statements in the program. This characterization of **S** requires that every statement be scheduled infinitely often. Disabled statements effect the identity transition. This formalization is equivalent to weak fairness. Furthermore, if all program statements are total functions, this reduces to unconditional fairness.

Introducing extra skip states can be considered stuttering and is legitimate since repeated states do not interfere with either the safety or liveness properties discussed in section 3.1. Fairness notions presented later will guarantee that a statement eventually executes an effective transition.

The term **(S PRG I)** represents the **I**'th state in the execution of program **PRG**. The function **S** is characterized by the following two constraints specifying the relationship between successive states in a computation:

Constraint: S-Effective-Transition[6]

```
(IMPLIES (AND (LISTP PRG)
              (∃ NEW (N (S PRG I) NEW (CHOOSE PRG I))))
         (N (S PRG I)
            (S PRG (ADD1 I))
            (CHOOSE PRG I)))
```

This constraint states that, given two assumptions, the state **(S PRG (ADD1 I))** is a successor state to **(S PRG I)** and the statement governing that transition is chosen by the function **CHOOSE** in the term **(CHOOSE PRG I)**. **CHOOSE** is a fair scheduler. Additional constraints about **CHOOSE** will be presented later.

The two assumptions are:

- The program must be non-empty. This is stated by the term **(LISTP PRG)**. If the program has no statements, then no execution may be deduced.
- There is some successor state from **(S PRG I)** under the statement scheduled by **(CHOOSE PRG I)**. (If the statement is disabled, no effective transition is possible. The next constraint specifies that the null transition occurs in this case.)

The second constraint specifies the relationship between successive states when the scheduled statement is disabled:

[6] This constraint is equivalent to the following unquantified formula, because the existential may be moved outside the formula.

```
(IMPLIES (AND (LISTP PRG)
              (N (S PRG I) NEW (CHOOSE PRG I)))
         (N (S PRG I)
            (S PRG (ADD1 I))
            (CHOOSE PRG I)))
```

Indeed, the unquantified formula is used in the mechanization since it is easier to formalize in the Boyer-Moore logic. However, the quantified formula is simpler for exposition.

Constraint: S-Idle-Transition[7]

```
(IMPLIES (AND (LISTP PRG)
              (NOT (∃ NEW (N (S PRG I) NEW
                             (CHOOSE PRG I)))))
         (EQUAL (S PRG (ADD1 I))
                (S PRG I)))
```

This constraint states that if a disabled statement is scheduled, then no progress is made (i.e., a skip statement is executed instead).

2.3 The Scheduler

The function **CHOOSE** is a scheduler. It is characterized by the following constraints:
Constraint: Choose-Chooses

```
(IMPLIES (LISTP PRG)
         (MEMBER (CHOOSE PRG I) PRG))
```

This constraint states that **CHOOSE** schedules statements from the non-empty program **PRG**.

We now guarantee that every statement is scheduled infinitely often. We do this without regard for enabled or disabled statements; effective transitions will be guaranteed by subsequent fairness notions.

Scheduling every statement infinitely often is equivalent to always scheduling each statement again. This property is specified by the function **NEXT** and its relationship to **CHOOSE**:
Constraint: Next-Is-At-Or-After

```
(IMPLIES (MEMBER E PRG)
         (NOT (LESSP (NEXT PRG E I) I)))
```

This constraint states that for statements in the program, **NEXT** returns a value at or after **I**. Furthermore, **(NEXT PRG E I)** returns a future point in the schedule when statement **E** is scheduled:
Constraint: Choose-Next

```
(IMPLIES (MEMBER E PRG)
         (EQUAL (CHOOSE PRG (NEXT PRG E I))
                E))
```

There are four constraints that coerce non-numeric index arguments to **0** and identify **NEXT**'s type as numeric. These are presented as a conjunction:

---

[7]This constraint is equivalent to the following unquantified formula, by introducing a Skolem function **(NEWX E OLD)** which returns a successor state to **OLD** for statement **E** if possible. To prove that the null transition is effected, one must prove that **NEWX** is not a successor state. Since one knows nothing about **NEWX**, this is equivalent to demonstrating that no successor state exists. This formula is the one used in the formalization:

```
(IMPLIES (AND (LISTP PRG)
              (NOT (N (S PRG I)
                      (NEWX (CHOOSE PRG I)
                            (S PRG I))
                      (CHOOSE PRG I))))
         (EQUAL (S PRG (ADD1 I))
                (S PRG I)))
```

Constraint: Index-Is-Numeric

```
(AND (IMPLIES (MEMBER E PRG)
              (NUMBERP (NEXT PRG E I)))
     (IMPLIES (AND (LISTP PRG)
                   (NOT (NUMBERP I)))
              (EQUAL (S PRG I)
                     (S PRG 0)))
     (IMPLIES (AND (LISTP PRG)
                   (NOT (NUMBERP I)))
              (EQUAL (CHOOSE PRG I)
                     (CHOOSE PRG 0)))
     (IMPLIES (AND (MEMBER E PRG)
                   (NOT (NUMBERP I)))
              (EQUAL (NEXT PRG E I)
                     (NEXT PRG E 0))))
```

This set is somewhat redundant, since it is unnecessary to type **NEXT** as a number if the index argument in each of **S**, **CHOOSE**, and **NEXT** are coerced to numbers. However, the extra constraint does eliminate considering the non-number case, when reasoning about **NEXT**.

This completes the definition of the operational semantics of concurrency. Since **S**, **CHOOSE**, and **NEXT** are characterized only by the constraints listed above, **S** defines an arbitrary computation of a concurrent program. **S** guarantees that every statement will be scheduled infinitely often; transitions need not be effective. Statements proved about **S** are true for any fair computation.[8] So theorems in which **PRG** is a free variable are really proof rules, and this is the focus of the next chapter in section 3.2.

2.4 Soundness

The constraints presented in the previous section are axioms about the undefined function symbols **S**, **CHOOSE**, and **NEXT**. Since these nine axioms were introduced using the functional instantiation enhancement described earlier (section 1.2.3), they do not introduce any unsoundness into the theory. This is guaranteed by the support of witness functions that satisfy the constraints. As stated earlier, these witness functions do not have any logical connection to the functions they are modeling. It is instructive, however, to describe the witness functions, for two reasons:

- The witnesses provide an execution model for the constrained symbols and may provide additional intuition into the correctness of the constraints. This execution model is not executable, however, since the statements are relations.
- The construction of these witnesses and the statement of the constraints, especially the encoding of the existential quantifier in the **S-Idle-Transition** constraint by the Skolem function **NEWX**, were non-trivial and may be helpful in other formalizations.

A good model for the fair scheduler characterized by **CHOOSE** and **NEXT** is a round-robin type scheduler. We define the witness function **MCHOOSE** as a model for **CHOOSE**. **(MCHOOSE PRG I)** returns the statement in **PRG** scheduled at the **I**'th point in the computation:

Definition:

```
(MCHOOSE PRG I)
   =
(NTH PRG (REMAINDER I (LENGTH PRG)))
```

where **(NTH LIST N)** returns the **N+1**'th element in the list **LIST**. **(REMAINDER I J)** returns the remainder of **I** divided by **J**.

The complementary function **MNEXT** serves as a witness for **NEXT**. **(MNEXT PRG E I)** returns some position in the schedule at least as far along as **I** such that **E** is scheduled at that point. (Of course, **E** must be a member of the program **PRG**.) **MNEXT** is defined as follows:

---

[8] That is, **S**, **CHOOSE**, and **NEXT** are constrained function symbols, and theorems proved about them can be instantiated with terms representing any computation also satisfying those constraints.

Definition:

```
(MNEXT PRG E I)
   =
(PLUS I
      (IF (LESSP (POSITION PRG E)
                 (REMAINDER I (LENGTH PRG)))
          (PLUS (POSITION PRG E)
                (DIFFERENCE (LENGTH PRG)
                            (REMAINDER I
                                       (LENGTH PRG))))
        (DIFFERENCE (POSITION PRG E)
                    (REMAINDER I (LENGTH PRG)))))
```

(POSITION LIST E) returns the position of the first occurrence of **E** in the list **LIST**, where the position of the first element is **0**. **DIFFERENCE** is the subtraction operator for naturals. **MNEXT** works in the following way: Imagine that a schedule is the repeated concatenation of the list **PRG**. Any **E** in **PRG** occurs in the schedule at least once every **(LENGTH PRG)** elements (**PRG** is not necessarily a set). If index **I** is past the first occurrence of **E** in this repetition, then **MNEXT** returns the index of the appropriate occurrence in the next repetition. However, if index **I** precedes the first occurrence of **E** in this repetition, **MNEXT** returns that index.

At this point, one may prove that the combination of **MCHOOSE** and **MNEXT** satisfy the axioms mentioning only **CHOOSE** and **NEXT**. For example, the analog of **Choose-Next** is:
Theorem: Mnext-Choice-2

```
(IMPLIES (MEMBER E PRG)
         (EQUAL (MCHOOSE PRG (MNEXT PRG E I)) E))
```

Analogs of the other axioms are stated in a similar way.

The witness function for the computation **S** is **MS**. **MS** would naturally be defined in the following way: Starting from some initial state, apply the statement scheduled by **MCHOOSE** at index **0**, yielding the second state in the computation. This is repeated indefinitely. The difficulty is that statements in this model are defined as relations and not functions, and may not simply be applied to a state to determine the successor state. At first, this appears to be an insurmountable problem. But, it is solved in the following way: Remember that **S** only requires the generation of a successor state if some successor state exists. (Otherwise, the predecessor is repeated.) Therefore, we define the function **Exists-Successor**, where the term **(EXISTS-SUCCESSOR OLD E)** is true if and only if statement **E** permits some successor to state **OLD**. **Exists-Successor** is defined using the Defn-Sk extension (section 1.2.4) as follows:[9]
Definition:

```
(EXISTS-SUCCESSOR OLD E)
   ⇔
(EXISTS NEW (N OLD NEW E))
```

Remember, that this definition is really encoded in the logic as two Skolemized formulas. These are:

```
    (AND (IMPLIES (EXISTS-SUCCESSOR OLD E)
                  (N OLD (NEWX E OLD) E))
         (IMPLIES (N OLD NEW E)
                  (EXISTS-SUCCESSOR OLD E)))
```

The Skolem function introduced is **NEWX**, which is the same function used in constraint **S-Idle-Transition** (section 2.2). What does the term **(NEWX E OLD)** mean? The first conjunct in the formula implies that **(NEWX E OLD)** represents a successor state to **OLD** for statement **E**. But this is only true if **(EXISTS-SUCCESSOR OLD E)** is true. That term is proved by appealing to the second conjunct: to satisfy the hypothesis there, one must pick some instance of **NEW** such that **(N OLD NEW E)** is true.

---

[9]A prover directive (not show here) is included with this definition, indicating that the names of Skolem functions introduced by this definition in place of existentially quantified variables, should be based upon those variable names with the suffix **X** added. Hence, **NEW** is replaced by the Skolem function symbol **NEWX**. **NEWX**'s formal parameters are the universally quantified variables upon which the existentially quantified **NEW** depends. These parameters are ordered alphabetically.

This arrangement produces the following situation, which is perfect for this formalization: if one can demonstrate that a successor state exists, then **`(NEWX E OLD)`** represents some successor state (not necessarily the same one used to demonstrate existence). Therefore, the term **`(NEWX E OLD)`** can be used in two ways: to generate some successor state, if one exists, and as a hypothesis in theorems depending upon the existence of a successor state, since the latter would be proved by demonstrating **`(EXISTS-SUCCESSOR OLD E)`** and consequently the existence of some **`NEW`** satisfying **`(N OLD NEW E)`**. The second way was used in the constraint **S-Idle-Transition**. The first way is used in the definition of **`MS`**, the model of **`S`**:

Definition:

```
(MS PRG I)
   =
(IF (ZEROP I)
    NIL
  (IF (N (MS PRG (SUB1 I))
         (NEWX (MCHOOSE PRG (SUB1 I))
               (MS PRG (SUB1 I)))
         (MCHOOSE PRG (SUB1 I)))
      (NEWX (MCHOOSE PRG (SUB1 I))
            (MS PRG (SUB1 I)))
    (MS PRG (SUB1 I))))
```

`MS` is defined recursively in the following way: **`(MS PRG 0)`** returns **`NIL`**, an arbitrarily fixed initial state. Subsequent states are generated from their predecessors. If the scheduled statement permits a successor state, that successor is generated. If, however, no successor state is defined, then the predecessor is repeated. **`MS`** satisfies the axioms about **`S`**. Here are the analogs of **S-Effective-Transition** and **S-Idle-Transition**:

Theorem: Ms-Transition-Successful

```
(IMPLIES (AND (LISTP PRG)
              (N (MS PRG I) NEW (MCHOOSE PRG I)))
         (N (MS PRG I)
            (MS PRG (ADD1 I))
            (MCHOOSE PRG I)))
```

Theorem: Ms-Transition-Idle

```
(IMPLIES (AND (LISTP PRG)
              (NOT (N (MS PRG I)
                      (NEWX (MCHOOSE PRG I)
                            (MS PRG I))
                      (MCHOOSE PRG I))))
         (EQUAL (MS PRG (ADD1 I))
                (MS PRG I)))
```

Once **`MS`**, **`MCHOOSE`**, and **`MNEXT`** are defined and these and other theorems proved about them, the constrain event introducing all nine axioms presented earlier may be submitted to the theorem prover. The axioms are presented in this text separately, for ease of reference, but must, logically, be introduced simultaneously, since satisfying the constraints may only be done if all three undefined function symbols are replaced by their witnesses.

Chapter 3

THE PROOF SYSTEM

The previous chapter (chapter 2) characterized an arbitrary fair computation of a concurrent program. That formalization is sufficient for reasoning about program correctness. In fact, it is (relatively) complete, since the constraints provide all relevant information about a fair computation. However, it is useful to abstract away from sequences of states and deduce correctness properties directly from the program text using a set of proof rules. These proof rules are theorems about the arbitrary fair computation. In this methodology, there are two forms of reasoning: the deduction of basic correctness properties as consequences of individual program statements; and the combination of more basic properties into more complicated ones.

This chapter presents proof rules similar to those found in Unity for reasoning about both safety and liveness properties. Additionally, it presents proof rules for deducing basic liveness properties for the three fairness notions of unconditional, weak, and strong fairness, and for the safety notion of deadlock freedom.

Before formalizing these proof rules, we must define predicates for specifying correctness properties. Proof rules will be theorems permitting the proofs of correctness properties.

3.1 Specification Predicates

The Unity logic permits reasoning about the safety and liveness (progress) properties of concurrent programs. Safety properties are those that state that something bad will never happen [3]; examples are invariant properties such as mutual exclusion and freedom from deadlock. Liveness properties state that something good will eventually happen [2]; examples are termination and freedom from starvation. We borrow Unity's predicates for safety (**UNLESS**) and liveness (**LEADS-TO**) and present the definitions of these predicates in the context of this proof system.

3.1.1 Eval

Specification predicates take other predicates as arguments and evaluate those predicates on either computation states or arbitrary (not necessarily reachable) states. Therefore, it is necessary to define a function that will evaluate a quoted term with respect to some environment. We now define the function **EVAL**, using Nqthm's built-in interpreter function **EVAL$**, which was described earlier (section 1.2.2). **(EVAL PRED STATE)** evaluates the formula **PRED** in the context of the state **STATE** and is defined as follows:

Definition:

```
(EVAL PRED STATE)
   =
(EVAL$ T PRED (LIST (CONS 'STATE STATE)))
```

When **EVAL** is used, the formula must use **'STATE** as the name of the ‘‘variable’’ representing the state. Notice that **EVAL** has the expected property:

Theorem: Eval-Or

```
(EQUAL (EVAL (LIST 'OR P Q) STATE)
       (OR (EVAL P STATE)
           (EVAL Q STATE)))
```

That is, **EVAL** distributes over **OR**. Similarly, **EVAL** distributes over the other logical connectives and it interprets other total functions in the obvious way.[10]

3.1.2 Unless

Unity's predicate for specifying safety properties is **UNLESS**. The definition of **UNLESS** here is:
Definition:

```
(UNLESS P Q PRG)
   ⇔
(FORALL (OLD NEW E)
        (IMPLIES (AND (MEMBER E PRG)
                      (N OLD NEW E)
                      (EVAL (LIST 'AND P (LIST 'NOT Q))
                            OLD))
                 (EVAL (LIST 'OR P Q) NEW)))
```

(UNLESS P Q PRG) states that every statement in the program **PRG** takes states where **P** holds and **Q** does not hold to states where **P** or **Q** holds. Intuitively, this means that once **P** holds in a computation, it continues to hold (it is stable), at least until **Q** holds (this may occur immediately). A subtle point is that if the precondition **P** disables some statement, then **UNLESS** holds vacuously for that statement. This is consistent with the operational semantics presented earlier, since a disabled statement, if scheduled, will effect the null transition. Hence, the successor state will be identical to the previous state and the precondition **P** will be preserved.

Notice that if **(UNLESS P '(FALSE) PRG)** is true for program **PRG** (that is **P** is a stable property) and **P** holds on the initial state (e.g., **(EVAL P (S PRG 0))**), then **P** is an invariant of **PRG** (that is, **P** is true of every state in the computation). In Unity, this implication is an equivalence: **P UNLESS FALSE** is true for every invariant. This is because Unity's **UNLESS** is defined with respect to the computation (reachable states). This definition has its advantages, but it complicates the precise statement of the union theorems (section 3.4.1). Instead, this **UNLESS** is not restricted to reachable states in the computation, but another predicate **INVARIANT** (section 3.1.4) is defined independently and is identical to Unity's definition of **INVARIANT** which is defined in terms of Unity's **UNLESS**. In addition, the definition of an **UNLESS** type property which is unrelated to the computation is key to the formalization of the safety notion of deadlock freedom.

3.1.3 Leads-To

LEADS-TO is the general progress predicate. It is defined as follows:[11]

---

[10]The definition of **EVAL** in terms of the interpreter function **EVAL$** restricts correctness properties provable here to partial recursive ones. In practice, this does not appear to be a problem, since quantification is usually over some finite domain. It is often possible to quantify over an infinite domain, if the predicate can be shown to be partial recursive anyway. This is demonstrated in the mutual exclusion program where we rely on the fact that the counter has a value, even though that value may be any natural number. This makes the quantified term *is the counter equal to some i* equal to true, even though the quantification is unbounded (and true is certainly partial recursive). It would have been possible to constrain **EVAL** to depend only upon the proper interpretation of the logical connectives (and not be restricted to partial recursive functions), but although that approach would have better characterized Unity, it would have required the instantiation of the entire proof system for a useful **EVAL** before proving any real program.

[11]A prover directive (not shown here) is included with this definition, indicating that the names of Skolem variables introduced by this definition should be derived from the existentially quantified variable they replace, with the suffix **LEADS** added.

Definition:

```
(LEADS-TO P Q PRG)
   ⇔
(FORALL I (IMPLIES (EVAL P (S PRG I))
                   (EXISTS J
                           (AND (NOT (LESSP J I))
                                (EVAL Q (S PRG J))))))
```

(LEADS-TO P Q PRG) states that if **P** holds at some point in a computation of program **PRG**, then **Q** holds at that point or at some later point in the computation.

Unity's theorems about **LEADS-TO** are theorems in this proof system as well. In the statement of several of these theorems, it is necessary to use the Skolem functions introduced by the definition of **LEADS-TO**. The two axioms resulting from the Skolemization of **LEADS-TO** are:

```
    (AND (IMPLIES (IMPLIES (EVAL P (S PRG (ILEADS P PRG Q)))
                           (AND (NOT (LESSP J (ILEADS P PRG
                                                          Q)))
                                (EVAL Q (S PRG J))))
                  (LEADS-TO P Q PRG))
         (IMPLIES (AND (LEADS-TO P Q PRG)
                       (EVAL P (S PRG I)))
                  (AND (NOT (LESSP (JLEADS I PRG Q) I))
                       (EVAL Q (S PRG (JLEADS I PRG Q))))))
```

The second conjunct states that if **LEADS-TO** is true and **P** holds at some state, then **Q** holds at some later state which may be identified using the function **JLEADS**. Notice that the function **JLEADS** replaces the existential **EXISTS J** in the definition of **LEADS-TO**. This conjunct is used to derive consequences of **LEADS-TO**.

The first conjunct states that **LEADS-TO** is true if, for an arbitrary starting point in the computation at which **P** holds, one can find a later **J** at which **Q** holds. The function **ILEADS** serves to fix the arbitrary point and it replaces the universal **FORALL I** in the definition of **LEADS-TO**. This conjunct is used to prove **LEADS-TO**. It is important to note, however, that understanding the Skolemization is only necessary to understand the statement and proofs of several of the proof rules. This same understanding is not needed for the use of those proof rules.

Curiously, Unity's **LEADS-TO** can be used to specify invariance properties.[12] **(LEADS-TO P '(FALSE) PRG)** implies that the negation of **P** is invariant (section 3.2.2). This is deduced by contradiction: if **P** does hold at some point in the computation, then **'(FALSE)** would have to hold subsequently, which is impossible. Notice, that **(LEADS-TO P '(FALSE) PRG)** is only concerned with reachable states in the computation: it does not imply **(UNLESS (LIST 'NOT P) '(FALSE) PRG)**, even though **(LIST 'NOT P)** evaluates to the negation of **P**. (See section 3.1.2.)

3.1.4 Invariance Properties

Invariants are properties that are preserved throughout the computation. Invariants are specified using the term **(INVARIANT INV PRG)** which is defined as follows:

Definition:

```
(INVARIANT INV PRG)
   ⇔
(FORALL I (EVAL INV (S PRG I)))
```

(INVARIANT INV PRG) is true only if **INV** holds on every state in the computation. Often, it is proved by assuming that **INV** holds initially and proving **(UNLESS INV '(FALSE) PRG)**. Also, if **INV** is a consequence of any other invariant, then **(INVARIANT INV PRG)** is true as well. This is the key difference between specifying **(INVARIANT INV PRG)** and **(UNLESS INV '(FALSE) PRG)**.

[12] This is obvious in temporal logic.

Initial conditions are postulated using the predicate **INITIAL-CONDITION** which is defined as follows:
Definition:

```
(INITIAL-CONDITION IC PRG)
   =
(EVAL IC (S PRG 0))
```

Stating **(INITIAL-CONDITION IC PRG)** in the hypothesis of a theorem implies that **IC** holds on the initial state.

3.1.5 Eventual Invariance

The next predicate specifies properties that are eventually invariant. Such properties are important, as they represent generalizations of the notion of fixed points in the computation. (Unity specifies such properties using **UNLESS** and **LEADS-TO** and adding auxiliary variables to the program [67].) The predicate **EVENTUALLY-INVARIANT** is defined as follows:
Definition:

```
(EVENTUALLY-INVARIANT R PRG)
   ⇔
(EXISTS I (FORALL J (IMPLIES (NOT (LESSP J I))
                             (EVAL R (S PRG J)))))
```

(EVENTUALLY-INVARIANT R PRG) states that there exists a point in the computation, at and after which **R** holds continuously.

As in **LEADS-TO**, many theorems about **EVENTUALLY-INVARIANT** require the Skolem functions introduced by the definition of **EVENTUALLY-INVARIANT**. The two axioms resulting from the Skolemization of **EVENTUALLY-INVARIANT** are:

```
    (AND (IMPLIES (IMPLIES (NOT (LESSP (JES I PRG R) I))
                           (EVAL R (S PRG (JES I PRG R))))
                  (EVENTUALLY-INVARIANT R PRG))
         (IMPLIES (AND (EVENTUALLY-INVARIANT R PRG)
                       (NOT (LESSP J (IES PRG R))))
                  (EVAL R (S PRG J))))
```

Many other predicates may be defined and their associated proof rules proved. However, this set (along with **ENSURES** which is defined later) is sufficient for the types of properties we are concerned with here.

3.2 Proof Rules

Proof rules facilitate the proof of program properties in much the same way that lemmas aid a mathematical proof. In fact, the proof rules presented here are theorems about computations. Some of the theorems are not stated in the most general way possible because they are more easily used in their current form.

3.2.1 Liveness Theorems

LEADS-TO is transitive. This property is especially important since it can be applied repeatedly using the Boyer-Moore logic's induction principle.
Theorem: Leads-To-Transitive

```
(IMPLIES (AND (LEADS-TO P Q PRG)
              (LEADS-TO Q R PRG))
         (LEADS-TO P R PRG))
```

The next two theorems demonstrate how the beginning and ending predicates in **LEADS-TO** can be manipulated. Just like an implication, the beginning predicate can be strengthened and the ending predicate can be weakened.

Theorem: Leads-To-Strengthen-Left

```
(IMPLIES (AND (IMPLIES (EVAL Q (S PRG
                                  (ILEADS Q PRG R)))
                       (EVAL P (S PRG
                                  (ILEADS Q PRG R))))
              (LEADS-TO P R PRG))
         (LEADS-TO Q R PRG))
```

This theorem states that if **(LEADS-TO P R PRG)** holds, and **Q** is stronger than **P**, then one can deduce **(LEADS-TO Q R PRG)**. Since the obvious hypothesis, ∀ **STATE (IMPLIES (EVAL Q STATE) (EVAL P STATE))**, stating that **Q** is stronger than **P** cannot be stated (easily) in the Boyer-Moore logic, we must use a term that is implied by this hypothesis and still makes this statement a theorem. Such a term is obtained by taking advantage of the arbitrary initial point in the computation, using the function **ILEADS** (section 3.1.3, page 23). Notice that when using this theorem, one must still prove that **Q** is stronger than **P**, and that one may assume any program invariants about **(S PRG (ILEADS Q PRG R))**, since it is a computation state. The next theorem states that the ending predicate can be weakened, using the function **JLEADS**.

Theorem: Leads-To-Weaken-Right

```
(IMPLIES (AND (IMPLIES (EVAL Q (S PRG
                                  (JLEADS
                                   (ILEADS P PRG R)
                                   PRG Q)))
                       (EVAL R (S PRG
                                  (JLEADS
                                   (ILEADS P PRG R)
                                   PRG Q))))
              (LEADS-TO P Q PRG))
         (LEADS-TO P R PRG))
```

Leads-To-Strengthen-Left can also be used in place of Unity's infinite disjunction proof rule for **LEADS-TO**. This is because one can take advantage of the arbitrary point at which **Q** must be stronger than **P** and then pick the appropriate **P**. As an example, assume that **P** is a predicate indexed by **I**, and let **Q** be ∃ **I P(I)**, the infinite disjunction of **P** over all **I**. The goal is to prove that **Q** leads to some **R** given that **P(I)** leads to that same **R** for every **I**. This is proved by applying the **Leads-To-Strengthen-Left** proof rule, instantiating **I** to the particular value that makes the infinite disjunction true at the arbitrary point in the computation **(S PRG (ILEADS Q PRG R))**.[13]

The next theorem provides one method of proving a disjunction of beginning predicates: simply prove **LEADS-TO** for each one. The analogous theorem for ending predicates is a simple consequence of **Leads-To-Weaken-Right**; the complementary statement (using **AND**) for ending predicates is false.

Theorem: Disjoin-Left

```
(IMPLIES (AND (LEADS-TO P R PRG)
              (LEADS-TO Q R PRG))
         (LEADS-TO (LIST 'OR P Q) R PRG))
```

The cancellation theorem is a twist on transitivity. **Leads-To-Weaken-Right** is often used prior to this theorem when it is necessary to commute the term **(LIST 'OR Q B)** to **(LIST 'OR B Q)**.

Theorem: Cancellation-Leads-To

```
(IMPLIES (AND (LEADS-TO P (LIST 'OR Q B) PRG)
              (LEADS-TO B R PRG))
         (LEADS-TO P (LIST 'OR Q R) PRG))
```

The next proof rule demonstrates that an invariant is preserved throughout a computation. This proof rule is included in this section of liveness proof rules, because it is most often used to delete or add invariants in conjunction with the **Leads-To-Strengthen-Left** and **Leads-To-Weaken-Right** proof rules.

---

[13] Of course, the predicates being manipulated must be partial recursive ones, and the use of quantifiers in this discussion is only for exposition.

Theorem: Invariant-Implies

```
(IMPLIES (INVARIANT INV PRG)
         (EVAL INV (S PRG I)))
```

Using this proof rule and the **LEADS-TO** weakening and strengthening proof rules presented earlier, it is possible to add or remove an invariant, or any consequence of an invariant, to or from the beginning or ending predicate in a **LEADS-TO**. As an example, assume that **(LEADS-TO (LIST 'AND Q INV) R PRG)** is true and **INV** is an invariant of the program **PRG**. We wish to conclude that **(LEADS-TO Q R PRG)** is true as well. The latter property is obviously true by the following argument: if we are at a state satisfying **Q** and **INV** is an invariant of the program, then that state satisfies **(LIST 'AND Q INV)**. This makes the two **LEADS-TO** statements identical. Using the proof rules, this conclusion is proved by applying the theorem **Leads-To-Strengthen-Left** in the following way: let **Q** be **Q** and let **P** be **(LIST 'AND Q INV)**. The hypothesis requires that **Q** imply **(LIST 'AND Q INV)** when each is evaluated in the context of the state **(S PRG (ILEADS Q PRG R))**. **Q** holds trivially; it remains necessary to show that **INV** holds at that point. Appealing to the theorem **Invariant-Implies**, we see that if **INV** is an invariant, it holds on any other state in the computation. This satisfies the second proof obligation, so we may conclude that **(LEADS-TO Q R PRG)** is true. This use of the two theorems **Invariant-Implies** and **Leads-To-Strengthen-Left** (or **Leads-To-Weaken-Right**) replaces Unity's substitution axiom. In this proof system, Unity's substitution axiom only applies to computation properties [79, 68]; these are **LEADS-TO**, **INVARIANT**, and **EVENTUALLY-INVARIANT**.

The last theorem of this section, the **PSP** theorem, combines a progress and a safety property to yield a progress property [17].

Theorem: Psp

```
(IMPLIES (AND (LEADS-TO P Q PRG)
              (UNLESS R B PRG)
              (LISTP PRG))
         (LEADS-TO (LIST 'AND P R)
                   (LIST 'OR (LIST 'AND Q R) B)
                   PRG))
```

This theorem is proved by induction on the computation. Intuitively, if some state satisfies both **P** and **R**, the **UNLESS** hypothesis states that **R** holds until **B** holds; furthermore, **Q** holds eventually. The only question is which of **Q** or **B** is reached first. This theorem also illustrates that whenever a computation property is deduced from a program property (**UNLESS** and **ENSURES**), the program must be non-empty.

3.2.2 Safety Theorems

This section presents theorems useful for proving invariants about the computation. Usually, invariants hold only if some initial condition is satisfied; therefore, many of these theorems include an **INITIAL-CONDITION** assumption as a hypothesis. The first theorem proves an **INVARIANT** property from an **UNLESS** property:

Theorem: Unless-Proves-Invariant

```
(IMPLIES (AND (INITIAL-CONDITION IC PRG)
              (UNLESS P '(FALSE) PRG)
              (IMPLIES (EVAL IC (S PRG 0))
                       (EVAL P (S PRG 0)))
              (LISTP PRG))
         (INVARIANT P PRG))
```

The intuition behind this theorem is that if **(UNLESS P '(FALSE) PRG)** is true, then we know that **P** persists once it holds. **(INITIAL-CONDITION IC PRG)** implies that **IC** holds initially, and the third hypothesis implies that **P** holds initially as well. Therefore, **P** holds throughout the computation and is an invariant of **PRG**.

Another useful theorem permits the weakening of an invariant. If **P** is an invariant of program **PRG**, so is any consequence of **P**:

Theorem: Invariant-Consequence

```
(IMPLIES (AND (INVARIANT P PRG)
              (IMPLIES (EVAL P (S PRG (II Q PRG)))
                       (EVAL Q (S PRG (II Q PRG)))))
         (INVARIANT Q PRG))
```

II is the Skolem function replacing the existentially quantified variable **I** in the definition of **INVARIANT**. One proves that **P** is stronger than **Q** at some (particular) arbitrary point in the computation.

The next theorem notes that **(LEADS-TO P ’(FALSE) PRG)** is another way of stating that the negation of **P** is an invariant. This is formalized, in a more general way, in the following theorem:

Theorem: Leads-To-False-Invariant

```
(IMPLIES (AND (LEADS-TO P ’(FALSE) PRG)
              (IMPLIES (EVAL (LIST ’NOT P)
                             (S PRG (II INV PRG)))
                       (EVAL INV
                             (S PRG (II INV PRG)))))
         (INVARIANT INV PRG))
```

The next theorems are concerned with proving and manipulating the predicate **EVENTUALLY-INVARIANT**. The first theorem states that if a predicate is stable and is reached, it is eventually invariant:

Theorem: Stable-Occurs-Proves-Eventually-Invariant

```
(IMPLIES (AND (LISTP PRG)
              (UNLESS P ’(FALSE) PRG)
              (LEADS-TO ’(TRUE) P PRG))
         (EVENTUALLY-INVARIANT P PRG))
```

The next theorem states that if a predicate is eventually invariant, any consequence of it is eventually invariant also. The theorem is similar to **Invariant-Consequence**. The arbitrary computation point is computed by the composition of the Skolem functions **JES** and **IES**, from the definition of **EVENTUALLY-INVARIANT**:

Theorem: Eventually-Invariant-Weaken

```
(IMPLIES (AND (EVENTUALLY-INVARIANT P PRG)
              (IMPLIES (EVAL P
                             (S PRG
                                (JES (IES PRG P)
                                     PRG R)))
                       (EVAL R
                             (S PRG
                                (JES (IES PRG P)
                                     PRG R)))))
         (EVENTUALLY-INVARIANT R PRG))
```

The next theorem states that if both **P** and **Q** are eventually invariant, then any consequence of their conjunction is also eventually invariant.

Theorem: Eventually-Invariant-Conjunction

```
(IMPLIES
 (AND (EVENTUALLY-INVARIANT P PRG)
      (EVENTUALLY-INVARIANT Q PRG)
      (IMPLIES
       (EVAL (LIST 'AND P Q)
             (S PRG
                (JES (IF (LESSP
                          (IES PRG P)
                          (IES PRG Q))
                         (IES PRG Q)
                       (IES PRG P))
                     PRG R)))
       (EVAL R (S PRG
                  (JES (IF (LESSP
                            (IES PRG P)
                            (IES PRG Q))
                           (IES PRG Q)
                         (IES PRG P))
                       PRG R)))))
 (EVENTUALLY-INVARIANT R PRG))
```

As with **Leads-To-Strengthen-Left** and **Leads-To-Weaken-Right**, it is necessary to appeal to the Skolemization of the definition of **EVENTUALLY-INVARIANT** in order to obtain the arbitrary point in the computation at which the conjunction of **P** and **Q** must be stronger than **R**.[14]

LEADS-TO and **EVENTUALLY-INVARIANT** are basically negations of each other. For example, if **(LEADS-TO P Q PRG)** is not true, then eventually the negation of **Q** is invariant. Why? This **LEADS-TO** states that whenever **P** holds, **Q** holds sometime later (perhaps right away). The negation is that there exists a point at which **P** holds and subsequently **Q** never holds. Equivalently, the negation of **Q** is eventually invariant. This is formalized in the next theorem:

Theorem: Not-Leads-To-Proves-Eventually-Invariant

```
(IMPLIES
 (AND (NOT (LEADS-TO P NOT-R PRG))
      (IMPLIES (NOT (EVAL NOT-R
                          (S PRG
                             (JES
                              (ILEADS P PRG NOT-R)
                              PRG R))))
               (EVAL R
                     (S PRG
                        (JES
                         (ILEADS P PRG NOT-R)
                         PRG R)))))
 (EVENTUALLY-INVARIANT R PRG))
```

Notice that Skolem functions from the definitions of both **LEADS-TO** and **EVENTUALLY-INVARIANT** are composed. The contrapositive, where the first hypothesis is negated and exchanged with the negation of the conclusion, is also a theorem, obviously, and is named **Not-Eventually-Invariant-Proves-Leads-To**. These theorems are used extensively in the proof of the dining philosophers algorithm (chapter 6).

[14] The following simple theorem, used together with **Eventually-Invariant-Weaken**, may be used in place of **Eventually-Invariant-Conjunction**. But the use of a single theorem is often more convenient:

```
(IMPLIES (AND (EVENTUALLY-INVARIANT P PRG)
              (EVENTUALLY-INVARIANT Q PRG))
         (EVENTUALLY-INVARIANT (LIST 'AND P Q) PRG))
```

3.3 Fairness and Deadlock Freedom

The next sections present proof rules for Unity's fairness notion (unconditional fairness), for weak and strong fairness, and for the safety assumption of deadlock freedom. All four of these notions place restrictions on a scheduler (this scheduler is already both unconditionally and weakly fair) and therefore yield computations satisfying stronger properties. Furthermore, the fairness proof rules permit the proof of basic liveness properties directly from the program text, which are then combined together using the proof rules described in the previous section (section 3.2).

3.3.1 Unconditional Fairness

The weakest fairness notion, *unconditional fairness*, requires that program statements be always enabled (statements are total functions).[15]

Consequently, no restrictions are placed on a scheduler, other than that every statement be scheduled infinitely often (in any particular order). It is easy to imagine scenarios where starvation or deadlock occur under unconditional fairness.

The first requirement of unconditional fairness, that all program statements be total functions, is captured by defining the function **TOTAL**:
Definition:

```
(TOTAL PRG)
   ⇔
(FORALL E (IMPLIES (MEMBER E PRG)
                   (FORALL OLD
                           (EXISTS NEW
                                   (N OLD NEW E)))))
```

(TOTAL PRG) is true only if every statement in **PRG** specifies at least one successor state for every previous state. The successor state may be unchanged (the statement may be a skip statement).

The proof rule for deducing the liveness properties of programs executed under unconditional fairness is supported by the following intuition. We wish to prove a simple **LEADS-TO** property: **(LEADS-TO P Q PRG)**. That is, every **P** state is eventually followed by some **Q** state. Suppose that every program statement takes **P** states to states where **P** or **Q** holds. Then we know that **P** persists, at least until **Q** holds. (This is formalized by **(UNLESS P Q PRG)**.) Furthermore, if there exists some statement that transforms all **P** states to **Q** states, then, by fairness, we know that statement will eventually be executed. If **Q** has not yet held, since **P** persists, **Q** will hold subsequent to the first execution of that statement. The notion of some statement transforming all **P** states to **Q** states is captured by the function **ENSURES** (borrowed from Unity):
Definition:

```
(ENSURES P Q PRG)
   ⇔
(EXISTS E (AND
           (MEMBER E PRG)
           (FORALL (OLD NEW)
                   (IMPLIES
                    (AND (N OLD NEW E)
                         (EVAL (LIST 'AND P
                                     (LIST 'NOT Q))
                               OLD))
                    (EVAL Q NEW)))))
```

Therefore, if a program is **TOTAL**, and **P** persists until **Q** and some statement transforms all **P** states to **Q**

[15] Actually, this isn't a fairness property at all, since it does not satisfy the criterion of feasibility in [5], since demonstrating that statements are total functions is unrelated to building a scheduler. In Unity, where statements are total by definition, weak fairness and unconditional fairness are the same.

states, unconditional fairness implies that **(LEADS-TO P Q PRG)** holds as well. This argument is formalized in the following theorem, which is the proof rule for unconditional fairness:

Theorem: Unconditional-Fairness

```
(IMPLIES (AND (UNLESS P Q PRG)
              (ENSURES P Q PRG)
              (TOTAL PRG))
         (LEADS-TO P Q PRG))
```

Notice, that this theorem does not require any assumptions about the computation (other than what is implied by the characterization of **S**). This is because any arbitrary ordering of statements (provided every statement is scheduled infinitely often) is sufficient for unconditional fairness.

Notice also, that **ENSURES** is inappropriate if a program is not **TOTAL**, because if some statement is disabled by the precondition **P**, **(ENSURES P Q PRG)** is vacuously true.

3.3.2 Weak Fairness

Weak fairness is an extension of unconditional fairness for programs that are not **TOTAL**. Weak fairness excludes from consideration computations in which a statement is enabled continuously but is not scheduled effectively. That is, in order to guarantee that a statement will effect an effective transition under weak fairness, one must ensure that once it is enabled, it remains enabled (at least) until it is scheduled. Notice, however, that this is a simple property of the computation **S**: since, by fairness, every statement is scheduled again, if some statement is continuously enabled from some point in the computation, it will execute effectively the next time it is scheduled. Hence, the proof rule describing weak fairness is a theorem that requires some knowledge about statements' enabling conditions.

To specify this notion, we first introduce a predicate that identifies a statement's enabling condition.

Definition:

```
(ENABLING-CONDITION C E PRG)
   ⇔
(AND (MEMBER E PRG)
     (FORALL (OLD NEW)
             (IMPLIES (N OLD NEW E)
                      (EVAL C OLD)))
     (FORALL OLD (IMPLIES (EVAL C OLD)
                          (EXISTS NEW (N OLD NEW E)))))
```

(ENABLING-CONDITION C E PRG) states that **C** is the enabling condition for statement **E** in program **PRG**. That is, for all possible transitions, **C** holds on the previous state, and if **C** holds on the previous state, some successor state exists.

We now define a predicate similar to **ENSURES** that considers enabling conditions:

Definition:

```
(E-ENSURES P Q C PRG)
   ⇔
(EXISTS E
        (AND (MEMBER E PRG)
             (ENABLING-CONDITION C E PRG)
             (FORALL (OLD NEW)
                     (IMPLIES
                      (AND (N OLD NEW E)
                           (EVAL (LIST 'AND P
                                       (LIST 'NOT Q))
                                 OLD))
                      (EVAL Q NEW))))))
```

(E-ENSURES P Q C PRG) says that some statement takes some **P** states to **Q** states (and is disabled for all the rest) and has enabling condition **C**.

The intuition behind the proof rule for weak fairness is as follows: We wish to prove **(LEADS-TO P Q**

PRG). Assume that **P** persists at least until **Q** holds (**(UNLESS P Q PRG)**). Also assume that some key statement transforms **P** states to **Q** states and has enabling condition **C**. Then, if **P** implies **C** during the interval starting when **P** first holds and ending when the key statement is scheduled (or when **Q** holds), we may deduce that **Q** ultimately occurs, for if **Q** has not yet held, then the key statement will be scheduled effectively and **Q** will hold subsequently. This argument is formalized in the following theorem:

Theorem: Weak-Fairness

```
(IMPLIES
 (AND (UNLESS P Q PRG)
      (E-ENSURES P Q C PRG)
      (IMPLIES
       (EVAL (LIST 'AND P (LIST 'NOT Q))
             (S PRG (WITNESS P Q C PRG)))
       (EVAL C (S PRG (WITNESS P Q C PRG)))))
 (LEADS-TO P Q PRG))
```

At first glance, this theorem seems not to follow the reasoning outlined above, for it appears to only check whether the key statement's enabling condition **C** holds at the single point **(WITNESS P Q C PRG)** and not whether **C** holds continuously over the appropriate interval. However, **WITNESS** is defined to inspect that interval and return the first point where the key statement is disabled. (If the key statement is enabled continuously, then **WITNESS** returns the first point when either **Q** holds or the key statement is scheduled.) In this theorem, the hypothesis requires that the key statement be enabled (or **Q** holds) even at that point. If that is the case, then we may deduce that the key statement is enabled continuously until the state before **Q** holds.

The advantage of defining **WITNESS** in this way is that it transforms an inductive argument (inspecting an arbitrary interval) to analysis at a single arbitrary point; this simplifies reasoning. The definition of **WITNESS** is:

Definition:

```
(WITNESS P Q C PRG)
   =
(WFW (ILEADS P PRG Q)
     (NEXT PRG (EEE C P PRG Q)
           (ILEADS P PRG Q))
     P Q C PRG)
```

EEE is the Skolem function representing the existentially quantified **E** in function **E-ENSURES**. **WFW** is defined as:

Definition:

```
(WFW I J P Q C PRG)
   =
(IF (LESSP I J)
    (IF (EVAL Q (S PRG I))
        I
      (IF (EVAL P (S PRG I))
          (IF (EVAL C (S PRG I))
              (WFW (ADD1 I) J P Q C PRG)
            I)
        I))
  (FIX I))
```

Notice, that weak fairness is more general than unconditional fairness since programs need not be **TOTAL**, yet does not require special scheduling not already guaranteed by the computation **S**. However, both starvation and deadlock are still possible under weak fairness.

3.3.3 Strong Fairness

Strong fairness precludes computations where a statement that is enabled infinitely often is never scheduled effectively. Equivalently, strong fairness guarantees that if a statement is enabled infinitely often, it is scheduled effectively infinitely often.

Strong fairness might be used, in special circumstances, to guarantee freedom from starvation. Starvation can occur when a process needs a resource, which is available infinitely often (but not necessarily continuously), yet only requests the resource when it is unavailable.

The proof rule for deducing strong fairness properties is supported by the following intuition. Suppose that we wish to prove **`(LEADS-TO P Q PRG)`** and we know that **`P`** persists at least until **`Q`** holds (**`(UNLESS P Q PRG)`**). Suppose further that there exists some key statement with enabling condition **`C`** that transforms states where both **`P`** and **`C`** hold to **`Q`** states. Under strong fairness, to guarantee that the key statement is scheduled effectively, one must demonstrate that **`P`** and **`C`** occur together often enough (e.g., could occur infinitely often). Therefore, to prove **`(LEADS-TO P Q PRG)`** it is sufficient to prove **`(LEADS-TO P (LIST 'OR Q C))`**, since, by hypothesis, **`P`** persists until **`Q`**, and if **`C`** holds before **`Q`**, then the key statement could be scheduled effectively. If it is not scheduled at that point, then we repeat the argument. By strong fairness, eventually, the key statement will be scheduled effectively.

The proof rule formalizing this argument is:
Constraint: Strong-Fairness

```
(IMPLIES (AND (UNLESS P Q PRG)
              (E-ENSURES P Q C PRG)
              (LEADS-TO P (LIST 'OR Q C) PRG)
              (STRONGLY-FAIR PRG))
         (LEADS-TO P Q PRG))
```

The term **`(STRONGLY-FAIR PRG)`** introduces a new (undefined) function symbol that, essentially, tags uses of this proof rule. Since **`(STRONGLY-FAIR PRG)`** cannot be proved, it must be a hypothesis to any **`LEADS-TO`** property deduced using this proof rule. Furthermore, since reasoning directly about the operational semantics **`S`** yields nothing more than weak fairness, it is impossible to deduce stronger results about **`S`** without appealing to this proof rule. This proof rule is consistent with the rest of this theory because there exists a model for the function **`STRONGLY-FAIR`** satisfying this constraint: any unary function which is always false. Completeness and appropriateness is justified by the correctness of the supporting literature [61, 58]. Notice, that this proof rule is not a proved theorem, but a sound constraint. An interpreter for strongly fair computations cannot be formalized in first order logic.

It may appear that this proof rule requires circular reasoning: it proves one **`LEADS-TO`** property by appealing to another. A clever answer is found in [61]: We can disregard the key statement when proving the **`LEADS-TO`** property in the hypothesis, since if it is ultimately scheduled effectively, we can ignore the **`LEADS-TO`** property in the hypothesis (since **`Q`** is then reached), and if it not scheduled effectively, we can ignore it when deducing that hypothesis (since it is equivalent to a skip statement). Other researchers emphasize this point by permitting the **`LEADS-TO`** property in the hypothesis to be proved with respect to a smaller program: the original less the key statement [58]. In this proof system, the circularity may be broken in the following way: at some point in a proof, one of the **`LEADS-TO`** properties is proved by appealing to the weak fairness proof rule.

3.3.4 Deadlock Freedom

Deadlock freedom guarantees the absence of deadlock in the computation. Deadlock occurs when a statement which ought to be able to execute remains disabled. More precisely, a deadlocked condition is a stable condition that disables some program statement.

The proof rule formalizing this notion is:

Constraint: Deadlock-Freedom

```
    (IMPLIES (AND (UNLESS INV '(FALSE) PRG)
                  (ENABLING-CONDITION C E PRG)
                  (IMPLIES (EVAL INV
                                 (S PRG
                                    (NEXT PRG E
                                          (ILEADS INV PRG
                                                  '(FALSE)))))
                           (NOT (EVAL C
                                      (S PRG
                                         (NEXT PRG E
                                               (ILEADS
                                                INV PRG
                                                '(FALSE))))))
                  (DEADLOCK-FREE PRG))
             (LEADS-TO INV '(FALSE) PRG))
```

This proof rules states that if **INV** is stable and **C** is statement **E**'s enabling condition, if **INV** is a stronger predicate than the negation of **C** then **INV** is false of every state in the computation. That is, the negation of **INV** is an invariant of the computation (section 3.2.2). Stating that **INV** is stronger than the negation of **C** with respect to computation states is more powerful than stating it with respect to all states (since computation states is a smaller set). As with strong fairness, the new function symbol **DEADLOCK-FREE** is an undefined function symbol which serves as a tag for uses of this proof rule. This proof rule is consistent with the rest of this theory because any unary function whose value is always false serves as a model for **DEADLOCK-FREE** in this constraint.

One might argue that deadlock freedom is too strong a proof rule, for unlike strong fairness, it is not, in general, implementable [5]. But strong fairness is a useful abstraction: for example, a high level algorithm may be proved correct assuming deadlock freedom, and then implemented following locking rules that do prevent deadlock.[16]

In the next chapters, the proof rules for weak fairness, strong fairness, and deadlock freedom are illustrated by the proof of four programs.

3.4 Proof Rules about UNLESS, ENSURES, and TOTAL

There are many theorems for manipulating the predicates **UNLESS**, **ENSURES**, and **TOTAL**. These predicates consider program statements and all states, not just reachable ones.

3.4.1 Program Composition

This section presents three theorems about program composition. Since programs are simply a list of statements, the composition of two programs is the concatenation of two lists (using the function **APPEND**). The predicates **TOTAL**, **UNLESS**, and **ENSURES** all compose. Computation properties, like **LEADS-TO** and **INVARIANT**, do not compose in a straightforward manner.

Both **TOTAL** and **UNLESS** are properties satisfied by every statement in the program. The first theorem states that **TOTAL** composes.

Theorem: Total-Union

```
(IFF (TOTAL (APPEND PRG-1 PRG-2))
     (AND (TOTAL PRG-1)
          (TOTAL PRG-2)))
```

A similar fact holds for **UNLESS**.

[16] Thanks to Jeannette Wing for suggesting this strategy.

Theorem: Unless-Union

```
(IFF (UNLESS P Q (APPEND PRG-1 PRG-2))
     (AND (UNLESS P Q PRG-1)
          (UNLESS P Q PRG-2)))
```

ENSURES composes as well; however the key statement need be present in only one of the component programs:[17]

Theorem: Ensures-Union

```
(IFF (ENSURES P Q (APPEND PRG-1 PRG-2))
     (OR (ENSURES P Q PRG-1)
         (ENSURES P Q PRG-2)))
```

3.4.2 Strengthening and Weakening UNLESS, and ENSURES

Sometimes is it useful to be able to strengthen or weaken the arguments to **UNLESS** and **ENSURES**. This section presents several useful theorems. First, we define a predicate that relates predicates:

Definition:

```
(STRONGER-P P Q)
   ⇔
(FORALL STATE (IMPLIES (EVAL P STATE)
                       (EVAL Q STATE)))
```

The term **(STRONGER-P P Q)** is true if and only if **P** is a stronger predicate than **Q**. Using this predicate, we may state and prove several theorems about **UNLESS** and **ENSURES**.

Theorem: Unless-Weaken-Right

```
(IMPLIES (AND (UNLESS P Q PRG)
              (STRONGER-P Q R))
         (UNLESS P R PRG))
```

Theorem: Ensures-Weaken-Right

```
(IMPLIES (AND (ENSURES P Q PRG)
              (STRONGER-P Q R))
         (ENSURES P R PRG))
```

Theorem: Ensures-Strengthen-Left

```
(IMPLIES (AND (ENSURES Q R PRG)
              (STRONGER-P P Q))
         (ENSURES P R PRG))
```

Several conjunction and disjunction theorems were proved as well.

3.5 Comparison With Unity Predicates

The major differences between the **UNLESS** and **ENSURES** defined here, and Unity's definitions, are that these predicates consider all states and not just reachable ones, and that statements here may be non-deterministic and may be disabled. Aside from this, however, the definition of **ENSURES** here differs slightly from Unity's. In Unity, using Hoare triples [41], the definition of **ENSURES** is:

```
P ENSURES Q
   ≡
(P UNLESS Q ∧ ⟨∃S : S IN PRG :: {P∧¬Q} S {Q}⟩)
```

Our **ENSURES** does not imply **UNLESS**; to achieve the same effect, one states that both hold (e.g., **(AND (UNLESS P Q PRG) (ENSURES P Q PRG))**).[18]

---

[17]Unity's **ENSURES**, which implies **UNLESS**, requires that **UNLESS** hold in each component program.

[18]We find the statement of several theorems more convenient when **ENSURES** and **UNLESS** are separate.

Unity does not provide a definition for **`LEADS-TO`**. Rather, it presents three proof rules and defines **`LEADS-TO`** to be the strongest predicate satisfying those rules. In this way, Unity avoids formalizing an operational semantics that may be used to define **`LEADS-TO`**. Furthermore, Unity's method for defining **`LEADS-TO`** allows one to use induction on the length of the proof (structural induction) to prove theorems about **`LEADS-TO`**. The soundness and completeness of Unity's **`LEADS-TO`** are discussed in [50].

However, if an operational semantics is formalized, and **`LEADS-TO`** is correctly defined using those functions, then the definition is sound. Furthermore, it is easy to infer the meaning of this **`LEADS-TO`**, and to see that it is the predicate that Unity's axioms mean to define. Also, **`LEADS-TO`** may be meaningfully negated; this negation implies eventual stability. All theorems (except for several new compositional theorems [69, 82]) about Unity's **`LEADS-TO`** are theorems of the **`LEADS-TO`** presented here and are proved by induction on the second argument to **`S`** (the index in the computation). Such theorems allow the proof of progress properties without appealing to the operational semantics and are equivalent to Unity's proof rules.

3.6 Conclusion

This chapter is based on [29, 30, 31].

All the theorems in this and the previous chapter were verified on the extended Boyer-Moore prover with many extra lemmas, and together comprise the proof system called Mechanized Unity. Taking the underlying naturals library for granted [7], this proof required 34 definitions and 156 lemmas filling 2052 pretty printed lines of text.

The entire event list for constructing this proof system is presented in Appendix B.

Chapter 4
MUTUAL EXCLUSION

This chapter presents a mechanically verified n-processor program satisfying both mutual exclusion and absence of starvation. The purpose of this chapter is to describe, by means of a simple example, how to mechanically verify a concurrent program on the Boyer-Moore prover using the theorems presented earlier.

Mutual exclusion is a resource allocation problem. A solution must ensure that only one process may have the resource at a time. Absence of starvation requires that any process desiring the resource will eventually receive it. The first requirement is an invariance property; the second is a liveness property.

Informally, the program described here defines a ring of identical processes, each of which differs only by its location in the ring. A process can send a message to the next process in the ring and receive a message from the previous process in the ring. Each process has three states: non-critical, wait, and critical.[19] A non-critical process non-deterministically becomes waiting and remains waiting until a token becomes available on its incoming channel. It then absorbs that token and becomes critical, and remains critical for a finite number of steps after which it releases a token upon its outgoing channel and becomes non-critical.[20] A non-critical process that does not become waiting will pass a token, if one is available, from its incoming to its outgoing channel.

The following sections are written using a bottom-up approach, where most functions are defined before they are used. Section 4.1 formalizes the transitions each process is permitted. Section 4.2 defines the set of statements that make up the ring. Section 4.3 specifies the correctness theorems for a solution to this mutual exclusion problem. Finally, sections 4.4 and 4.5 present the proofs of the correctness theorems. These proofs have been validated on the extended Boyer-Moore prover.

4.1 The Processes

Each process is a single statement in the program. Before defining the process, we must define several other functions.

The state of the system is a list of pairs (an *association list* or *alist*) which may be accessed by the **`ASSOC`** function. **`(ASSOC KEY ALIST)`** returns the first pair in **`ALIST`** such that the **`CAR`** of that pair is **`KEY`**. If no such pair exists, then **`ASSOC`** returns **`F`**.

We now define several functions which test which state a process is in, and whether a channel has a token on it. The function **`STATUS`** finds the state of a process. The key for each process's status is the pair **`(CONS 'ME INDEX)`** where **`INDEX`** is the process's index in the ring.

---

[19]Actually, the critical state is any of a set of states. See section 4.1 for the definition of **`CRITICAL`**.

[20]This sequence of steps could be hidden, and the token released immediately. However, this complication is a nice demonstration of Unity's infinite disjunction proof rule.

Definition:

```
(STATUS STATE INDEX)
   =
(CDR (ASSOC (CONS 'ME INDEX) STATE))
```

A process is non-critical if its status is **'NON-CRITICAL**. Similarly, a process is waiting if its status is **'WAIT**.

Definition:

```
(NON-CRITICAL STATE INDEX)
   =
(EQUAL (STATUS STATE INDEX) 'NON-CRITICAL)
```

Definition:

```
(WAIT STATE INDEX)
   =
(EQUAL (STATUS STATE INDEX) 'WAIT)
```

A process is critical if its status is neither **'NON-CRITICAL** nor **'WAIT**.

Definition:

```
(CRITICAL STATE INDEX)
   =
(AND (NOT (NON-CRITICAL STATE INDEX))
     (NOT (WAIT STATE INDEX)))
```

If a process is critical, then its status is a number representing the maximum number of its own transitions for which the process may remain critical. This number is identified using the function **TICKS**:

Definition:

```
(TICKS STATE INDEX)
   =
(FIX (STATUS STATE INDEX))
```

CHANNEL returns the contents of the **INDEX**'th channel. The key for a channel is **(CONS 'C INDEX)**. The **INDEX**'th process's incoming channel has key **(CONS 'C INDEX)** and its outgoing channel has key **(CONS 'C (ADD1-MOD N INDEX))** where **ADD1-MOD** adds one modulo **N**, where **N** is the number of processes in the ring.

Definition:

```
(CHANNEL STATE INDEX)
   =
(CDR (ASSOC (CONS 'C INDEX) STATE))
```

Finally, **TOKEN** tests whether a channel has a token on it, by checking whether the channel is non-empty. A token is simply a message on the channel; the message itself is unimportant.

Definition:

```
(TOKEN STATE INDEX)
   =
(LISTP (CHANNEL STATE INDEX))
```

The function **ME** defines a generic process in the ring. It takes four arguments: **INDEX** instantiates the function to be a specific process in the ring of size **SIZE**; **OLD** and **NEW** are old and new states. **ME** tests whether **NEW** is a possible successor state to **OLD**.

Definition:

```
(ME OLD NEW INDEX SIZE)
   =
(IF (NON-CRITICAL OLD INDEX)
    (IF (NON-CRITICAL NEW INDEX)
        (IF (TOKEN OLD INDEX)
            (IF (EQUAL (ADD1-MOD SIZE INDEX) INDEX)
                (AND (EQUAL (LENGTH (CHANNEL NEW INDEX))
                            (LENGTH (CHANNEL OLD INDEX)))
                     (CHANGED OLD NEW
                              (LIST (CONS 'C INDEX))))
              (AND (EQUAL (CHANNEL NEW INDEX)
                          (CDR (CHANNEL OLD INDEX)))
                   (EQUAL (LENGTH (CHANNEL NEW
                                           (ADD1-MOD SIZE
                                                     INDEX)))
                          (ADD1 (LENGTH
                                 (CHANNEL OLD
                                          (ADD1-MOD
                                           SIZE INDEX)))))
                   (CHANGED OLD NEW
                            (LIST (CONS 'C INDEX)
                                  (CONS 'C (ADD1-MOD
                                            SIZE INDEX))))))
          (CHANGED OLD NEW NIL))
      (AND (WAIT NEW INDEX)
           (CHANGED OLD NEW (LIST (CONS 'ME INDEX)))))
  (IF (WAIT OLD INDEX)
      (IF (TOKEN OLD INDEX)
          (AND (EQUAL (CHANNEL NEW INDEX)
                      (CDR (CHANNEL OLD INDEX)))
               (CRITICAL NEW INDEX)
               (CHANGED OLD NEW (LIST (CONS 'ME INDEX)
                                      (CONS 'C INDEX))))
        (CHANGED OLD NEW NIL))
    (OR (AND (LESSP (TICKS NEW INDEX) (TICKS OLD INDEX))
             (CRITICAL NEW INDEX)
             (CHANGED OLD NEW (LIST (CONS 'ME INDEX))))
        (AND (NON-CRITICAL NEW INDEX)
             (EQUAL (LENGTH (CHANNEL NEW (ADD1-MOD SIZE
                                                   INDEX)))
                    (ADD1 (LENGTH (CHANNEL OLD
                                           (ADD1-MOD
                                            SIZE INDEX)))))
             (CHANGED OLD NEW
                      (LIST (CONS 'C (ADD1-MOD SIZE INDEX))
                            (CONS 'ME INDEX))))))))
```

ME defines precisely what is a legal transition for each process. It must check whether there is only one process in the ring; in that case the incoming channel and the outgoing channel are really the same. Furthermore, it must specify that although each process changes certain parts of the state, each process preserves other parts. This is akin to not disturbing the local variables of other processes. The term **(CHANGED OLD NEW EXCPT)** tests whether **OLD** and **NEW** agree on every key except for keys in the list **EXCPT**. (Only keys in **EXCPT** may change.) Specifically, **(CHANGED OLD NEW NIL)** means that **ASSOC** cannot infer a difference between the two lists. (For the definition of **CHANGED**, see section 4.4.)

There are three places in **ME** where non-determinacy in used. The first is when the non-critical process may become waiting or remain non-critical. This choice is done without considering whether there is a token on the incoming channel. The second is when passing or releasing a token: **ME** specifies that the length of the outgoing channel is increased by one, not that any particular message is sent. The third is when the critical process decides whether to remain critical or become non-critical. **ME** guarantees only that if the process remains critical, the counter on that process decreases, although it does not specify by how much. Therefore, if the counter is zero (tested by **ZEROP**), the process must become non-critical and release a token. If the counter is non-zero, the process can either decrease the counter or become non-critical and release a token.

4.2 The Program

A program is a list of statements. Each statement is a list **(CONS FUNC ARGS)** where **FUNC** is a function symbol. Recall (section 2.1) that the definition of the function **N**, which interprets statements, is:
Definition:

```
(N OLD NEW E)
   =
(APPLY$ (CAR E) (APPEND (LIST OLD NEW) (CDR E)))
```

In the case of the mutual exclusion program, each statement is of the form **(LIST 'ME INDEX SIZE)** where **SIZE** is the number of processes (i.e., number of statements) and **INDEX** is some number less than **SIZE**. Therefore, **(N OLD NEW (LIST 'ME INDEX SIZE))** is **(ME OLD NEW INDEX SIZE)** which is a call to the function that determines legal transitions for generic processes in the ring, instantiated to the index **INDEX** with size **SIZE**.

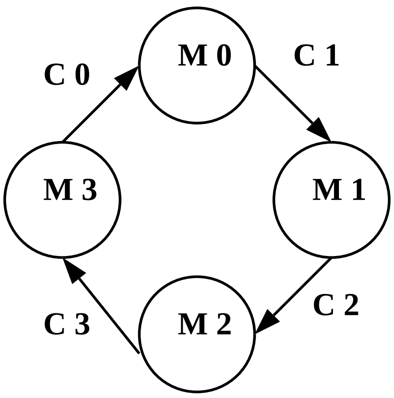
The entire program is a list of such statements where **INDEX** ranges from **0** to **(SUB1 SIZE)**. This is accomplished using the function **PROGRAM**.
Definition:

```
(PROGRAM INDEX SIZE)
   =
(IF (ZEROP INDEX)
    NIL
  (CONS (LIST 'ME (SUB1 INDEX) SIZE)
        (PROGRAM (SUB1 INDEX) SIZE)))
```

(PROGRAM SIZE SIZE) is the list of statements **(LIST (LIST 'ME (SUB1 SIZE) SIZE)** ... **(LIST 'ME 0 SIZE))**. This is abbreviated with the function **ME-PRG**, which defines the ring of processes.
Definition:

```
(ME-PRG SIZE)
   =
(PROGRAM SIZE SIZE)
```

For example, a ring of four processes is the list of statements returned by **(ME-PRG 4)** which equals **'((ME 3 4) (ME 2 4) (ME 1 4) (ME 0 4))**. This may be represented by the following picture, where **ME I** is the **I**'th process in the ring, and **C I** is the channel from process **I-1 MOD 4** to process **I**:



4.3 The Correctness Specification

Before proving that the program **(ME-PRG SIZE)** correctly implements both mutual exclusion and absence of starvation on a ring of size **SIZE**, we must state the program's invariance and liveness properties. The invariance property must guarantee that no two processes are critical simultaneously. This is implied by the property that the sum of the number of tokens in the system and the number of critical processes is always one. The term **(WEIGHT STATE SIZE)** recursively adds up the number of tokens and the number of critical processes in a ring of size **SIZE** under state **STATE**.

Definition:

```
(WEIGHT STATE SIZE)
   =
(IF (ZEROP SIZE)
    0
  (PLUS (IF (CRITICAL STATE (SUB1 SIZE))
            1 0)
        (LENGTH (CHANNEL STATE (SUB1 SIZE)))
        (WEIGHT STATE (SUB1 SIZE))))
```

The mutual exclusion property is simply that the weight is always one. This is defined in the following function.

Definition:

```
(MUTUAL-EXCLUSIONP STATE SIZE)
   =
(EQUAL (WEIGHT STATE SIZE) 1)
```

The invariance of mutual exclusion is then the following theorem:

Theorem: Mutual-Exclusionp-Is-Invariant

```
(IMPLIES (AND (INITIAL-CONDITION
                ‘(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                (ME-PRG SIZE))
               (NOT (ZEROP SIZE)))
         (INVARIANT ‘(MUTUAL-EXCLUSIONP STATE
                                        (QUOTE ,SIZE))
                    (ME-PRG SIZE)))
```

This theorem is in the form of invariance statements, and assumes that **MUTUAL-EXCLUSIONP** holds on the initial state. Since **SIZE** is a variable (section 1.2.2), mutual exclusion is an invariant of all ring sizes.

The liveness property specifies that any waiting process eventually becomes critical. This is formalized in the following theorem:

Theorem: Wait-Leads-To-Critical

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX SIZE)
              (INITIAL-CONDITION
                ‘(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                (ME-PRG SIZE)))
         (LEADS-TO ‘(WAIT STATE (QUOTE ,INDEX))
                   ‘(CRITICAL STATE (QUOTE ,INDEX))
                   (ME-PRG SIZE)))
```

The conclusion is a **LEADS-TO** statement which specifies that for any process which is waiting during the computation there exists a later state when that same process is critical. The hypotheses state that the initial conditions satisfy mutual exclusion and that the process's index must be a number less than **SIZE**. (This also implies that the program is non-empty.) Again, this theorem holds for all non-empty ring sizes.

The proof of these theorems is discussed in the next sections.

4.4 The Proof of Mutual Exclusion

In this program, all statements are identical, except for an index that ranges between zero and **`SIZE-1`**. Therefore, when proving certain properties, it is simpler to prove that the property holds for an arbitrary process in the ring, rather than for each process in the ring. Recall that the definitions of **`UNLESS`** (section 3.1.2) and **`TOTAL`** (section 3.3.1) contain the term **`(MEMBER E PRG)`**, where **`E`** is universally quantified. For this program, a useful theorem is:
Theorem: Member-Me-Prg

```
(EQUAL (MEMBER STATEMENT (ME-PRG SIZE))
       (IF (ZEROP SIZE)
           F
           (AND (EQUAL (CAR STATEMENT) 'ME)
                (NUMBERP (CADR STATEMENT))
                (LESSP (CADR STATEMENT) SIZE)
                (EQUAL (CADDR STATEMENT) SIZE)
                (EQUAL (CDDDR STATEMENT) NIL))))
```

This theorem rewrites terms of the form **`(MEMBER STATEMENT (ME-PRG SIZE))`**[21] to a conjunction that states that the name of the function is **`ME`**, the second element in the statement is a number less than **`SIZE`** and the last element in the statement is **`SIZE`**. Hence, this theorem rewrites formulas whose proofs are inductive (due to **`MEMBER`**) to formulas whose proofs proceed by case analysis on a single element of the program.

Another useful theorem is about the function **`CHANGED`**. The definition of **`CHANGED`** is:
Definition:

```
(CHANGED OLD NEW EXCPT)
   =
(UC NEW OLD (STRIP-CARS (APPEND OLD NEW)) EXCPT)
```

`STRIP-CARS` collects the **`CAR`**'s of every element in a list. The function **`UC`** is defined as:
Definition:

```
(UC OLD NEW KEYS EXCPT)
   =
(IF (LISTP KEYS)
    (IF (MEMBER (CAR KEYS) EXCPT)
        (UC OLD NEW (CDR KEYS) EXCPT)
      (IF (EQUAL (ASSOC (CAR KEYS) OLD)
                 (ASSOC (CAR KEYS) NEW))
          (UC OLD NEW (CDR KEYS) EXCPT)
        F))
  T)
```

`(CHANGED OLD NEW KEY)` checks whether every key not in **`EXCPT`** has the same **`ASSOC`** value in both **`OLD`** and **`NEW`**. The useful theorems about **`CHANGED`** are best stated with respect to **`UC`**. First, **`UC`** is commutative on its first two arguments:
Theorem: Uc-Commutative

```
(EQUAL (UC OLD NEW KEYS EXCPT)
       (UC NEW OLD KEYS EXCPT))
```

Second, the order of elements in **`KEY`** is unimportant:
Theorem: Uc-Commutative-2

```
(EQUAL (UC OLD NEW (APPEND A B) EXCPT)
       (UC OLD NEW (APPEND B A) EXCPT))
```

Finally, the important property of **`UC`** equates **`ASSOC`**'s.

---

[21]Prover detail: both the function **`ME-PRG`** and the executable **`*1*ME-PRG`** function are disabled, to ensure that such terms are rewritten away.

Theorem: About-Uc

```
(IMPLIES (AND (UC A B (APPEND (STRIP-CARS A)
                              (STRIP-CARS B))
                      EXCPT)
              (NOT (MEMBER KEY EXCPT)))
         (EQUAL (ASSOC KEY A)
                (ASSOC KEY B)))
```

This theorem is a rewrite rule which normalizes **ASSOC**'s in a formula if there is a **UC** in the hypotheses.[22] Typically, such formulas will state that values associated with certain **KEY**'s are changed and those keys will comprise **EXCPT**. The remaining values are unchanged. This theorem rewrites the **ASSOC**'s of the unchanged keys to use the second argument of **UC** instead of the first. Remember that the previous two commutative theorems will have already normalized the **UC** term. This theorem is critical for overcoming some of the complexity introduced by defining statements as relations instead of as functions. Part of the difficulty still remains, because the theorem prover does not simplify the hypothesis **(NOT (MEMBER KEY EXCPT))** while backchaining[23] as well as it would a first class term. (Another useful theorem is **Uc-Of-Update-Assoc**. See Appendix B, page 122).

Now, we must prove that the program **(ME-PRG SIZE)** is total. This fact allows us to use unconditional fairness in the liveness proof. Totality is proved in the following theorem:

Theorem: Total-Prg

```
(TOTAL (ME-PRG SIZE))
```

The key lemma used to prove this theorem is:

Theorem: Total-Me

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX SIZE))
         (ME OLD (ME-FUNCTION INDEX SIZE OLD)
             INDEX SIZE))
```

Recall that the definition of **TOTAL** (section 3.3.1) requires that there be a successor state for every previous state. The function **ME-FUNCTION** is a witness for these successor states (i.e., it computes a valid **NEW** state). The function is defined as follows:

[22] One might ask why **(APPEND (STRIP-CARS A) (STRIP-CARS B))** was used here, while the shorter but equivalent form **(STRIP-CARS (APPEND OLD NEW))** was used in the definition of **CHANGED**. In general, it is preferable to pre-simplify, if possible, the hypotheses of a rewrite rule. The definition will expand and simplify to the longer form.

[23] Backchaining is used to satisfy the hypotheses of a rewrite rule. Before the conclusion of a rewrite rule may be used, its hypotheses must be proved.

Definition:

```
(ME-FUNCTION INDEX SIZE STATE)
   =
(IF (NON-CRITICAL STATE INDEX)
    (IF (TOKEN STATE INDEX)
        (UPDATE-ASSOC
         (CONS 'C (ADD1-MOD SIZE INDEX))
         (CONS 'TOKEN
               (CHANNEL
                (UPDATE-ASSOC
                 (CONS 'C INDEX)
                 (CDR (CHANNEL STATE INDEX))
                 STATE)
                (ADD1-MOD SIZE INDEX)))
         (UPDATE-ASSOC
          (CONS 'C INDEX)
          (CDR (CHANNEL STATE INDEX))
          STATE))
        STATE)
    (IF (WAIT STATE INDEX)
        (IF (TOKEN STATE INDEX)
            (UPDATE-ASSOC
             (CONS 'C INDEX)
             (CDR (CHANNEL STATE INDEX))
             (UPDATE-ASSOC (CONS 'ME INDEX)
                           0
                           STATE))
          STATE)
      (IF (ZEROP (TICKS STATE INDEX))
          (UPDATE-ASSOC
           (CONS 'C (ADD1-MOD SIZE INDEX))
           (CONS 'TOKEN (CHANNEL
                         STATE
                         (ADD1-MOD SIZE INDEX)))
           (UPDATE-ASSOC
            (CONS 'ME INDEX)
            'NON-CRITICAL
            STATE))
        (UPDATE-ASSOC
         (CONS 'ME INDEX)
         (SUB1 (TICKS STATE INDEX))
         STATE))))
```

The term **`(UPDATE-ASSOC KEY VALUE ALIST)`** returns a new association list where key **`KEY`** is paired with **`VALUE`**. **`UPDATE-ASSOC`** is best characterized by the following theorem:

Theorem: Simplify-Assoc

```
(EQUAL (ASSOC KEY-1 (UPDATE-ASSOC KEY-2 VALUE ALIST))
       (IF (EQUAL KEY-1 KEY-2)
           (CONS KEY-1 VALUE)
           (ASSOC KEY-1 ALIST)))
```

The necessity of defining executable versions of statements (in this approach) is a direct and tedious consequence of using relations to describe statements when using unconditional fairness, which requires the statements to be total functions. However, relations do permit the easy specification of non-determinacy and disabled transitions.

The proof of mutual exclusion is done in three steps. First, one proves that for the execution of any process the sum of the weights of its left channel, status, and right channel is preserved (we call these three components a triple). Then one proves that the weight of every other part of the state is unchanged. Finally, one notes that the weight of the entire state is the sum of the weights of the triple and the rest. (This proof scheme is not valid for rings of size one, but that exception is simple case analysis.) Here, we present the key events showing that the weight of the triple is constant.

Definition:

```
(WEIGHT-OF-TRIPLE STATE INDEX SIZE)
   =
(PLUS (IF (CRITICAL STATE INDEX)
          1 0)
      (LENGTH (CHANNEL STATE INDEX))
      (LENGTH (CHANNEL STATE (ADD1-MOD SIZE INDEX))))
```

Theorem: Weight-Of-Triple-Preserved

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP 1 SIZE)
              (LESSP INDEX SIZE)
              (ME OLD NEW INDEX SIZE))
         (EQUAL (WEIGHT-OF-TRIPLE NEW INDEX SIZE)
                (WEIGHT-OF-TRIPLE OLD INDEX SIZE)))
```

These theorems imply the invariance property:
Theorem: Mutual-Exclusionp-Is-Invariant

```
(IMPLIES (AND (INITIAL-CONDITION
               '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
               (ME-PRG SIZE))
              (NOT (ZEROP SIZE)))
         (INVARIANT '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                    (ME-PRG SIZE)))
```

4.5 The Proof of Absence of Starvation

The proof of liveness requires three **ENSURES** properties which demonstrate how a token moves around the ring. These **ENSURES** properties are also **LEADS-TO** properties. The first **ENSURES** property states that if a process is non-critical and has a token on its incoming channel, then it either passes the token to its outgoing channel, or becomes waiting and leaves the token on its incoming channel.
Theorem: Non-Critical-Left-Ensures-Wait-Left-Or-Right

```
(IMPLIES
 (AND (NUMBERP INDEX)
      (LESSP INDEX SIZE))
 (ENSURES '(AND (NON-CRITICAL STATE (QUOTE ,INDEX))
                (TOKEN STATE (QUOTE ,INDEX)))
          '(OR (AND (WAIT STATE (QUOTE ,INDEX))
                    (TOKEN STATE (QUOTE ,INDEX)))
               (TOKEN STATE (QUOTE ,(ADD1-MOD SIZE
                                              INDEX))))
           (ME-PRG SIZE)))
```

In order to deduce that this is also a **LEADS-TO** property, one must also prove that this is an **UNLESS** property as well (that is, other statements in the program do not disturb the precondition):
Theorem: Non-Critical-Left-Unless-Wait-Left-Or-Right

```
(IMPLIES
 (AND (NUMBERP INDEX)
      (LESSP INDEX SIZE))
 (UNLESS '(AND (NON-CRITICAL STATE (QUOTE ,INDEX))
               (TOKEN STATE (QUOTE ,INDEX)))
         '(OR (AND (WAIT STATE (QUOTE ,INDEX))
                   (TOKEN STATE (QUOTE ,INDEX)))
              (TOKEN STATE (QUOTE ,(ADD1-MOD SIZE
                                             INDEX))))
          (ME-PRG SIZE)))
```

The second **ENSURES** property states that if a process is waiting and has a token on its left channel, then the process becomes critical. There is also a corresponding **UNLESS** property.

Theorem: Wait-And-Left-Channel-Ensures-Critical

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX SIZE))
         (ENSURES '(AND (WAIT STATE (QUOTE ,INDEX))
                        (TOKEN STATE (QUOTE ,INDEX)))
                  '(CRITICAL STATE (QUOTE ,INDEX))
                  (ME-PRG SIZE)))
```

The third **ENSURES** property says that if a process is critical, then it either remains critical and decreases its counter, or it releases a token on its outgoing channel. This **ENSURES** is an **UNLESS** property also.

Theorem: Critical-Ensures-Less-Critical-Or-Right

```
(IMPLIES
 (AND (NUMBERP INDEX)
      (LESSP INDEX SIZE))
         (ENSURES '(AND (CRITICAL STATE (QUOTE ,INDEX))
                        (LESSP (TICKS STATE
                                      (QUOTE ,INDEX))
                               (QUOTE ,(ADD1 TICKS))))
                  '(OR (AND
                        (CRITICAL STATE
                                  (QUOTE ,INDEX))
                        (LESSP (TICKS STATE
                                      (QUOTE ,INDEX))
                               (QUOTE ,TICKS)))
                       (TOKEN STATE
                              (QUOTE ,(ADD1-MOD
                                       SIZE INDEX))))
                      (ME-PRG SIZE)))
```

The corresponding **LEADS-TO** theorems are proved by appeal to the **Unconditional-Fairness** proof rule (section 3.3.1, page 30). This last **ENSURES** theorem is especially interesting because we can use induction to show that every critical process eventually releases a token on its outgoing channel. This is done by induction on the counter **TICKS**, and appealing to the proof rules **Leads-To-Weaken-Right** and **Cancellation-Leads-To**. The theorem is:

Theorem: Critical-Ticks-Leads-To-Right

```
(IMPLIES
 (AND (NUMBERP INDEX)
      (LESSP INDEX SIZE))
 (LEADS-TO '(AND (CRITICAL STATE (QUOTE ,INDEX))
                 (LESSP (TICKS STATE (QUOTE ,INDEX))
                        (QUOTE ,(ADD1 TICKS))))
           '(TOKEN STATE (QUOTE ,(ADD1-MOD SIZE
                                           INDEX)))
           (ME-PRG SIZE)))
```

Notice that the initial condition here is not specified since it is irrelevant. Using the theorem, **Leads-To-Strengthen-Left**, we can simplify this result to a process simply being critical, not critical with some value on its counter. This is possible, because every critical process certainly has some value on its counter.

Theorem: Critical-Leads-To-Right

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX SIZE))
         (LEADS-TO '(CRITICAL STATE (QUOTE ,INDEX))
                   '(TOKEN STATE
                           (QUOTE
                            ,(ADD1-MOD SIZE INDEX)))
                   (ME-PRG SIZE)))
```

The application of **Leads-To-Strengthen-Left** in the proof of this theorem demonstrates how the proper instantiation of **P** in that proof rule lets one use that proof rule to accomplish infinite disjunction, which is another proof rule in Unity (section 3.2.1, page 25). The instantiation of **P** used here is:

```
'(AND
  (CRITICAL STATE (QUOTE ,INDEX))
  (LESSP (TICKS STATE (QUOTE ,INDEX))
         (QUOTE
          ,(ADD1 (TICKS (S (ME-PRG SIZE)
                           (ILEADS '(CRITICAL
                                     STATE
                                     (QUOTE ,INDEX))
                                    (ME-PRG SIZE)
                                    '(TOKEN STATE
                                            (QUOTE
                                             ,(ADD1-MOD
                                               SIZE
                                               INDEX)))))
                        INDEX)))))
```

This predicate calculates the ticks remaining on the critical counter by inspecting the state at the arbitrary point at which **Q** must be stronger than **P** in the theorem **Leads-To-Strengthen-Left**.

The next significant lemma uses induction to prove that the token moves around the ring. The hypothesis of mutual exclusion simplifies the proof because it guarantees that if a channel has a token on it, the neighboring process is not critical.

Theorem: Any-Leads-To-Right

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX SIZE)
              (INITIAL-CONDITION
               '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
               (ME-PRG SIZE)))
         (LEADS-TO '(TRUE)
                   '(TOKEN STATE (QUOTE ,INDEX))
                   (ME-PRG SIZE)))
```

We now wish to use the **PSP** theorem to say that a waiting process remains waiting while the token moves around the ring, or it becomes critical. This requires proving the following **UNLESS** property:

Theorem: Wait-Unless-Critical

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX SIZE)
              (NOT (ZEROP SIZE)))
         (UNLESS '(WAIT STATE (QUOTE ,INDEX))
                 '(CRITICAL STATE (QUOTE ,INDEX))
                 (ME-PRG SIZE)))
```

This lemma and and **Any-leads-to-Right** are preconditions for the **PSP** theorem. The resulting **LEADS-TO** is:

Theorem: Wait-Leads-To-Left-Wait-Or-Critical

```
(IMPLIES
 (AND (NUMBERP INDEX)
      (LESSP INDEX SIZE)
      (INITIAL-CONDITION
       '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
       (ME-PRG SIZE)))
 (LEADS-TO '(AND (TRUE)
                 (WAIT STATE (QUOTE ,INDEX)))
           '(OR (AND (TOKEN STATE (QUOTE ,INDEX))
                     (WAIT STATE (QUOTE ,INDEX)))
                (CRITICAL STATE (QUOTE ,INDEX)))
           (ME-PRG SIZE)))
```

It is simpler to include the unnecessary **(TRUE)** in **(AND (TRUE) ...)** because of the structure of the proof. In any case, this simplifies, using **Cancellation-Leads-To** and the **LEADS-TO** theorem derived from **Wait-And-Left-Channel-Ensures-Critical** to:

Theorem: Wait-Leads-To-Critical

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX SIZE)
              (INITIAL-CONDITION
               '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
               (ME-PRG SIZE)))
         (LEADS-TO '(WAIT STATE (QUOTE ,INDEX))
                   '(CRITICAL STATE (QUOTE ,INDEX))
                   (ME-PRG SIZE)))
```

This is the liveness result we need to prove correctness.

4.6 Conclusion

The completion of the mechanically verified correctness proof of this algorithm marked the first time the proofs of both non-propositional safety and liveness properties were mechanically checked for a non-finite state concurrent algorithm [21]. This chapter is based on [29]. [22] has automatically verified several asynchronous finite state machines implementing mutual exclusion.

All the theorems in this chapter were verified on the extended Boyer-Moore prover, with many extra lemmas. Taking the underlying proof system and the naturals library for granted, this proof required 19 definitions and 65 lemmas filling 1047 pretty printed lines of text. Much of the case analysis, especially the possible transitions of each statement, was done automatically by the prover. The entire event list is presented in Appendix C.

Chapter 5

A MINIMUM TREE VALUE ALGORITHM

This chapter presents a mechanically verified distributed algorithm that computes the minimum node value in a tree [56]. Although the algorithm is relatively simple, the proof was quite difficult. This is the most difficult algorithm verified with this proof system. The proof here is based on a detailed hand proof prepared by Lamport, but is extended, since his proof only proved partial correctness. Furthermore, Lamport's proof did not analyze the inductive data structure, and was therefore incomplete. Although one may argue that proving the invariant adequately demonstrates correctness, the liveness proof was nearly as difficult.

The notable elements of this proof are twofold. First, this algorithm contains multiple instances of multiple statements. This presented several new obstacles when developing the proof. Second, this algorithm is based on a tree data structure and involved inductions over the depth of the tree. The theorems developed here for dealing with tree structures may form a basis for a tree library.

Informally, this algorithm assumes a tree of nodes, each assigned some value. The minimum value is computed in the following way: the root node requests minimum node values from each of its children. The minimum of those values and the root's own value is the minimum value in the tree. Each node responds to a request for a minimum value recursively; if it has children, it initiates the same scheme the root did to compute the minimum value of its subtree. If the node is a leaf, then the minimum value is its own value. Once the node computes the minimum value for its subtree, it returns that value to its parent.

The following sections are written using a bottom-up approach, where most functions are defined before they are used. Section 5.1 formalizes the transitions that each node is permitted. Section 5.2 defines the set of statements that comprises the program. Section 5.3 specifies the correctness theorems for a solution to this problem. Finally, section 5.4 presents the proof of the correctness theorems. The proofs have been validated on the extended Boyer-Moore prover.

5.1 The Transitions

The actions of each node are governed by several statements. Recall that the statement interpreter **`N`** (section 2.1, page 15), considers the **`CAR`** of the statement to be a function name, and uses the remainder of the statement to instantiate that generic function to a particular instance. We now present the names of these generic functions, from which we can build the program statements:

- The **`START`** statement initiates the process; the root node sends a message to each of its children requesting the minimum value of its subtree. We call these requests *find* requests.
- The **`RECEIVE-FIND`** statement is a **`START`** statement for non-root nodes.
- The **`RECEIVE-REPORT`** statement collects the result of a *find* request to a child node. If all children are accounted for, the minimum value computed so far (considering the node's own value) is sent to the node's parent.
- The **`ROOT-RECEIVE-REPORT`** statement is a special **`RECEIVE-REPORT`** statement for the root, since the root has no parent node.

As in the mutual exclusion example, the state of the system is a list of pairs (an *association list* or *alist*) which are accessed by the **ASSOC** function. **(ASSOC KEY ALIST)** returns the first pair in **ALIST** such that the **CAR** of that pair is **KEY**. If no such pair exists, then **ASSOC** returns **F**. Each of these pairs represents a binding of a variable name to that variable's value.

Each node possesses several variables. These are:

- **NODE-VALUE** is the value of the node. It will be constant.
- **STATUS** indicates whether the node has begun searching its subtree for its minimum value.
- **OUTSTANDING** is the number of children nodes that have not yet responded to this node's *find* request. When the node starts, **OUTSTANDING** equals the number of children of that node.
- **FOUND-VALUE** is the minimum value computed so far. Each time a node receives a response to a *find* request, it assigns **FOUND-VALUE** the minimum of its current value and the response.

In addition, there is a channel from every node to each of its children, and a channel from every node to its parent. Messages passed between nodes are buffered in these channels.

A variable name is of the form **(CONS VAR-NAME NODE-NAME)** where **VAR-NAME** is one of the variable names given above, and **NODE-NAME** is the name of the node owning that variable (the pair serves as the key for that variable in the state). We require that all node names be numbers, so there is no possible confusion between variable names and channel names. A channel from node **1** to node **2** would have the name **(CONS 1 2)**. The channel in the opposite direction would have the name **(CONS 2 1)**. (As with variable names, the pair serves as the key for that channel in the state.)

We now define several functions which look up a node's variables and manipulate channels. We define a utility function that abbreviates **CDR** of **ASSOC**:

Definition:

```
(VALUE KEY STATE)
   =
(CDR (ASSOC KEY STATE))
```

The function **STATUS** finds the status of a node. The key for each node's status variable is the pair **(CONS 'STATUS NODE)** where **NODE** is the node's name.

Definition:

```
(STATUS NODE STATE)
   =
(VALUE (CONS 'STATUS NODE) STATE)
```

The function **NODE-VALUE** returns the value of the node, which will be constant.

Definition:

```
(NODE-VALUE NODE STATE)
   =
(VALUE (CONS 'NODE-VALUE NODE) STATE)
```

The function **FOUND-VALUE** returns the value of the variable **FOUND-VALUE**, for a particular node. This is the subtree's minimum value computed so far.

Definition:

```
(FOUND-VALUE NODE STATE)
   =
(VALUE (CONS 'FOUND-VALUE NODE) STATE)
```

OUTSTANDING returns the value of the the variable **OUTSTANDING**, which represents the number of children from which the node has still not received a response.

Definition:

```
(OUTSTANDING NODE STATE)
   =
(VALUE (CONS 'OUTSTANDING NODE) STATE)
```

Channels are accessed using several functions. **CHANNEL** returns the contents of a channel. The name of a channel is a pair **(CONS FROM TO)**, where **FROM** and **TO** are the names of the sending and receiving nodes, respectively.
Definition:

```
(CHANNEL NAME STATE)
   =
(VALUE NAME STATE)
```

The function **EMPTY** tests whether a channel is empty:
Definition:

```
(EMPTY NAME STATE)
   =
(NOT (LISTP (CHANNEL NAME STATE)))
```

The first message on a non-empty channel is identified using the function **HEAD**:
Definition:

```
(HEAD NAME STATE)
   =
(CAR (CHANNEL NAME STATE))
```

A message is sent upon a channel using the function **SEND**. **SEND** returns a value that is equal to the old channel appended with the new message.
Definition:

```
(SEND CHANNEL MESSAGE STATE)
   =
(APPEND (CHANNEL CHANNEL STATE)
        (LIST MESSAGE))
```

A non-empty channel is shortened using the **RECEIVE** function:
Definition:

```
(RECEIVE CHANNEL STATE)
   =
(CDR (CHANNEL CHANNEL STATE))
```

We additionally define several functions which identify a node's parent, a node's children, the names of the nodes in the tree, and whether the tree is truly a tree. A tree is a data structure with the following form: **(CONS ROOT (LIST SUBTREE-1 SUBTREE-2** ... **))** where each subtree is another non-empty tree. The empty tree is any value that is not a **LIST**. For example a tree consisting of only a root named **1** is **'(1)**, while a tree consisting of a root named **'1** with children **'2** and **'3** is **'(1 (2) (3))**.

The nodes in a tree are identified by the function **NODES**, which uses the auxiliary function **NODES-REC**:
Definition:

```
(NODES TREE)
   =
(NODES-REC 'TREE TREE)
```

NODES-REC is function that most naturally would have been written as two mutually recursive functions, one which traverses trees and the other which traverses forests. However, since the Boyer-Moore logic does not admit mutually recursive definitions, we add a flag to **NODES-REC**'s argument list. Depending on the value of the flag, the function behaves as one or the other of the mutually recursive functions that we wanted.
Definition:

```
(NODES-REC FLAG TREE)
   =
(IF (LISTP TREE)
    (IF (EQUAL FLAG 'TREE)
        (CONS (CAR TREE)
              (NODES-REC 'FOREST (CDR TREE)))
      (APPEND (NODES-REC 'TREE (CAR TREE))
              (NODES-REC 'FOREST (CDR TREE))))
  NIL)
```

There are two criteria to determine whether an alleged tree is truly a tree. One is content: there must not be duplicate node names. The other is structural: does the tree possess the structure described earlier. The function **SETP** checks whether its argument possesses duplicate elements. All node names in a tree must be unique because each name identifies a single node:

Definition:

```
(SETP LIST)
   =
(IF (LISTP LIST)
    (IF (MEMBER (CAR LIST) (CDR LIST))
        F
      (SETP (CDR LIST)))
  T)
```

The structural requirement is checked using the function **PROPER-TREE**. It is also most naturally described as a mutually recursive function: the typical call is **(PROPER-TREE 'TREE TREE)**:

Definition:

```
(PROPER-TREE FLAG TREE)
   =
(IF (EQUAL FLAG 'TREE)
    (IF (LISTP TREE)
        (PROPER-TREE 'FOREST (CDR TREE))
      F)
  (IF (LISTP TREE)
      (AND (PROPER-TREE 'TREE (CAR TREE))
           (PROPER-TREE 'FOREST (CDR TREE)))
    (EQUAL TREE NIL)))
```

A fortunate consequence of the definition of **PROPER-TREE** is that a tree cannot be empty (a forest may be empty, however). This is good, because our correctness properties are false for empty trees. The sort of tree that we will deal with here is a **PROPER-TREE** with no duplicate nodes, whose nodes are all numbers. This is tested for by the function **TREEP**:

Definition:

```
(TREEP TREE)
   =
(AND (SETP (NODES TREE))
     (ALL-NUMBERPS (NODES TREE))
     (PROPER-TREE 'TREE TREE))
```

where **ALL-NUMBERPS** is defined in the obvious way.

The term **(CHILDREN NODE TREE)** returns a list of the names of the children of **NODE** in tree **TREE**. It is also defined by two functions:

Definition:

```
(CHILDREN NODE TREE)
   =
(CHILDREN-REC 'TREE NODE TREE)
```

Definition:

```
(CHILDREN-REC FLAG NODE TREE)
   =
(IF (LISTP TREE)
    (IF (EQUAL FLAG 'TREE)
        (IF (EQUAL (CAR TREE) NODE)
            (APPEND (ROOTS (CDR TREE))
                    (CHILDREN-REC 'FOREST NODE
                                  (CDR TREE)))
          (CHILDREN-REC 'FOREST NODE (CDR TREE)))
      (APPEND (CHILDREN-REC 'TREE NODE (CAR TREE))
              (CHILDREN-REC 'FOREST NODE (CDR TREE))))
  NIL)
```

The function **ROOTS** returns the roots of a forest:

Definition:

```
(ROOTS FOREST)
   =
(IF (LISTP FOREST)
    (CONS (CAAR FOREST)
          (ROOTS (CDR FOREST)))
  FOREST)
```

One might argue that the definition of **CHILDREN-REC** may be simplified, by observing that once children are found, the recursion may terminate. But there does not appear to be any good way to do this in both the tree and forest alternatives of the function, so this observation is best exploited in a theorem.

Similarly, **(PARENT NODE TREE)** returns the name of the parent of **NODE** in tree **TREE**. If **NODE** is the root, then **(PARENT NODE TREE)** returns some arbitrary value (we choose **0**):

Definition:

```
(PARENT NODE TREE)
   =
(CAR (PARENT-REC 'TREE NODE TREE))
```

Definition:

```
(PARENT-REC FLAG NODE TREE)
   =
(IF (LISTP TREE)
    (IF (EQUAL FLAG 'TREE)
        (IF (MEMBER NODE (ROOTS (CDR TREE)))
            (CONS (CAR TREE)
                  (PARENT-REC 'FOREST NODE (CDR TREE)))
          (PARENT-REC 'FOREST NODE (CDR TREE)))
      (APPEND (PARENT-REC 'TREE NODE (CAR TREE))
              (PARENT-REC 'FOREST NODE (CDR TREE))))
  NIL)
```

We are now able to define the transitions of nodes in the tree. **START** statements use the following function:

Definition:

```
(START OLD NEW ROOT TO-CHILDREN)
   =
(IF (EQUAL (STATUS ROOT OLD) 'NOT-STARTED)
    (AND (EQUAL (STATUS ROOT NEW) 'STARTED)
         (EQUAL (FOUND-VALUE ROOT NEW)
                (NODE-VALUE ROOT OLD))
         (EQUAL (OUTSTANDING ROOT NEW)
                (LENGTH TO-CHILDREN))
         (SEND-FIND TO-CHILDREN OLD NEW)
         (CHANGED OLD NEW
                  (APPEND
                   (LIST (CONS 'STATUS ROOT)
                         (CONS 'FOUND-VALUE ROOT)
                         (CONS 'OUTSTANDING ROOT))
                   TO-CHILDREN)))
  (CHANGED OLD NEW NIL))
```

(**SEND-FIND** will be defined later.) This function defines the permitted transitions between old and new states under the **START** statement. **(START OLD NEW ROOT TO-CHILDREN)** states that for a start action to take place on state **OLD**, the status of the node **ROOT** in **OLD** must be **'NOT-STARTED**. In that case, a **START** action does the following:

- The new status of node **ROOT** is **'STARTED**.
- **ROOT**'s variable **FOUND-VALUE** is set to **ROOT**'s **NODE-VALUE**.
- **OUTSTANDING** is set to the number of **ROOT**'s children.
- *find* messages are sent to each of **ROOT**'s children, using the function **SEND-FIND**.

Furthermore, using the function **CHANGED**, all variables and channels, except for the ones mentioned above, are unchanged (see section 4.4, page 42).

The function **SEND-FIND** sends a *find* message upon each of the channels in the list **TO-CHILDREN**. **TO-CHILDREN** must be a list of the names of channels between the node **ROOT** and each of its children. The definition of **SEND-FIND** is:

Definition:

```
(SEND-FIND TO-CHILDREN OLD NEW)
   =
(IF (LISTP TO-CHILDREN)
    (AND (EQUAL (CHANNEL (CAR TO-CHILDREN) NEW)
                (SEND (CAR TO-CHILDREN) 'FIND OLD))
         (SEND-FIND (CDR TO-CHILDREN) OLD NEW))
  T)
```

The statement **RECEIVE-FIND** is the **START** statement for non-root nodes and is specified by the function **RECEIVE-FIND**, which is defined as follows:

Definition:

```
(RECEIVE-FIND OLD NEW NODE FROM-PARENT
              TO-PARENT TO-CHILDREN)
   =
(IF (EQUAL (HEAD FROM-PARENT OLD) 'FIND)
    (AND (EQUAL (CHANNEL FROM-PARENT NEW)
                (RECEIVE FROM-PARENT OLD))
         (EQUAL (STATUS NODE NEW) 'STARTED)
         (EQUAL (FOUND-VALUE NODE NEW)
                (NODE-VALUE NODE OLD))
         (EQUAL (OUTSTANDING NODE NEW)
                (LENGTH TO-CHILDREN))
         (SEND-FIND TO-CHILDREN OLD NEW)
         (EQUAL (CHANNEL TO-PARENT NEW)
                (IF (ZEROP (LENGTH TO-CHILDREN))
                    (SEND TO-PARENT
                          (NODE-VALUE NODE OLD)
                          OLD)
                  (CHANNEL TO-PARENT OLD)))
         (CHANGED OLD NEW
                  (APPEND
                   (LIST FROM-PARENT TO-PARENT
                         (CONS 'STATUS NODE)
                         (CONS 'FOUND-VALUE NODE)
                         (CONS 'OUTSTANDING NODE))
                   TO-CHILDREN)))
  (CHANGED OLD NEW NIL))
```

RECEIVE-FIND states that if node **NODE** receives a *find* message along its incoming channel (from its parent) then it should initiate a **START** action, with the following twist: if it has no children, then it should immediately send its **NODE-VALUE** to its parent as the minimum value of its subtree.

The third statement is **RECEIVE-REPORT**. It is defined as follows:

Definition:

```
(RECEIVE-REPORT OLD NEW NODE FROM-CHILD TO-PARENT)
   =
(IF (EMPTY FROM-CHILD OLD)
    (CHANGED OLD NEW NIL)
  (AND (EQUAL (CHANNEL FROM-CHILD NEW)
              (RECEIVE FROM-CHILD OLD))
       (EQUAL (FOUND-VALUE NODE NEW)
              (MIN (FOUND-VALUE NODE OLD)
                   (HEAD FROM-CHILD OLD)))
       (EQUAL (OUTSTANDING NODE NEW)
              (SUB1 (OUTSTANDING NODE OLD)))
       (EQUAL (CHANNEL TO-PARENT NEW)
              (IF (ZEROP (OUTSTANDING NODE NEW))
                  (SEND TO-PARENT
                        (FOUND-VALUE NODE NEW)
                        OLD)
                (CHANNEL TO-PARENT OLD)))
       (CHANGED OLD NEW
                (LIST FROM-CHILD TO-PARENT
                      (CONS 'OUTSTANDING NODE)
                      (CONS 'FOUND-VALUE NODE)))))
```

RECEIVE-REPORT handles the responses to a *find* request sent to a node's child. **RECEIVE-REPORT** takes an **OLD** state where the **FROM-CHILD** channel holds a message for **NODE**. This message is interpreted as the the minimum value of that child's subtree. If this value is less than the value stored at **FOUND-VALUE** then **FOUND-VALUE** is assigned the smaller value. The number of **OUTSTANDING** responses is decremented by one; if the result is zero, then the new **FOUND-VALUE** is sent to the parent node as the minimum value of the subtree.

MIN is defined in the following way:
Definition:

```
(MIN X Y)
   =
(IF (LESSP X Y)
    (FIX X)
  (FIX Y))
```

ROOT-RECEIVE-REPORT is the **RECEIVE-REPORT** statement for the root. It is defined as follows:
Definition:

```
(ROOT-RECEIVE-REPORT OLD NEW ROOT FROM-CHILD)
   =
(IF (EMPTY FROM-CHILD OLD)
    (CHANGED OLD NEW NIL)
  (AND (EQUAL (CHANNEL FROM-CHILD NEW)
              (RECEIVE FROM-CHILD OLD))
       (EQUAL (FOUND-VALUE ROOT NEW)
              (MIN (FOUND-VALUE ROOT OLD)
                   (HEAD FROM-CHILD OLD)))
       (EQUAL (OUTSTANDING ROOT NEW)
              (SUB1 (OUTSTANDING ROOT OLD)))
       (CHANGED OLD NEW
                (LIST FROM-CHILD
                      (CONS 'OUTSTANDING ROOT)
                      (CONS 'FOUND-VALUE ROOT)))))
```

The only difference between **ROOT-RECEIVE-REPORT** and **RECEIVE-REPORT** is that **ROOT-RECEIVE-REPORT** does not send a message to its parent when no outstanding responses remain.

These four statements specify the types of actions that the program may take. However, they are generic statements, and need to be instantiated for each node in the tree in order to form the program.

5.2 The Program

Recall that a program is a list of statements. Each statement is a list **(CONS FUNC ARGS)** where **FUNC** is the name of a function symbol. In the case of this minimum value program, each statement is a list whose first element is of one of the four symbols **START**, **RECEIVE-FIND**, **RECEIVE-REPORT**, and **ROOT-RECEIVE-REPORT**. The entire program is a list of such statements.

We form the program by first collecting the groups of statements of each type. We collect one **RECEIVE-FIND** statement for each non-root node in the tree in the following manner:

Definition:

```
(RECEIVE-FIND-PRG NODES TREE)
   =
(IF (LISTP NODES)
    (CONS (LIST 'RECEIVE-FIND
                (CAR NODES)
                (CONS (PARENT (CAR NODES) TREE)
                      (CAR NODES))
                (CONS (CAR NODES)
                      (PARENT (CAR NODES) TREE))
                (RFP (CAR NODES)
                     (CHILDREN (CAR NODES) TREE)))
          (RECEIVE-FIND-PRG (CDR NODES)
                            TREE))
  NIL)
```

Each **RECEIVE-FIND** statement is of the form **'(RECEIVE-FIND NODE FROM-PARENT TO-PARENT TO-CHILDREN)**, where **FROM-PARENT** is the name of the channel from the parent to the node, and **TO-PARENT** is the channel from the node to the parent. **TO-CHILDREN** is a list of the names of the channels from the node to its children; this list is formed by the function **RFP**, which is defined as follows:

Definition:

```
(RFP NODE CHILDREN)
   =
(IF (LISTP CHILDREN)
    (CONS (CONS NODE (CAR CHILDREN))
          (RFP NODE (CDR CHILDREN)))
  NIL)
```

Using **RFP**, the single **START** statement may be formed as well, using the function **START-PRG**:

Definition:

```
(START-PRG ROOT TREE)
   =
(LIST (LIST 'START ROOT
            (RFP ROOT (CHILDREN ROOT TREE))))
```

RECEIVE-REPORT statements are formed in two stages. The first collects all the **RECEIVE-REPORT** statements for a node, from each of its children. The second stage collects this group for each non-root node. The first stage is formed by the function **RRP**:

Definition:

```
(RRP NODE CHILDREN PARENT)
   =
(IF (LISTP CHILDREN)
    (CONS (LIST 'RECEIVE-REPORT
                NODE
                (CONS (CAR CHILDREN) NODE)
                (CONS NODE PARENT))
          (RRP NODE (CDR CHILDREN) PARENT))
  NIL)
```

Each **RECEIVE-REPORT** statement is of the form **'(RECEIVE-REPORT NODE FROM-CHILD TO-PARENT)**, where **FROM-CHILD** is the name of the channel connecting the child to the node and **TO-PARENT** is the name of the channel connecting the node to its parent. **RRP** must be collected for each non-root node in the tree. This is done by the function **RECEIVE-REPORT-PRG**:

Definition:

```
(RECEIVE-REPORT-PRG NODES TREE)
   =
(IF (LISTP NODES)
    (APPEND (RRP (CAR NODES)
                 (CHILDREN (CAR NODES) TREE)
                 (PARENT (CAR NODES) TREE))
            (RECEIVE-REPORT-PRG (CDR NODES) TREE))
  NIL)
```

RECEIVE-REPORT-PRG collects all the **RECEIVE-REPORT** statements for the program.

The set of **ROOT-RECEIVE-REPORT** statements is formed using a function modeled after **RRP**:
Definition:

```
(RRRP ROOT CHILDREN)
   =
(IF (LISTP CHILDREN)
    (CONS (LIST 'ROOT-RECEIVE-REPORT
                ROOT
                (CONS (CAR CHILDREN) ROOT))
          (RRRP ROOT (CDR CHILDREN)))
  NIL)
```

This function is captured more conveniently by **ROOT-RECEIVE-REPORT-PRG**:
Definition:

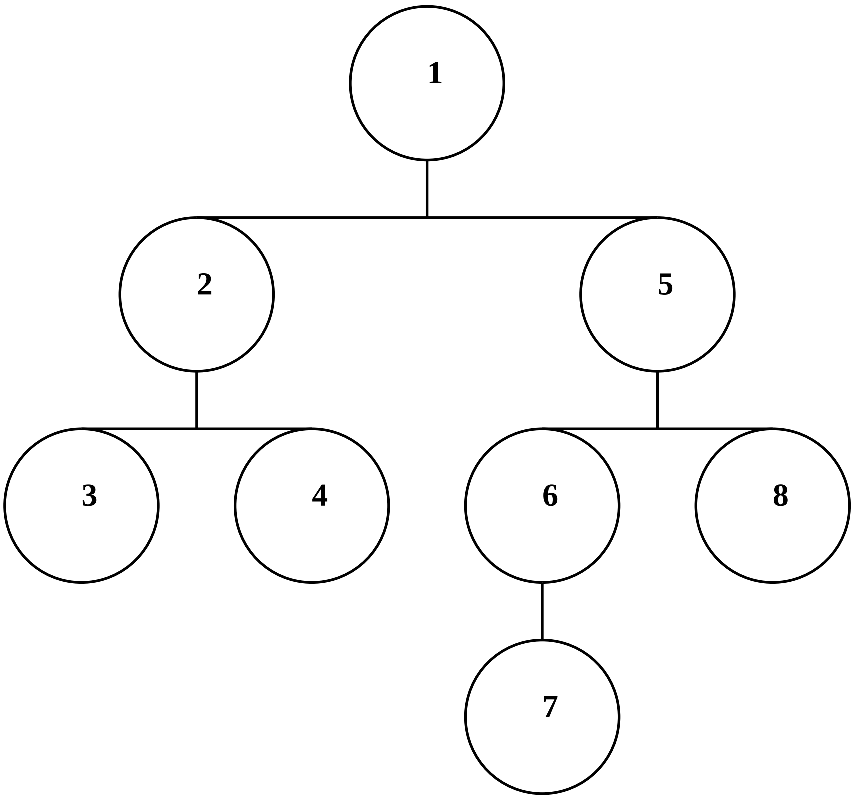
```
(ROOT-RECEIVE-REPORT-PRG ROOT TREE)
   =
(RRRP ROOT (CHILDREN ROOT TREE))
```

The entire program is the concatenation of all the program parts above. This is accomplished by the function **TREE-PRG**:
Definition:

```
(TREE-PRG TREE)
   =
(APPEND
 (START-PRG (CAR TREE) TREE)
 (APPEND
  (ROOT-RECEIVE-REPORT-PRG (CAR TREE)
                           TREE)
  (APPEND
   (RECEIVE-FIND-PRG (CDR (NODES TREE))
                     TREE)
   (RECEIVE-REPORT-PRG (CDR (NODES TREE))
                       TREE))))
```

The term **(TREE-PRG TREE)** collects all the instances of the four types of statements required to specify the program.

For example, imagine the tree '(1 (2 (3) (4)) (5 (6 (7)) (8))). This may be represented by the following picture:



The program for computing the minimum node of this tree is **`(TREE-PRG '(1 (2 (3) (4)) (5 (6 (7)) (8))))`**, which equals:

```
'((START 1 ((1 . 2) (1 . 5)))
  (ROOT-RECEIVE-REPORT 1 (2 . 1))
  (ROOT-RECEIVE-REPORT 1 (5 . 1))
  (RECEIVE-FIND 2 (1 . 2) (2 . 1) ((2 . 3) (2 . 4)))
  (RECEIVE-FIND 3 (2 . 3) (3 . 2) NIL)
  (RECEIVE-FIND 4 (2 . 4) (4 . 2) NIL)
  (RECEIVE-FIND 5 (1 . 5) (5 . 1) ((5 . 6) (5 . 8)))
  (RECEIVE-FIND 6 (5 . 6) (6 . 5) ((6 . 7)))
  (RECEIVE-FIND 7 (6 . 7) (7 . 6) NIL)
  (RECEIVE-FIND 8 (5 . 8) (8 . 5) NIL)
  (RECEIVE-REPORT 2 (3 . 2) (2 . 1))
  (RECEIVE-REPORT 2 (4 . 2) (2 . 1))
  (RECEIVE-REPORT 5 (6 . 5) (5 . 1))
  (RECEIVE-REPORT 5 (8 . 5) (5 . 1))
  (RECEIVE-REPORT 6 (7 . 6) (6 . 5)))
```

where **`(1 . 2)`** is an abbreviation[24] for **`(CONS 1 2)`**.

[24]Lisp dot notation [85].

5.3 The Correctness Specification

To prove that the program **(TREE-PRG TREE)** correctly implements the minimum value algorithm it is necessary to state the correctness condition. First we state the initial conditions. Initially, all the channels are empty, and the status of each node is **'NOT-STARTED**. The function that tests whether every node has status **'NOT-STARTED** is:

Definition:

```
(NOT-STARTED NODES STATE)
   =
(IF (LISTP NODES)
    (AND (EQUAL (STATUS (CAR NODES) STATE)
                'NOT-STARTED)
         (NOT-STARTED (CDR NODES) STATE))
  T)
```

The term that tests whether every channel is empty is **(ALL-EMPTY (ALL-CHANNELS TREE) STATE)**. **ALL-CHANNELS** returns a list containing the name of every channel in the system. **ALL-EMPTY** checks whether all channels are empty and is defined as follows:

Definition:

```
(ALL-EMPTY CHANNELS STATE)
   =
(IF (LISTP CHANNELS)
    (AND (EMPTY (CAR CHANNELS) STATE)
         (ALL-EMPTY (CDR CHANNELS) STATE))
  T)
```

ALL-CHANNELS is defined as follows:

Definition:

```
(ALL-CHANNELS TREE)
   =
(APPEND (UP-LINKS (CDR (NODES TREE)) TREE)
        (DOWN-LINKS (NODES TREE) TREE))
```

where **UP-LINKS** and **DOWN-LINKS** are defined as follows:

Definition:

```
(UP-LINKS NODES TREE)
   =
(IF (LISTP NODES)
    (CONS (CONS (CAR NODES) (PARENT (CAR NODES) TREE))
          (UP-LINKS (CDR NODES) TREE))
  NIL)
```

Definition:

```
(DOWN-LINKS NODES TREE)
   =
(IF (LISTP NODES)
    (APPEND (DOWN-LINKS-1 (CAR NODES)
                          (CHILDREN (CAR NODES)
                                    TREE))
            (DOWN-LINKS (CDR NODES) TREE))
  NIL)
```

Definition:[25]

```
(DOWN-LINKS-1 PARENT CHILDREN)
   =
(IF (LISTP CHILDREN)
    (CONS (CONS PARENT (CAR CHILDREN))
          (DOWN-LINKS-1 PARENT (CDR CHILDREN)))
  NIL)
```

[25] **DOWN-LINKS-1** is indeed identical to **RFP**, but this, unfortunately, was noticed well into the proof. Since different rewrite rules may have been proved about each function, it would have been non-trivial to replace either function by the other.

The initial condition is the term **`(AND (NOT-STARTED (NODES TREE) STATE) (ALL-EMPTY (ALL-CHANNELS TREE) STATE))`**. The correctness condition is the term **`(CORRECT TREE STATE)`** where **`CORRECT`** is defined as:

Definition:

```
(CORRECT TREE STATE)
   =
(EQUAL (FOUND-VALUE (CAR TREE) STATE)
       (MIN-NODE-VALUE (CDR (NODES TREE)) STATE
                       (NODE-VALUE (CAR TREE) STATE)))
```

where **`MIN-NODE-VALUE`** is defined as:

Definition:

```
(MIN-NODE-VALUE NODES STATE MIN)
   =
(IF (LISTP NODES)
    (MIN (NODE-VALUE (CAR NODES) STATE)
         (MIN-NODE-VALUE (CDR NODES) STATE MIN))
  MIN)
```

`(CORRECT TREE STATE)` checks whether the value of the root's **`FOUND-VALUE`** variable is equal to the minimum of all the nodes' **`NODE-VALUE`** variables. Of course, it is necessary to prove that the **`NODE-VALUE`** variables are constants. That will be an invariant.

The correctness condition is specified as a **`LEADS-TO`**. It is:

Theorem: Correctness-Condition

```
(IMPLIES (AND (TREEP TREE)
              (INITIAL-CONDITION
               '(AND (ALL-EMPTY
                      (QUOTE ,(ALL-CHANNELS TREE))
                      STATE)
                     (NOT-STARTED (QUOTE ,(NODES TREE))
                                  STATE))
               (TREE-PRG TREE)))
         (LEADS-TO '(TRUE)
                   '(CORRECT (QUOTE ,TREE) STATE)
                   (TREE-PRG TREE)))
```

The conclusion implies that a state will eventually be reached where the correctness condition **`(CORRECT TREE STATE)`** is true. The hypotheses state that the tree is really a tree (no duplicate nodes, all nodes are numbers, and the tree is proper), and that the initial state satisfies the initial conditions (all statuses are **`NOT-STARTED`** and all channels are empty). Since **`TREE`** is a variable (section 1.2.2, page 7), this **`LEADS-TO`** statement holds for all the programs for all trees.

The invariant states that **`NODE-VALUE`**'s are constant:

Theorem: Node-Values-Constant-Invariant

```
(IMPLIES (AND (INITIAL-CONDITION
               '(AND (ALL-EMPTY
                      (QUOTE ,(ALL-CHANNELS TREE))
                      STATE)
                     (AND (NOT-STARTED
                           (QUOTE ,(NODES TREE))
                           STATE)
                          (EQUAL (NODE-VALUE
                                  (QUOTE ,NODE)
                                  STATE)
                                 (QUOTE ,K))))
               (TREE-PRG TREE))
              (TREEP TREE)
              (MEMBER NODE (NODES TREE)))
         (INVARIANT '(EQUAL (NODE-VALUE (QUOTE ,NODE)
                                        STATE)
                            (QUOTE ,K))
                     (TREE-PRG TREE)))
```

This invariant states that if a **NODE-VALUE** variable has value **K**, then at any point in the execution, that **NODE-VALUE** variable will still have value **K**. This implies that **NODE-VALUE** variables are constant. This also implies that the minimum of all **NODE-VALUE** variables is constant, but this was not proved.

The proof of these theorems is discussed in the next section.

5.4 The Proof of Correctness

In this program, there are four types of statements, which are instantiated in various ways to account for every node in the tree. Therefore, when proving certain properties, it is simpler to prove that the property holds for an arbitrary instance of each of the statements, rather than for each statement that is truly in the program. As we did with the theorem **Member-Me-Prg** (section 4.4, page 42), a useful theorem is one that rewrites an inductive term like **(MEMBER E (TREE-PRG TREE))** to properly constrained instances of program statements. That is, assuming that **E** is a statement in the program is identical to knowing another set of facts about **E**; those facts happen to be easier to reason with. The appropriate theorem for this program is:

Theorem: Member-Tree-Prg [26]

```
(EQUAL (MEMBER STATEMENT (TREE-PRG TREE))
;(START ROOT TO-CHILDREN)
       (OR (AND (EQUAL (CAR STATEMENT) 'START)
                (EQUAL (CADR STATEMENT) (CAR TREE))
                (EQUAL (CADDR STATEMENT)
                       (RFP (CAR TREE)
                            (CHILDREN (CAR TREE) TREE)))
                (EQUAL (CDDDR STATEMENT) NIL))
;(ROOT-RECEIVE-REPORT ROOT FROM-CHILD)
           (AND (EQUAL (CAR STATEMENT) 'ROOT-RECEIVE-REPORT)
                (EQUAL (CADR STATEMENT) (CAR TREE))
                (LISTP (CADDR STATEMENT))
                (MEMBER (CAADDR STATEMENT)
                        (CHILDREN (CAR TREE) TREE))
                (EQUAL (CDADDR STATEMENT) (CAR TREE))
                (EQUAL (CDDDR STATEMENT) NIL))
;(RECEIVE-FIND NODE FROM-PARENT TO-PARENT TO-CHILDREN)
           (AND (EQUAL (CAR STATEMENT) 'RECEIVE-FIND)
                (MEMBER (CADR STATEMENT) (CDR (NODES TREE)))
                (LISTP (CADDR STATEMENT))
                (EQUAL (CAADDR STATEMENT)
                       (PARENT (CADR STATEMENT) TREE))
                (EQUAL (CDADDR STATEMENT) (CADR STATEMENT))
                (LISTP (CADDDR STATEMENT))
                (EQUAL (CAADDDR STATEMENT) (CADR STATEMENT))
                (EQUAL (CDADDDR STATEMENT)
                       (PARENT (CADR STATEMENT) TREE))
                (EQUAL (CADDDDR STATEMENT)
                       (RFP (CADR STATEMENT)
                            (CHILDREN (CADR STATEMENT)
                                      TREE)))
                (EQUAL (CDDDDDR STATEMENT) NIL))
;(RECEIVE-REPORT NODE FROM-CHILD TO-PARENT)
           (AND (EQUAL (CAR STATEMENT) 'RECEIVE-REPORT)
                (MEMBER (CADR STATEMENT) (CDR (NODES TREE)))
                (LISTP (CADDR STATEMENT))
                (MEMBER (CAADDR STATEMENT)
                        (CHILDREN (CADR STATEMENT) TREE))
                (EQUAL (CDADDR STATEMENT) (CADR STATEMENT))
                (LISTP (CADDDR STATEMENT))
                (EQUAL (CAADDDR STATEMENT) (CADR STATEMENT))
                (EQUAL (CDADDDR STATEMENT)
                       (PARENT (CADR STATEMENT) TREE))
                (EQUAL (CDDDDR STATEMENT) NIL))))
```

Although this theorem looks frightening, its construction is straightforward. Assuming that **E** is a statement in the program means that **(CAR E)** has one of the following values: **START**, **ROOT-RECEIVE-REPORT**, **RECEIVE-FIND**, or **RECEIVE-REPORT**. The syntax for each of these statements is highlighted in the comment (indicated by a semicolon) preceding each **AND** block in the theorem. Depending on which type of statement **E** represents, the remaining parameters in the list are equally well defined. For example, in the **START** and **ROOT-RECEIVE-REPORT** statements, the second element of the list is the name of the root node. In the **RECEIVE-FIND** and **RECEIVE-REPORT** statements, the second element of the list is the name of some non-root node in the tree. The final **CDR** of every statement is **NIL**. Careful analysis of the parameters in each statement yields the rest of the information. The important part of this theorem is that it is an equality: knowing that **E** is a statement in the program is equivalent to knowing that **E** has the specific syntax of one of the statements in the program. The latter information is much more easily used by the theorem prover, since it lends itself to case analysis.

---

[26] Actually, this theorem was always disabled and the function **(TREE-PRG TREE)** was allowed to expand to its four components. Then, four rewrite rules, components of this one, were allowed to simplify the components. But, this theorem gets the point across.

This theorem rewrites formulas whose proofs are inductive (because of **MEMBER**) to formulas whose proofs proceed by case analysis on statement types in the program. Each type of statement will be analyzed, assuming the appropriate constraints on its arguments.

We must prove that each of the statements in the program is **TOTAL**. This is proved in the following theorem:

Theorem: Total-Tree-Prg

```
(IMPLIES (TREEP TREE)
         (TOTAL (TREE-PRG TREE)))
```

This theorem is proved by inventing witness functions that compute a valid **NEW** state for any **OLD** state such that the **NEW** state satisfies each relation in the program (each of the four types of program statements).

The proof of the invariant is trivial. One proves that each relation implies that the value of **NODE-VALUE** variables remain unchanged. This implies that every statement in the program keeps those variables constant.

We now concentrate on the proof of the **LEADS-TO** statement. We introduce a new formula called the *augmented correctness condition*. The hypothesis of this formula is a termination condition and the conclusion is the original correctness statement **(CORRECT TREE STATE)**. We then prove two theorems:

- The augmented correctness condition is a consequence of another invariant of the program.
- The termination condition in the augmented correctness condition is eventually reached in the computation.

These two facts imply that the correctness condition is reached as well. This will complete the proof. We now present the termination condition, the augmented correctness condition, and the other invariant.

The termination condition is a well-founded measure that decreases upon every (non no-op) action in the program. The term **(TOTAL-OUTSTANDING (NODES TREE) TREE STATE)** sums the number of **'NOT-STARTED** nodes and the number of responses that have yet to occur. Since every (non no-op) action either starts a node or responds to a *find* request, this measure eventually decreases to zero. At that point, the computation reaches a fixed point, and the root node has the tree's minimum **NODE-VALUE** in its **FOUND-VALUE** variable. The definition of **TOTAL-OUTSTANDING** is:

Definition:

```
(TOTAL-OUTSTANDING NODES TREE STATE)
   =
(IF (LISTP NODES)
    (PLUS (TOTAL-OUTSTANDING (CDR NODES) TREE STATE)
          (IF (EQUAL (STATUS (CAR NODES) STATE)
                     'STARTED)
              (OUTSTANDING (CAR NODES) STATE)
            (ADD1 (LENGTH (CHILDREN (CAR NODES)
                                    TREE)))))
  0)
```

The augmented correctness condition is:

```
    (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                  (SETP (NODES-REC 'TREE TREE))
                  (EQUAL (TOTAL-OUTSTANDING (NODES TREE)
                                            TREE STATE)
                         0))
             (CORRECT TREE STATE))
```

Notice that the first two hypotheses in this formula are assumed in the **LEADS-TO** correctness condition. If we prove that the termination condition is reached, and that the augmented correctness condition is a consequence of an invariant that holds on the initial state, then we know that the correctness condition **(CORRECT TREE STATE)** is also reached.

The new invariant is somewhat complicated and requires the introduction of seven new functions:

Definition:

```
(DL DOWN-LINKS STATE)
   =
(IF (LISTP DOWN-LINKS)
    (AND (OR (AND (EMPTY (CAR DOWN-LINKS) STATE)
                  (EQUAL (STATUS (CAAR DOWN-LINKS)
                                 STATE)
                         (STATUS (CDAR DOWN-LINKS)
                                 STATE)))
             (AND (EQUAL (CHANNEL (CAR DOWN-LINKS)
                                  STATE)
                         (LIST 'FIND))
                  (EQUAL (STATUS (CAAR DOWN-LINKS)
                                 STATE)
                         'STARTED)
                  (EQUAL (STATUS (CDAR DOWN-LINKS)
                                 STATE)
                         'NOT-STARTED)))
         (DL (CDR DOWN-LINKS) STATE))
  T)
```

DL is the invariant between the contents of down-links (channels between parents and children) and the statuses of the sending and receiving nodes.

Definition:

```
(UL UP-LINKS STATE)
   =
(IF (LISTP UP-LINKS)
    (AND (OR (EMPTY (CAR UP-LINKS) STATE)
             (AND (EQUAL (CHANNEL (CAR UP-LINKS) STATE)
                         (LIST (FOUND-VALUE
                                 (CAAR UP-LINKS)
                                 STATE)))
                  (DONE (CAAR UP-LINKS) STATE)))
         (UL (CDR UP-LINKS) STATE))
  T)
```

where **DONE** is:

Definition:

```
(DONE NODE STATE)
   =
(AND (EQUAL (STATUS NODE STATE) 'STARTED)
     (ZEROP (OUTSTANDING NODE STATE)))
```

UL is the invariant between the contents of up-links (channels between children and parents) and the **FOUND-VALUE** of the child (the sender); this invariant guarantees that if a value is passed up to the parent, the child is done and the correct value is passed up.

Definition:

```
(NO NODES TREE STATE)
   =
(IF (LISTP NODES)
    (AND (IF (EQUAL (STATUS (CAR NODES) STATE)
                    'STARTED)
             (AND (EQUAL (OUTSTANDING (CAR NODES)
                                      STATE)
                         (NUMBER-NOT-REPORTED
                          (CHILDREN (CAR NODES)
                                    TREE)
                          (CAR NODES) STATE))
                  (EQUAL (FOUND-VALUE (CAR NODES)
                                      STATE)
                         (MIN-OF-REPORTED
                          (CHILDREN (CAR NODES) TREE)
                          (CAR NODES) STATE
                          (NODE-VALUE (CAR NODES)
                                      STATE))))
           T)
         (NO (CDR NODES) TREE STATE))
  T)
```

where **NUMBER-NOT-REPORTED** and **MIN-OF-REPORTED** are:
Definition:

```
(NUMBER-NOT-REPORTED CHILDREN PARENT STATE)
   =
(IF (LISTP CHILDREN)
    (IF (REPORTED (CAR CHILDREN) PARENT STATE)
        (NUMBER-NOT-REPORTED (CDR CHILDREN)
                             PARENT STATE)
      (ADD1 (NUMBER-NOT-REPORTED (CDR CHILDREN)
                                 PARENT STATE)))
  0)
```

where **REPORTED** is:
Definition:

```
(REPORTED NODE PARENT STATE)
   =
(AND (DONE NODE STATE)
     (EMPTY (CONS NODE PARENT) STATE))
```

Definition:

```
(MIN-OF-REPORTED CHILDREN PARENT STATE MIN)
   =
(IF (LISTP CHILDREN)
    (IF (REPORTED (CAR CHILDREN) PARENT STATE)
        (MIN (FOUND-VALUE (CAR CHILDREN) STATE)
             (MIN-OF-REPORTED (CDR CHILDREN) PARENT
                              STATE MIN))
      (MIN-OF-REPORTED (CDR CHILDREN) PARENT
                       STATE MIN))
  MIN)
```

NO implies that nodes update their **FOUND-VALUE** variables in a manner consistent with the values reported by its children.

Finally, we define the new invariant. The invariant is the term **(INV TREE STATE)** where **INV** is:

Definition:

```
(INV TREE STATE)
   =
(AND (DL (DOWN-LINKS (NODES TREE) TREE) STATE)
     (UL (UP-LINKS (CDR (NODES TREE)) TREE) STATE)
     (NO (NODES TREE) TREE STATE))
```

Now we state the theorem that shows that **(INV TREE STATE)** is an invariant:

Theorem: Inv-Is-Invariant

```
(IMPLIES (AND (INITIAL-CONDITION
              ‘(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS
                                         TREE))
                                STATE)
                    (NOT-STARTED (QUOTE ,(NODES TREE))
                                  STATE))
               (TREE-PRG TREE))
              (TREEP TREE))
         (INVARIANT ‘(INV (QUOTE ,TREE) STATE)
                     (TREE-PRG TREE)))
```

This theorem is proved by demonstrating both that the invariant is a consequence of the initial conditions, and that it is preserved by every statement in the program. The invariant is a consequence of the initial conditions:

Theorem: Initial-Conditions-Imply-Invariant

```
(IMPLIES (AND (PROPER-TREE ’TREE TREE)
              (ALL-EMPTY (ALL-CHANNELS TREE) STATE)
              (NOT-STARTED (NODES TREE) STATE))
         (INV TREE STATE))
```

The proof of this theorem is simple since initially all nodes are **’NOT-STARTED** and all channels are empty.

The proof that the invariant is preserved by every statement is more complicated. We decompose the proof in the following way: Each component of **INV** is a recursive function that collects all instances of its body. Therefore it is sufficient to prove that if **INV** holds before execution of a statement, then an arbitrary instance of each of the three functions **DL**, **UL**, and **NO** hold. Then, by induction on the nodes in the tree (actually on a list which is a subset of the nodes in the tree), we can demonstrate that **INV** holds subsequently.

Once **INV** is demonstrated to be an invariant, it is necessary to show that it implies the augmented correctness condition.

Theorem: Inv-Implies-Augmented-Correctness-Condition

```
(IMPLIES (AND (PROPER-TREE ’TREE TREE)
              (SETP (NODES-REC ’TREE TREE))
              (INV TREE STATE)
              (EQUAL (TOTAL-OUTSTANDING (NODES TREE)
                                         TREE STATE)
                     0))
         (CORRECT TREE STATE))
```

This theorem is proved by generalizing the tree to a forest of trees and is proved by induction. The inductive measure is the maximum depth of trees in the forest. The inductive step strips off the root of each tree, producing numerous forests for each tree in the forest. These are collected together. The maximum depth of any tree in the new forest is less than the maximum depth of any tree in the original forest. Hence, this is a valid induction. This is also a good generalization, since the intended case is a single tree, which can also be considered a forest of one tree.

Once the augmented correctness condition is proved, we prove the correctness condition by demonstrating the the measure **(TOTAL-OUTSTANDING (NODES TREE) TREE STATE)** decreases to **0**. This is proved by demonstrating that whenever **TOTAL-OUTSTANDING** is non-zero, it will decrease. If **TOTAL-OUTSTANDING** is non-zero, one of the following scenarios must be true:

- Root is **NOT-STARTED**.
- A down link is full.
- An up link to the root is full.
- An up link not to the root is full.

(If none of these are true, **TOTAL-OUTSTANDING** is **0**.) Not coincidentally, each of these cases corresponds to one of the statements in the program. Therefore, if any of these cases exist, then **TOTAL-OUTSTANDING** will decrease. The disjunction of these four cases (when **TOTAL-OUTSTANDING** is non-zero) is equivalent to **TRUE**. So, we may deduce:

Theorem: Total-Outstanding-Decreases-Leads-To

```
(IMPLIES (AND (TREEP TREE)
              (INITIAL-CONDITION
               '(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS
                                          TREE))
                                 STATE)
                     (NOT-STARTED (QUOTE ,(NODES TREE))
                                   STATE))
               (TREE-PRG TREE)))
         (LEADS-TO '(EQUAL (TOTAL-OUTSTANDING
                             (QUOTE ,(NODES TREE))
                             (QUOTE ,TREE)
                             STATE)
                            (QUOTE ,(ADD1 COUNT)))
                   '(LESSP (TOTAL-OUTSTANDING
                             (QUOTE ,(NODES TREE))
                             (QUOTE ,TREE)
                             STATE)
                            (QUOTE ,(ADD1 COUNT)))
                    (TREE-PRG TREE)))
```

By applying the theorem **Leads-To-Transitive** inductively on **COUNT**, it is possible to prove that **COUNT** decreases to zero. Furthermore, by application of the theorem **Leads-To-Strengthen-Left** (section 3.2.1, page 25), it is possible to substitute **(TRUE)** for the beginning condition, since every state certainly has some (numeric) **TOTAL-OUTSTANDING** measure. This is another example of the use of **Leads-To-Strengthen-Left** in infinite disjunction (section 3.2.1, page 25). The new theorem is:

Theorem: Termination

```
(IMPLIES (AND (TREEP TREE)
              (INITIAL-CONDITION
               '(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS
                                          TREE))
                                 STATE)
                     (NOT-STARTED (QUOTE ,(NODES TREE))
                                   STATE))
               (TREE-PRG TREE)))
         (LEADS-TO '(TRUE)
                   '(EQUAL (TOTAL-OUTSTANDING
                             (QUOTE ,(NODES TREE))
                             (QUOTE ,TREE)
                             STATE)
                            0)
                   (TREE-PRG TREE)))
```

Appealing to the theorems **Leads-To-Weaken-Right**, and **Inv-Implies-Augmented-Correctness-Condition**, we may now deduce the desired correctness theorem:

Theorem: Correctness-Condition

```
(IMPLIES (AND (TREEP TREE)
              (INITIAL-CONDITION
               '(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS TREE))
                                STATE)
                     (NOT-STARTED (QUOTE ,(NODES TREE))
                                  STATE))
               (TREE-PRG TREE)))
         (LEADS-TO '(TRUE)
                   '(CORRECT (QUOTE ,TREE) STATE)
                   (TREE-PRG TREE)))
```

5.5 Conclusion

This summarizes the correctness proof. The case analysis, in the proofs of both the complicated invariant and the decreasing measure, was tedious, because each different case had to be teased apart individually: the user had to point out to the prover, for example, that the individual statement being analyzed was affecting a region of the tree that was not being considered, so certainly the condition would be preserved.

This chapter is based on [30].

All the theorems in this chapter were verified on the extended Boyer-Moore prover with many extra lemmas. Taking the underlying proof system and the naturals library for granted, this proof required 72 definitions, 240 lemmas, and 6155 lines of pretty printed type. Again, many of the lemmas were broken down manually using the Kaufmann Proof Checker.

The entire proof script is presented in Appendix D.

Chapter 6

DINING PHILOSOPHERS

In this chapter, we prove that the classically incorrect solution to the dining philosopher's problem is indeed correct under the assumptions of strong fairness and deadlock freedom. The solution has **N** philosophers in a ring, with a shared fork between each philosopher; a hungry philosopher picks up a single free fork, becomes eating when it has both forks, and subsequently simultaneously becomes thinking and releases both forks. The important observation is that both strong fairness (a hungry philosopher will eventually be able to pick up a fork that becomes free infinitely often) and deadlock freedom (never do all philosophers own their left (right) forks simultaneously) are necessary in the proof, so this example demonstrates both proof rules.

6.1 The Transitions

We first present the statements for each philosopher:
Definition:

```
(THINKING-TO OLD NEW INDEX)
   =
(IF (THINKING OLD INDEX)
    (AND (OR (THINKING NEW INDEX)
             (HUNGRY NEW INDEX))
         (CHANGED OLD NEW (LIST (CONS 'S INDEX))))
  (CHANGED OLD NEW NIL))
```

This function represents the generic transition between thinking and hungry states for a philosopher with index **INDEX**. It states that a philosopher may take a transition between a thinking state and either another thinking state, or a hungry state. The function **CHANGED** states that only the variable **(CONS 'S INDEX)** representing the state of philosopher **INDEX** may change value during this transition. Notice that this transition is always enabled: if it is executed when the philosopher is not thinking, then no values change.

The next function specifies the transition where a philosopher picks up its free left fork:
Definition:

```
(HUNGRY-LEFT OLD NEW INDEX)
   =
(AND (HUNGRY OLD INDEX)
     (FREE OLD INDEX)
     (OWNS-LEFT NEW INDEX)
     (CHANGED OLD NEW (LIST (CONS 'F INDEX))))
```

This statement states that if, in the old state, the philosopher is hungry and its left fork is free, then in the new state it owns its left fork. If the philosopher is neither hungry nor is its left fork free, the statement is disabled. Again, all variables but the one capturing the status of the interesting fork remain unchanged.

The analogous function for picking up free right forks is:

Definition:

```
(HUNGRY-RIGHT OLD NEW INDEX N)
   =
(AND (HUNGRY OLD INDEX)
     (FREE OLD (ADD1-MOD N INDEX))
     (OWNS-RIGHT NEW INDEX N)
     (CHANGED OLD NEW (LIST (CONS 'F (ADD1-MOD N INDEX)))))
```

The important observation in this statement is that forks are indexed in the following way: the **`N`** philosophers have indices **`[0, ... , N-1]`** and a philosopher's left fork shares its index. A right fork, consequently, has the index of the philosopher's right neighbor: **`(ADD1-MOD N INDEX)`**.

The next statement represents the transition from hungry and owning both forks, to eating. It is always enabled:

Definition:

```
(HUNGRY-BOTH OLD NEW INDEX N)
   =
(IF (AND (HUNGRY OLD INDEX)
         (OWNS-LEFT OLD INDEX)
         (OWNS-RIGHT OLD INDEX N))
    (AND (EATING NEW INDEX)
         (CHANGED OLD NEW (LIST (CONS 'S INDEX))))
  (CHANGED OLD NEW NIL))
```

The final statement represents the transition between eating and thinking, with the simultaneous release of both forks. This statement is also always enabled:

Definition:

```
(EATING-TO OLD NEW INDEX N)
   =
(IF (EATING OLD INDEX)
    (AND (THINKING NEW INDEX)
         (FREE NEW INDEX)
         (FREE NEW (ADD1-MOD N INDEX))
         (CHANGED OLD NEW (LIST (CONS 'S INDEX)
                                (CONS 'F INDEX)
                                (CONS 'F (ADD1-MOD
                                          N INDEX)))))
  (CHANGED OLD NEW NIL))
```

Each philosopher in the ring is specified by five statements, captured by the following function:

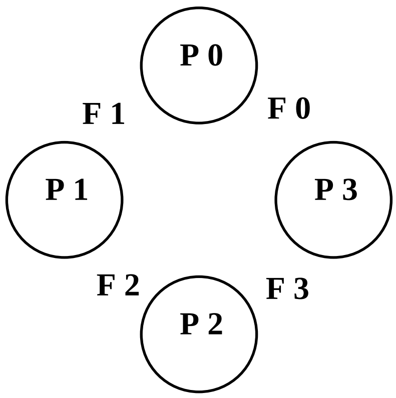
Definition:

```
(PHIL INDEX N)
   =
(LIST (LIST 'THINKING-TO INDEX)
      (LIST 'HUNGRY-LEFT INDEX)
      (LIST 'HUNGRY-RIGHT INDEX N)
      (LIST 'HUNGRY-BOTH INDEX N)
      (LIST 'EATING-TO INDEX N))
```

The first component in each statement is a function name; the remaining components are arguments to that function (supplementing the implicit arguments of the old and new states).

The program for the entire ring of philosophers is the concatenation of instances of **`(PHIL INDEX N)`** for values of **`INDEX`** from **`[0, ... , N-1]`**. This is represented by the term **`(PHIL-PRG N)`**.

For example, the a ring of four philosopher may be represented by the following picture, where **`P I`** is the **`I`**'th philosopher, and **`F I`** is his left fork:



The four process dining philosophers program is **(PHIL-PRG 4)** which is:

```
'((THINKING-TO 3)
  (HUNGRY-LEFT 3)
  (HUNGRY-RIGHT 3 4)
  (HUNGRY-BOTH 3 4)
  (EATING-TO 3 4)
  (THINKING-TO 2)
  (HUNGRY-LEFT 2)
  (HUNGRY-RIGHT 2 4)
  (HUNGRY-BOTH 2 4)
  (EATING-TO 2 4)
  (THINKING-TO 1)
  (HUNGRY-LEFT 1)
  (HUNGRY-RIGHT 1 4)
  (HUNGRY-BOTH 1 4)
  (EATING-TO 1 4)
  (THINKING-TO 0)
  (HUNGRY-LEFT 0)
  (HUNGRY-RIGHT 0 4)
  (HUNGRY-BOTH 0 4)
  (EATING-TO 0 4))
```

6.2 The Correctness Specification

The correctness specification will be a liveness property stating that every hungry philosopher eventually eats. This is captured by the following theorem:

Theorem: Correctness

```
(IMPLIES (AND (LESSP 1 N)
              (NUMBERP INDEX)
              (LESSP INDEX N)
              (INITIAL-CONDITION
               `(AND (PROPER-PHILS STATE (QUOTE ,N))
                      (PROPER-FORKS STATE (QUOTE ,N)))
               (PHIL-PRG N))
              (STRONGLY-FAIR (PHIL-PRG N))
              (DEADLOCK-FREE (PHIL-PRG N)))
         (LEADS-TO `(HUNGRY STATE (QUOTE ,INDEX))
                   `(EATING STATE (QUOTE ,INDEX))
                   (PHIL-PRG N)))
```

The conclusion of this theorem is a **LEADS-TO** statement, where the beginning predicate states that the

INDEX'ed philosopher is hungry, and the ending predicate states that that same philosopher is eating. The hypotheses indicate that we are assuming both strong fairness and deadlock freedom. Also, there is more than one philosopher in the ring, and **INDEX** is some number less than the size of the ring. Finally, we assume two conditions about the initial state: **PROPER-PHILS** and **PROPER-FORKS**. These properties are also invariants of the program.

The term **(PROPER-FORKS STATE N)** states that every fork is either free, or is owned by a neighboring philosopher. The term **(PROPER-PHILS STATE N)** states that **(PROPER-PHIL STATE PHIL RIGHT)** holds for every **PHIL** in the range **[0, ..., N-1]**, where **RIGHT** is **(ADD1-MOD N PHIL)**, where **PROPER-PHIL** is defined as follows:

Definition:

```
(PROPER-PHIL STATE PHIL RIGHT)
   =
(AND (IMPLIES (THINKING STATE PHIL)
              (AND (NOT (EQUAL (FORK STATE PHIL) PHIL))
                   (NOT (EQUAL (FORK STATE RIGHT) PHIL))))
     (IMPLIES (EATING STATE PHIL)
              (AND (EQUAL (FORK STATE PHIL) PHIL)
                   (EQUAL (FORK STATE RIGHT) PHIL)))
     (OR (THINKING STATE PHIL)
         (HUNGRY STATE PHIL)
         (EATING STATE PHIL)))
```

This states that a philosopher is either thinking, hungry, or eating. Also, thinking philosophers own no forks, and eating philosophers own both forks.

The two conditions **(PROPER-PHILS STATE N)** and **(PROPER-FORKS STATE N)** represent legal states and are invariants. This is stated in the following theorem:

Theorem: Phil-Prg-Invariant-1

```
(IMPLIES
 (AND (LESSP 1 N)
      (INITIAL-CONDITION
       '(AND (PROPER-PHILS STATE (QUOTE ,N))
             (PROPER-FORKS STATE (QUOTE ,N)))
       (PHIL-PRG N)))
 (AND (INVARIANT '(PROPER-PHILS STATE (QUOTE ,N))
                 (PHIL-PRG N))
      (INVARIANT '(PROPER-FORKS STATE (QUOTE ,N))
                 (PHIL-PRG N))
      (INVARIANT '(AND (PROPER-PHILS STATE
                                     (QUOTE ,N))
                       (PROPER-FORKS STATE
                                     (QUOTE ,N)))
                 (PHIL-PRG N))))
```

(The final conjunct in the conclusion is redundant but convenient to have around.) This theorem states that if the initial state is legal, then both **PROPER-PHILS** and **PROPER-FORKS** are invariant.

6.3 The Correctness Proof

The invariant properties are proved by demonstrating that every statement preserves the invariant. The liveness property is a more interesting proof and is the focus of this section. To prove that a hungry philosopher eventually eats, we must prove that:

- A hungry philosopher eventually picks up its left fork.
- A hungry philosopher eventually picks up its right fork.
- A hungry philosopher which owns both forks eventually eats.

The last theorem is simple and is proved by appealing to the weak fairness proof rule. (Hungry and owns both forks is stable until eating, and one statement transforms hungry and owns both forks to eating.) The theorem is:

Theorem: Owns-Both-Leads-To-Eating

```
(IMPLIES (AND (LESSP 1 N)
              (LESSP INDEX N)
              (NUMBERP INDEX)
              (INITIAL-CONDITION
               '(AND (PROPER-PHILS STATE (QUOTE ,N))
                     (PROPER-FORKS STATE (QUOTE ,N)))
               (PHIL-PRG N)))
         (LEADS-TO '(AND (OWNS-LEFT STATE (QUOTE ,INDEX))
                         (OWNS-RIGHT STATE (QUOTE ,INDEX)
                                     (QUOTE ,N)))
                   '(EATING STATE (QUOTE ,INDEX))
                   (PHIL-PRG N)))
```

The remaining theorems depend upon forks becoming free infinitely often. A necessary intermediate theorem states that an eating process eventually frees both of its forks. This theorem is also proved by appealing to the weak fairness proof rule, and is stated in the following way:

Theorem: Eating-Leads-To-Free

```
(IMPLIES (AND (LESSP 1 N)
              (LESSP INDEX N)
              (NUMBERP INDEX)
              (INITIAL-CONDITION
               '(AND (PROPER-PHILS STATE (QUOTE ,N))
                     (PROPER-FORKS STATE (QUOTE ,N)))
               (PHIL-PRG N)))
         (LEADS-TO '(EATING STATE (QUOTE ,INDEX))
                   '(AND (FREE STATE (QUOTE ,INDEX))
                         (FREE STATE
                               (QUOTE ,(ADD1-MOD N INDEX))))
                   (PHIL-PRG N)))
```

To prove that forks become free infinitely often, we show that if any fork does not become free infinitely often then a deadlocked condition will eventually exist. For example, if some philosopher's left fork does not become free infinitely often, then all philosophers eventually own their right forks. Later, we take advantage of this result, by the deadlock freedom proof rule: since the conclusion cannot occur, then the hypotheses must be false, and the left fork must become free infinitely often. The theorem is:

Theorem: Eventually-Invariant-Right-Implies-All-Rights

```
(IMPLIES (AND (LESSP 1 N)
              (LESSP J N)
              (NUMBERP J)
              (STRONGLY-FAIR (PHIL-PRG N))
              (INITIAL-CONDITION
               '(AND (PROPER-PHILS STATE (QUOTE ,N))
                     (PROPER-FORKS STATE (QUOTE ,N)))
               (PHIL-PRG N))
              (EVENTUALLY-INVARIANT '(AND (HUNGRY STATE
                                                  (QUOTE ,J))
                                          (OWNS-RIGHT
                                           STATE
                                           (QUOTE ,J)
                                           (QUOTE ,N)))
                                    (PHIL-PRG N)))
         (EVENTUALLY-INVARIANT '(ALL-RIGHTS STATE
                                            (QUOTE ,N))
                               (PHIL-PRG N)))
```

The negation of the **`EVENTUALLY-INVARIANT`** term in the hypotheses implies that the philosopher's right fork becomes free infinitely often, or is owned by the philosopher's right neighbor. More succinctly, this is equivalent to (substituting **`INDEX`** for **`J`**):

```
    (LEADS-TO '(TRUE)
              '(OR (FREE STATE (QUOTE ,(ADD1-MOD N INDEX)))
                   (OWNS-LEFT STATE
                              (QUOTE ,(ADD1-MOD N INDEX))))
              (PHIL-PRG N))
```

The fact that the index is **(ADD1-MOD N INDEX)** is not important, since the range of that term is equivalent to **INDEX**'s domain. Hence, this is equivalent to the **LEADS-TO** property that is needed when appealing to the strong fairness proof rule, when proving that a hungry philosopher will eventually own its left fork.

To prove that every hungry philosopher eventually owns its left fork, we use the deadlock freedom proof rule to prove that a state in which every philosopher owns its right fork cannot occur:

Theorem: Never-All-Rights

```
(IMPLIES (AND (DEADLOCK-FREE (PHIL-PRG N))
              (LESSP 1 N))
         (INVARIANT '(NOT (ALL-RIGHTS STATE (QUOTE ,N)))
                    (PHIL-PRG N)))
```

This is proved by observing that an **ALL-RIGHTS** state is stable and disables (forever) any **HUNGRY-LEFT** statement. Hence, **ALL-RIGHTS** satisfies the criterion of a deadlocked state and is, by deadlock freedom, guaranteed never to occur.

These theorems imply the following, by appealing to the strong fairness proof rule:

Theorem: Hungry-Leads-To-Owns-Left

```
(IMPLIES (AND (LESSP 1 N)
              (NUMBERP INDEX)
              (LESSP INDEX N)
              (INITIAL-CONDITION
               '(AND (PROPER-PHILS STATE (QUOTE ,N))
                     (PROPER-FORKS STATE (QUOTE ,N)))
               (PHIL-PRG N))
              (STRONGLY-FAIR (PHIL-PRG N))
              (DEADLOCK-FREE (PHIL-PRG N)))
         (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
                   '(OWNS-LEFT STATE (QUOTE ,INDEX))
                   (PHIL-PRG N)))
```

A similar argument permits the proof that a hungry philosopher eventually owns its right fork. Combining these results with the facts that a hungry philosopher that owns its left fork persists in that state until it eats and that a hungry philosopher remains hungry until it eats, permits the proof of the correctness theorem:

Theorem: Correctness

```
    (IMPLIES (AND (LESSP 1 N)
                  (NUMBERP INDEX)
                  (LESSP INDEX N)
                  (INITIAL-CONDITION
                   '(AND (PROPER-PHILS STATE (QUOTE ,N))
                         (PROPER-FORKS STATE (QUOTE ,N)))
                   (PHIL-PRG N))
                  (STRONGLY-FAIR (PHIL-PRG N))
                  (DEADLOCK-FREE (PHIL-PRG N)))
             (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
                       '(EATING STATE (QUOTE ,INDEX))
                       (PHIL-PRG N)))
```

6.4 Conclusion

This chapter is based on [31].

All the theorems in this chapter were verified on the extended Boyer-Moore prover, with many extra lemmas. Taking the underlying proof system and the naturals library for granted, the proof required 33 definitions and 78 theorems filling 2234 lines of pretty printed type. To complete this proof, it was necessary to extend the underlying proof system with the notion of eventual stability. The other three algorithms proved in this thesis were verified without changing the proof system.

The entire proof script is presented in Appendix E.

Chapter 7

A DELAY INSENSITIVE FIFO CIRCUIT

The chapter demonstrates that general purpose theorem provers may be used to verify both safety and liveness properties of delay insensitive circuits. Although such mechanized proofs are not automatic, correctness properties may be both non-propositional and describe circuits of arbitrary size. This chapter describes the verification of an n-node first in first out (FIFO) queue.

Many researchers have used the interleaved model of concurrency as a basis for modeling delay insensitive circuits [63, 83, 17]. By restricting the power of program statements and assuming a non-deterministic yet weakly fair scheduling paradigm, interleaving adequately models circuit behavior. Martin's production rules [63] are statements in a non-deterministic program, and are obtained by correct refinements from higher level specifications. Programs in the Synchronized Transitions [83, 84] notation are similar to Unity programs and several have been mechanically verified using LP [24]. However, Synchronized Transitions provides only for the verification of invariance properties.

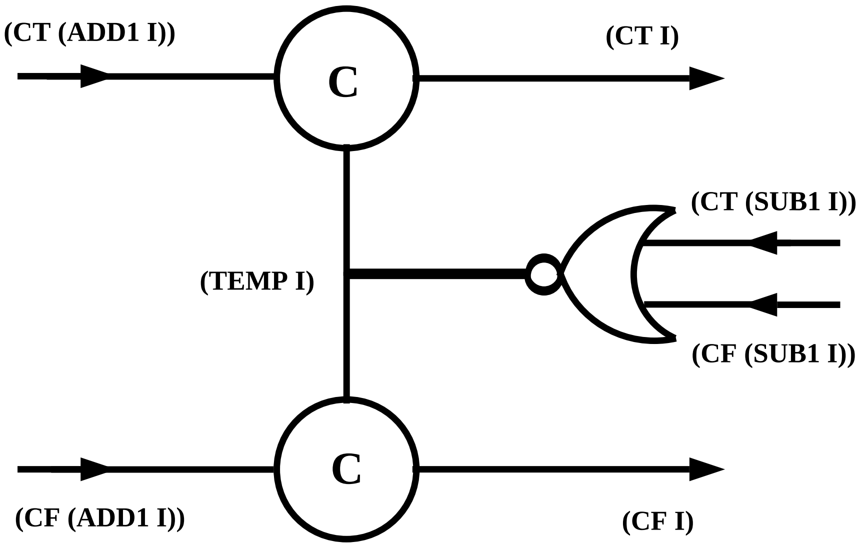
This chapter neither proposes criteria for determining whether a circuit is truly delay insensitive nor argues why such circuits may be analyzed under the interleaved model of concurrency. Rather, given a delay insensitive circuit, it describes how to verify its correctness properties under the interleaved model of concurrency. The example here formalizes an n-node FIFO queue and presents the verification of its safety and the liveness properties. The basic element in this circuit is described in [64].

This chapter is organized in the following way: Section 7.1 defines the FIFO circuit. Section 7.2 presents the correctness theorems, which are proved in section 7.3. Section 7.4 discusses related work and offers concluding remarks.

7.1 The FIFO Circuit

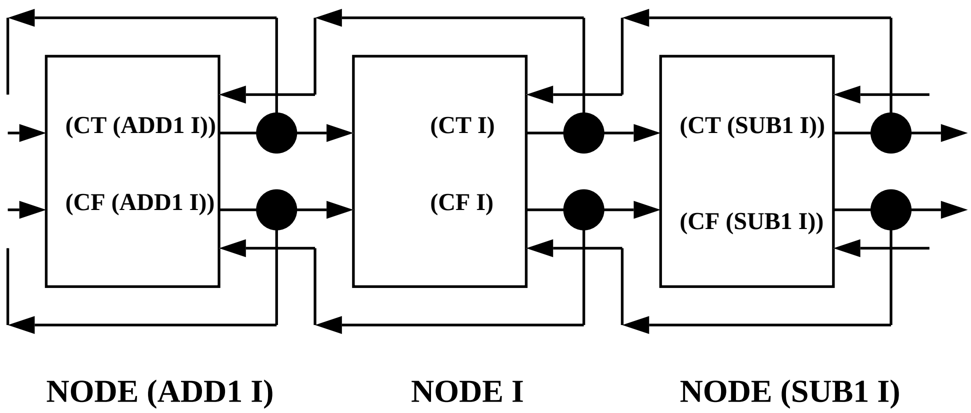
This FIFO circuit is composed of a producer and a consumer which *push* values upon and *pop* values from the internal nodes of the queue. The internal nodes are a sequence of similar nodes, each differing from the other by an index. Each node contains at most one bit; it may be **`TRUE`**, **`FALSE`**, or empty. A node attains its predecessor's value once it determines that its value has been copied to its successor. A node does not become empty simply because its value is copied to its successor. Therefore, in order for this circuit to operate correctly, the producer must push an empty value upon the queue between pushes of non-empty values. Furthermore, a popped value is considered non-empty only if it is non-empty and the previous popped value was empty. Intuitively, a value propagates along the queue leaving a trail of identical values. These copies are cleaned up by the empty value that is pushed upon the queue to delimit the next non-empty value.

An **`N`** node queue has **`N-1`** internal nodes, indexed **`N-1, ..., 1`**. The **`I`**'th internal node in the FIFO circuit has the following components:



The labels on the wires are wire names; notice that each of the output wires from the Muller C-elements [65] actually fork; one branch connects to the input of the successor node's corresponding C-element; the other connects to the predecessor's NOR gate. We take these forks to be isochronic [63] (thereby assuming that the signal propagates simultaneously to the gates at the end of each fork). C-elements are state-holding components, and we treat the value of the output wire of a C-element as its value.

Nodes are connected in the following way: Each node is a box, with four input wires and two output wires. Each output wire forks to be an input to both the successor and predecessor nodes. This picture represents the three consecutive nodes **`(ADD1 I)`**, **`I`**, and **`(SUB1 I)`**:



Each node behaves in the following way: A bit is encoded by double-rail coding. **`TRUE`** is represented by the C-element **`CT`** being **`TRUE`**, and the other C-element **`CF`** being **`FALSE`**. **`FALSE`** is represented by the opposite configuration. If the node is empty, both C-elements are **`FALSE`**; never will both C-elements be **`TRUE`** simultaneously. This is because this circuit requires (and maintains) that if two adjacent nodes are non-empty, they must also represent the same value.

A node copies a new value from its predecessor when its successor differs from its predecessor. For example, assume that the successor is empty, and the predecessor is non-empty. Therefore, the incoming **`TEMP`** becomes **`TRUE`** and permits the other C-elements to become true, if their other inputs are **`TRUE`**.

This node is described by three statements corresponding to the two C-elements and single NOR gate components. The NOR gate is represented by the following function:

Definition:

```
(NOR-GATE OLD NEW A B C)
   =
(AND (IFF (VALUE NEW C)
          (NOT (OR (VALUE OLD A)
                   (VALUE OLD B))))
     (CHANGED OLD NEW (LIST C)))
```

The term **`(VALUE OLD A)`** looks up the value of the variable named **`A`** in state **`OLD`**. **`(CHANGED OLD NEW (LIST C))`** states that only the variable **`C`** may change between states **`OLD`** and **`NEW`**. This function says that the value of **`C`** in state **`NEW`** becomes the *nor* of the values of **`A`** and **`B`** in state **`OLD`**. **`OLD`** and **`NEW`** represent successive states in the execution of the program. The statement for the NOR gate in the **`I`**'th node of the queue must instantiate **`A`**, **`B`**, and **`C`** to be the appropriate wire names. The statement is:

```
    (LIST 'NOR-GATE  (CT (SUB1 I)) (CF (SUB1 I)) (TEMP I))
```

A statement is a list; the first element is the name of the function representing the VLSI component, remaining elements are the names of the input and output wires of that component. Since wire names are indexed, the functions **`CT`**, **`CF`** and **`TEMP`** take arguments.

Similarly, the C-element is described by the following function:

Definition:

```
(C-ELEMENT OLD NEW A B C)
   =
(IF (IFF (VALUE OLD A)
         (VALUE OLD B))
    (AND (IFF (VALUE NEW C) (VALUE OLD A))
         (CHANGED OLD NEW (LIST C)))
  (CHANGED OLD NEW NIL))
```

This function states that **`C`** in state **`NEW`** becomes equal to the inputs, if both inputs **`A`** and **`B`** are equivalent in state **`OLD`**; otherwise, all variables remain unchanged (**`NIL`** is the empty list). This function is used in the following two statements, each representing a single C-element:

```
    (LIST 'C-ELEMENT (CT (ADD1 I)) (TEMP I) (CT I))
    (LIST 'C-ELEMENT (CF (ADD1 I)) (TEMP I) (CF I))
```

A single node of the FIFO circuit is a collection of the two statements representing the two C-elements and the single statement representing the NOR gate. We define the function **`FIFO-NODE`** to collect the three statements in node **`I`**:

Definition:

```
(FIFO-NODE I)
   =
(LIST (LIST 'C-ELEMENT (CT (ADD1 I)) (TEMP I) (CT I))
      (LIST 'C-ELEMENT (CF (ADD1 I)) (TEMP I) (CF I))
      (LIST 'NOR-GATE  (CT (SUB1 I)) (CF (SUB1 I)) (TEMP I)))
```

The fact that the **`TEMP`** wire is truly an isochronic fork is apparent in this formula. That is, **`(TEMP I)`** is the output of the NOR gate, and is an input to two C-elements. Adding a function that copies **`TEMP`** to another wire would add complexity to the verification without changing the overall behavior of the circuit. The output of each C-element is also an isochronic fork.

The internal nodes in our n-node queue will have indices **`(N-1, ..., 1)`**. Nodes **`N`** and **`0`** will be, respectively, producer and consumer nodes. These nodes must obey the four-phase signalling that this queue expects. These nodes also keep track of the *pushed* and *popped* values, in the history variables **`INPUT`** and **`OUTPUT`**, in order to permit the statement of the correctness theorems. The producer node is defined as follows:

Definition:

```
(IN-NODE OLD NEW I)
   =
(IF (IFF (VALUE OLD (TEMP I))
         (EMPTY-NODE OLD I))
    (IF (EMPTY-NODE OLD I)
        (OR (CHANGED OLD NEW NIL)
            (AND (OR (TRUE-NODE NEW I)
                     (FALSE-NODE NEW I))
                 (EQUAL (VALUE NEW 'INPUT)
                        (CONS (TRUE-NODE NEW I)
                              (VALUE OLD 'INPUT)))
                 (CHANGED OLD NEW
                          (LIST (CT I) (CF I) 'INPUT))))
      (OR (CHANGED OLD NEW NIL)
          (AND (EMPTY-NODE NEW I)
               (CHANGED OLD NEW (LIST (CT I) (CF I))))))
  (CHANGED OLD NEW NIL))
```

The term **(EMPTY-NODE OLD I)** tests whether this **I**'th node is empty in state **OLD**. (Neither C-element in node **I** is **TRUE**.) Terms **(TRUE-NODE NEW I)** and **(FALSE-NODE NEW I)** test whether the **I**'th node in state **NEW** contains a **TRUE** or **FALSE** bit, respectively. In our example, the producer node will have index **N**. Its behavior is as follows: If the value of the tail of the queue (node **N**) has already been copied into node **N-1** (as indicated by the value of wire **(TEMP N)**) and the tail of the queue is empty, then a new value *may* be placed upon the tail of the queue. If the new value is **TRUE** or **FALSE**, then the variable **INPUT** is updated (by inserting the new value at its head) to reflect the newly pushed value. If the tail of the queue is not empty, yet has already been copied, then an empty value may be placed upon the tail of the queue. If the tail of the queue has not yet been copied, no change occurs.

The producer node is nondeterministic in several ways: It may push one of three values upon the queue, arbitrarily (as long as it satisfies the protocol). The producer node may also halt, since it never need push any new value. Therefore, the environment may stop.

The consumer node is defined as follows:

Definition:

```
(OUT-NODE OLD NEW)
   =
(AND (IFF (VALUE NEW (CT 0))
          (VALUE OLD (CT 1)))
     (IFF (VALUE NEW (CF 0))
          (VALUE OLD (CF 1)))
     (IF (AND (EMPTY-NODE OLD 0)
              (NOT (EMPTY-NODE NEW 0)))
         (EQUAL (VALUE NEW 'OUTPUT)
                (CONS (TRUE-NODE NEW 0)
                      (VALUE OLD 'OUTPUT)))
       (EQUAL (VALUE NEW 'OUTPUT) (VALUE OLD 'OUTPUT)))
     (CHANGED OLD NEW (LIST (CT 0) (CF 0) 'OUTPUT)))
```

The consumer node's index is **0**. Node **1** is copied into the head of the queue. If the head of the queue is thereby changed from empty to non-empty, then the variable **OUTPUT** representing popped values is updated appropriately (the newly popped value is inserted into its head). Since the schedule of statements is unknown, the internal nodes in the queue cannot depend upon the rate at which values are popped.

The entire queue, consisting of a consumer, the internal nodes, and a producer (with the extra **(TEMP N)** wire), is represented using the following three functions. The first collects the internal nodes:

Definition:

```
(INTERNAL-NODES N)
   =
(IF (ZEROP N)
    NIL
  (APPEND (FIFO-NODE N)
          (INTERNAL-NODES (SUB1 N))))
```

The next function collect the statements describing the external nodes:

Definition:

```
(EXTERNAL-NODES N)
   =
(LIST (LIST 'IN-NODE N)
      (LIST 'OUT-NODE)
      (LIST 'NOR-GATE  (CT (SUB1 N)) (CF (SUB1 N)) (TEMP N)))
```

Finally, the entire circuit is captured by the term **(FIFO-QUEUE N)**:

Definition:

```
(FIFO-QUEUE N)
   =
(APPEND (EXTERNAL-NODES N)
        (INTERNAL-NODES (SUB1 N)))
```

For example, the FIFO queue with 2 internal nodes, is represented by the term **(FIFO-QUEUE 3)** which equals:

```
     '((IN-NODE 3)
       (OUT-NODE)
       (NOR-GATE (CT . 2) (CF . 2) (TEMP . 3))
       (C-ELEMENT (CT . 3) (TEMP . 2) (CT . 2))
       (C-ELEMENT (CF . 3) (TEMP . 2) (CF . 2))
       (NOR-GATE (CT . 1) (CF . 1) (TEMP . 2))
       (C-ELEMENT (CT . 2) (TEMP . 1) (CT . 1))
       (C-ELEMENT (CF . 2) (TEMP . 1) (CF . 1))
       (NOR-GATE (CT . 0) (CF . 0) (TEMP . 1)))
```

In the correctness specifications, we use the term **(FIFO-QUEUE N)** denoting a FIFO queue of length **N**. As with all variables, **N** is universally quantified, so the theorems are true for queues of any length. (A hypothesis in these theorems requires that **N** exceed **1**, implying the existence of at least one internal node.)

7.2 The Correctness Specifications

The important correctness properties, that pushed values are not lost, and that pushed values are eventually popped, both depend upon an invariant which characterizes legal states. Recall that the correct operation of the circuit depends upon adjacent non-empty nodes being equivalent. In addition, if a node differs from its successor, then its incoming **TEMP** wire must be up-to-date. These requirements are formalized in the following way:

Definition:

```
(PROPER-NODE STATE I)
   =
(AND (IMPLIES (AND (NOT (EMPTY-NODE STATE I))
                   (EMPTY-NODE STATE (SUB1 I)))
              (VALUE STATE (TEMP I)))
     (IMPLIES (AND (EMPTY-NODE STATE I)
                   (NOT (EMPTY-NODE STATE (SUB1 I))))
              (NOT (VALUE STATE (TEMP I))))
     (OR (TRUE-NODE STATE I)
         (FALSE-NODE STATE I)
         (EMPTY-NODE STATE I))
     (IMPLIES (NOT (EMPTY-NODE STATE I))
              (OR (EMPTY-NODE STATE (SUB1 I))
                  (IF (TRUE-NODE STATE I)
                      (TRUE-NODE STATE (SUB1 I))
                    (FALSE-NODE STATE (SUB1 I))))))
```

The term **(PROPER-NODES STATE N)** checks whether nodes **(N, ..., 1)** are proper. The invariance property is stated as follows:

Theorem: Proper-Nodes-Invariant

```
(IMPLIES (AND (LESSP 1 N)
              (INITIAL-CONDITION '(PROPER-NODES STATE
                                                (QUOTE ,N))
                                 (FIFO-QUEUE N)))
         (INVARIANT '(PROPER-NODES STATE (QUOTE ,N))
                    (FIFO-QUEUE N)))
```

This theorem states that if the initial state is legal, then all subsequent states are legal. The legal state predicate is encoded as a backquoted [85] term, in the following way: The first element of the term is the function symbol **PROPER-NODES**, so **PROPER-NODES** is the function that is invariant. The second element is **STATE**, which is a dummy literal: upon evaluating the backquoted term in the context of some state in the execution, **STATE** is bound to that state. The third element is **(QUOTE ,N)** which is a shorthand for introducing a variable into the formula. That is, the **N** in the hypothesis is the same **N** that is in the conclusion, and is the same universally quantified **N** specifying the size of the queue that we are describing via the function **(FIFO-QUEUE N)**.

The next invariant states that values are consumed in the order in which they are produced. To specify this, we define the term **(QUEUE-VALUES STATE N)** which returns a list of the values in the queue:

Definition:

```
(QUEUE-VALUES STATE N)
   =
(IF (ZEROP N)
    NIL
  (IF (AND (NOT (EMPTY-NODE STATE N))
           (EMPTY-NODE STATE (SUB1 N)))
      (CONS (TRUE-NODE STATE N)
            (QUEUE-VALUES STATE (SUB1 N)))
    (QUEUE-VALUES STATE (SUB1 N))))
```

Specifically, a node only *counts* if it is non-empty and its successor is empty. The invariant depends upon the queue being in a legal configuration and is specified in the following way:

Theorem: Queue-Values-Invariant

```
(IMPLIES (AND (INITIAL-CONDITION
               '(AND (PROPER-NODES STATE (QUOTE ,N))
                     (EQUAL (VALUE STATE (QUOTE INPUT))
                            (APPEND (QUEUE-VALUES STATE
                                                  (QUOTE ,N))
                                    (VALUE STATE
                                           (QUOTE OUTPUT)))))
               (FIFO-QUEUE N))
              (LESSP 1 N))
         (INVARIANT '(EQUAL (VALUE STATE (QUOTE INPUT))
                            (APPEND (QUEUE-VALUES STATE
                                                  (QUOTE ,N))
                                    (VALUE STATE
                                           (QUOTE OUTPUT))))
                    (FIFO-QUEUE N)))
```

This invariant states that the produced values always equal the concatenation of the values in the queue and the consumed values. Interestingly, this invariant can be satisfied by an incorrect program: we must also prove that the variables **INPUT** and **OUTPUT** only grow. These statements have been proved as **UNLESS** properties stating that for all statements in the program, the variables **INPUT** and **OUTPUT** either remain unchanged, or become extended by the values **TRUE** or **FALSE**.

Theorem: Input-Only-Adds-Boolean

```
(IMPLIES (LESSP 1 N)
         (UNLESS '(EQUAL (VALUE STATE (QUOTE INPUT))
                         (QUOTE ,K))
                 '(OR (EQUAL (VALUE STATE (QUOTE INPUT))
                             (CONS (TRUE) (QUOTE ,K)))
                      (EQUAL (VALUE STATE (QUOTE INPUT))
                             (CONS (FALSE) (QUOTE ,K))))
                 (FIFO-QUEUE N)))
```

Theorem: Output-Only-Adds-Boolean

```
(IMPLIES (LESSP 1 N)
         (UNLESS '(EQUAL (VALUE STATE (QUOTE OUTPUT))
                         (QUOTE ,K))
                 '(OR (EQUAL (VALUE STATE (QUOTE OUTPUT))
                             (CONS (TRUE) (QUOTE ,K)))
                      (EQUAL (VALUE STATE (QUOTE OUTPUT))
                             (CONS (FALSE) (QUOTE ,K))))
                 (FIFO-QUEUE N)))
```

The liveness condition requires that values be passed through the queue. Without tagging queue values, this must be stated in the following way: if the queue is non-empty, then eventually the number of consumed values increases. This is expressed in the following **LEADS-TO** property:

Theorem: Output-Grows

```
(IMPLIES (AND (INITIAL-CONDITION '(PROPER-NODES STATE N)
                                  (FIFO-QUEUE N))
              (LESSP 1 N))
         (LEADS-TO '(AND (LISTP (QUEUE-VALUES STATE N))
                         (EQUAL (LENGTH (VALUE
                                         STATE
                                         (QUOTE OUTPUT)))
                                (QUOTE ,K)))
                   '(LESSP (QUOTE ,K)
                           (LENGTH (VALUE STATE
                                          (QUOTE OUTPUT))))
                   (FIFO-QUEUE N)))
```

7.3 The Correctness Proof

The proof of the invariance theorems proceeded by case analysis on the various statements in the program. Since the functions specifying both legal states and queue values are defined recursively, the proofs of these theorems were inductive and required that generalizations of the invariance theorems be proved first. It is unfortunate that the legal state invariant cannot be decomposed: although, the invariant is really three conjuncts, the stability of each depends upon all three.

The liveness property is a more interesting proof and is the focus of this section. We wish to prove that non-empty values on the queue are eventually popped off the queue; this was formalized by stating that the length of the history variable **OUTPUT** recording popped values eventually increases. We prove this by demonstrating a decreasing measure: non-empty values move forward in the queue; when one reaches node **1**, it is popped and the length of **OUTPUT** grows. We prove the decreasing measure in a restricted sense: if a node in the queue is non-empty and the entire subqueue ahead of it is empty, then that non-empty value moves forward. It is obvious that any non-empty queue also has a most forward element, so it is sufficient to prove this theorem. Furthermore, it is simpler to prove this theorem than to consider the interactions of unknown elements in the queue. The theorem is stated in the following way:

Theorem: Full-Rest-Empty-Moves-Forward

```
(IMPLIES (AND (INITIAL-CONDITION ‘(PROPER-NODES STATE
                                                (QUOTE ,N))
                                 (FIFO-QUEUE N))
              (LESSP 1 N)
              (LESSP I N)
              (NOT (ZEROP I)))
         (LEADS-TO ‘(AND (NOT (EMPTY-NODE
                                STATE (QUOTE ,(ADD1 I))))
                         (AND (EMPTY-NODE STATE (QUOTE ,I))
                              (NOT (LISTP (QUEUE-VALUES
                                             STATE
                                             (QUOTE ,I))))))
                   ‘(AND (NOT (EMPTY-NODE STATE (QUOTE ,I)))
                         (AND (EMPTY-NODE STATE
                                          (QUOTE ,(SUB1 I)))
                              (NOT (LISTP
                                     (QUEUE-VALUES
                                      STATE
                                      (QUOTE ,(SUB1 I)))))))
                   (FIFO-QUEUE N)))
```

This theorem can be applied inductively, since it says that if the **(ADD1 I)**’th queue value is the topmost non-empty value, eventually the **I**’th element will be the topmost non-empty value. This is applied inductively until the node **1** is the topmost non-empty value. The following theorem states that **OUTPUT** then grows:

Theorem: Output-Grows-Immediately

```
(IMPLIES (AND (INITIAL-CONDITION ‘(PROPER-NODES STATE
                                                (QUOTE ,N))
                                 (FIFO-QUEUE N))
              (LESSP 1 N))
         (LEADS-TO ‘(AND (NOT (EMPTY-NODE STATE 1))
                         (AND (EMPTY-NODE STATE 0)
                              (EQUAL (LENGTH (VALUE STATE
                                                    ’OUTPUT))
                                     (QUOTE ,K))))
                   ‘(LESSP (QUOTE ,K)
                           (LENGTH (VALUE STATE ’OUTPUT)))
                   (FIFO-QUEUE N)))
```

The theorem **Output-Only-Adds-Boolean** implies that the length of **OUTPUT** only grows. This fact, along with the previously stated two theorems, permits the proof of the liveness property, **Output-Grows**:

Theorem: Output-Grows

```
(IMPLIES (AND (INITIAL-CONDITION '(PROPER-NODES STATE N)
                                  (FIFO-QUEUE N))
              (LESSP 1 N))
         (LEADS-TO '(AND (LISTP (QUEUE-VALUES STATE N))
                         (EQUAL (LENGTH (VALUE
                                         STATE
                                         (QUOTE OUTPUT)))
                                (QUOTE ,K)))
                   '(LESSP (QUOTE ,K)
                           (LENGTH (VALUE STATE
                                          (QUOTE OUTPUT))))
                   (FIFO-QUEUE N)))
```

7.4 Conclusion

This chapter demonstrates how techniques for reasoning about concurrent programs may be applied to delay insensitive circuits. In this case, a proof system mechanizing Unity has been used to specify and verify both safety and liveness properties for an n-node FIFO circuit. This specification of the queue formalizes the assumptions about its environment. Mechanized Unity permits the mechanically verified proof of circuits of arbitrary size.

This work is similar to Synchronized Transitions [83], especially the later work [84] which was mechanized on LP [24]. Synchronized Transitions uses a syntax similar to Unity for specifying hardware. It can only be used, however, to prove invariance properties. The invariance property of a high level specification of a FIFO circuit was mechanically verified in [84]. Synchronized Transitions does provide a nice composition mechanism for hierarchical circuit design.

The Boyer-Moore prover has been used to verify parameterized clocked hardware as well. [45] verified a microprocessor; its ALU was verified for arbitrary register sizes. [25] has verified several combinational designs also. The circuits discussed there are synchronous and depend upon a clock. Ideally, one would like to merge verification techniques, in order to be able to reason about asynchronous collections of synchronous hardware.

Other research has produced promising techniques for fully automatic verification of certain safety [22] and liveness properties [15] using trace theory and model checking. These systems check whether a finite state machine satisfies a formula by, essentially, completely simulating the machine. If the machine does not satisfy the formula, the system can return an offending trace; this facility is useful for debugging. Such systems may be more useful than semi-automatic techniques for verifying fixed size circuit components, since invariants specifying legal states become very complicated. However, these systems cannot reason about arbitrary sized components or about non-propositional correctness properties. Also, one must still determine the circuit's suitable initial states, which, in the general case, is similar to determining invariants. A useful system (which does not yet exist) might combine automatic techniques for verifying fixed sized circuit components with semi-automatic techniques for combining these components.

There are two assumptions underlying this work. The first is that the behavior of delay insensitive circuits is accurately modeled by the interleaved model of concurrency. This assumption permits one to ignore isochronic forks and use the same wire in several inputs. The second is that the circuit being verified is truly delay insensitive. Several criteria have been proposed to test delay insensitivity: Martin checks whether his production rules map to VLSI components and the rules' preconditions are mutually exclusive. Straunstrup et. al, propose the two conditions of consumed values and correspondence, while Chandy and Misra suggest stability of preconditions. Since these conditions sometimes conflict, characterizing delay insensitive circuits remains an incompletely answered question. In this chapter, an arbitrary sized circuit known to be delay insensitive was verified under the interleaved model of concurrency.

This chapter is based on [32, 33].

All the theorems in this section were verified on the extended Boyer-Moore prover, with many extra lemmas. Taking the underlying proof system and the naturals libraries for granted, this proof required 19 definitions and 49 lemmas, comprising 3177 lines of pretty printed text. Many of these lemmas, however, were manually broken down into multiple goals using the Kaufmann Proof Checker.

The entire event list is presented in Appendix F.

Chapter 8

CONCLUSION

This thesis presents a mechanically verified proof system based on the Unity [17] logic, for reasoning about concurrent programs on the Boyer-Moore prover [11]. Mechanized Unity also provides a theory for reasoning about the Unity logic, by providing an operational semantics as a foundation for Unity's proof rules.

The proof system was demonstrated by the mechanically verified proofs of four concurrent programs. These proofs exercised reasoning under a variety of fairness notions, demonstrated the safety notion of deadlock-freedom, and verified a delay insensitive circuit. All the algorithms verified here were parameterized; the correctness results are valid for arbitrary sized systems.

Specifications and algorithms in this proof system are presented in an extension of the Boyer-Moore logic that permits fully quantified non-recursive definitions. No front end is provided to translate formulas from some domain specific notation into the mechanized logic. This choice was deliberate, because proofs require so much interaction from the user that adding another level of indirection simply complicates matters. Formulas are still clear, since the quantifier extension permits the definition of meaningful specification predicates. Although proofs are long, specifications are typically short and readable.

Proofs in this system are expensive to construct. This may be because the proof system is clumsy, or because mechanized proofs, or proofs in general, are difficult to develop. It is reasonable to ask whether this proof system is harder to use than comparable systems. The only other proof system for concurrent programs that has been used to mechanically verify programs from first principles (i.e., completely) is [77, 78], which encodes a subset of Manna and Pnueli's temporal logic on the Boyer-Moore prover. That system possesses a front end, which, the author claims, permits the automatic proof of many complicated theorems. However, these automatic proofs are supported by specially constructed libraries. Furthermore, the comparison is somewhat uneven. Much of the complexity present here derives from the parameterized programs. To analyze these statements effectively, it appears to be necessary to divide the cases manually. [77] handles only fixed size programs. A valid comparison would be to verify some fixed size program on this system, taking advantage of similarly constructed libraries.

8.1 Lessons Learned

This project provided insight into both Unity and the Boyer-Moore logic. By defining an operational semantics of concurrency, the liveness specification predicate, **`LEADS-TO`** can be simply characterized. This predicate may be proved to satisfy Unity's defining axioms for **`LEADS-TO`**. This process demonstrates the soundness of those axioms. Many other theorems of Unity's **`LEADS-TO`** were proved about this **`LEADS-TO`** as well.

Since proofs in this proof system use only Unity's proof rules, and do not depend upon the operational semantics of concurrency, it would not be simpler to use a mechanization of Unity on a temporal logic theorem prover (assuming that the provers are otherwise equally powerful). Although certain proof rules do contain references to the arbitrary fair computation **S**, such references are only a logical device serving two purposes:

- They provide a particular state with which to compare predicates. In a first order logic, in order to determine whether one predicate is stronger than another, one must check the effect of the predicates on all relevant states.
- References to the fair computation **S** instead of some arbitrary state indicates that the state being considered is reachable. This provides additional information, since reachable states satisfy computation invariants.

These proof rules may not be as pretty as their counterparts in temporal logic, but they are no more complicated. Furthermore, the particular composition of Skolem functions, that identify which point in the computation must be considered, is generated automatically by the instantiation of other parts of the proof rule which would have to be instantiated in temporal logic too.

The ability to refer to arbitrary reachable states also resolves a point of possible confusion in Unity. In Unity, both **UNLESS** and **ENSURES** also refer to reachable states, and the substitution axiom may be used to simplify such terms. However, the use of the substitution axiom there may prevent the program composition of **UNLESS** and **ENSURES**. In this mechanization, **UNLESS** and **ENSURES** refer to arbitrary states, and the substitution axiom does not simplify them. Therefore, they always compose.

This project also prompted two extensions to the Boyer-Moore logic. The first, functional instantiation [13], permits the definition of partially characterized functions. This extension was necessary in order to reason about arbitrary computations. The second, Defn-Sk [53], permits full quantification in the body of non-recursive function definitions. Without this extension, predicates like **LEADS-TO** could not be defined, and specifications would have been more complicated, although proofs would have been very similar.

8.2 Mechanized Proofs

These proof were carried out using the Boyer-Moore prover extended with the Kaufmann Proof Checker [52]. Neither the basic speed of these programs nor of the supporting hardware was a significant factor in these proofs, although the fact that the programs are well coded and were run on relatively fast machines certainly made life more pleasant.[27] The bulk of one's proof time is spent developing the proof. It would have been much more difficult to complete these proofs on the Boyer-Moore prover alone, since most of Unity's proof rules have free variables in the hypotheses that cannot, in general, be instantiated automatically. These proof rules are similar to transitivity or other theorems with existentially quantified variables in the hypotheses. Although the Boyer-Moore prover does provide a hint mechanism to instantiate these variables, the proof checker provides more flexibility.

The proof checker both encourages the user to take larger proof steps and to take many small steps. For example, a theorem may be proved that requires backchaining over several lemmas which could never have been used together automatically. During the course of this manual proof, the user may make many small simplifications. This is one reason, for example, that the average lemma event in Appendix F is 62 lines long, although the average length of the actual formulas in those lemmas is only 12 pretty printed lines. The difference is the sequence of proof checker commands.

The two longest lemmas in that proof, both have 17 lines in their actual statement, while their instruction lists are 388 and 629 lines long. The long length of these two lemmas was due to repeated uses of the proof checker's generalize command which permits the replacement of a term by a new variable. Terms referring to Skolem functions are often extremely complicated since they include references to all the universally quantified variables they depend upon. Moreover, some of these variables are instantiated to literals representing correctness terms and are very long. In these proofs, these Skolem variables could typically be generalized away since the Skolem functions identified some arbitrary point in the

[27] Even so, some proofs took a day or more to prove again.

computation, but what is important is that the computation is being referred to and not some arbitrary state. The particular composition of Skolem functions is irrelevant. Generalizing the long terms away greatly speeds up the proof, but adds lines to the proof script.

Another advantage of the proof checker is the ability to do full simplification on backchained hypotheses. The Boyer-Moore prover does not attempt full simplification to relieve these hypotheses, for reasons of efficiency. Using the proof checker, rewrite rules may be applied manually, and the subsequent theorems submitted to the prover as first class lemmas, for automatic proof. This technique may also be used to control case analysis.

8.3 Future Work

Perhaps the most promising approach for simplifying proofs and reducing the cost of mechanical verification would be an effective combination of automatic and semi-automatic proof systems. In many cases, the analysis of program statements is tedious and difficult, but these statements are really only finite state machines. If a proof could be decomposed into its finite and non-finite components, the cost of verification may decrease dramatically. To do this effectively, it is necessary to analyze the domains being worked with.

For example, none of the programs considered here is finite state, since the programs are parameterized. However, components of the programs may still be finite state. Consider the delay-insensitive FIFO circuit presented in chapter 7. Each node in the circuit is a very simple circuit that modifies two output wires, in response to changes on four input wires. In fact, each node is isolated from every other node except for its left and right neighbors. Certain properties of three adjacent nodes ought to be automatically decidable. Of course, the process of decomposition may itself be difficult, but even semi-automatic proofs must be decomposed into manageable components. This approach would require induction over the size of the program and the proof of basic properties for each component.

The dining philosophers program (chapter 6) may be decomposed in a similar way. The decomposition of the simplest program, the mutual exclusion problem (chapter 4), is not finite state, since even a single process may cycle an arbitrary number of times after moving from waiting to critical. The decomposition of the minimum tree algorithm (chapter 5) is more complicated, since it has two levels of induction in the decomposition: one to descend the tree, and the other to span children of each node.

Doing such decompositions in Mechanized Unity introduces a subtle problem. It appears to be impossible, in general, to relate **LEADS-TO** properties of one program to another program. For example, one would like to prove the following non-theorem:

```
(IMPLIES (AND (PERMUTATIONP PRG-1 PRG-2)
              (LEADS-TO P Q PRG-1))
         (LEADS-TO P Q PRG-2))
```

where **PERMUTATIONP** checks whether its two arguments are permutations of one another. This formula is a non-theorem because **LEADS-TO** is defined with respect to some fair computation. Although this computation is arbitrary, it is still some particular computation which may possess peculiar **LEADS-TO** properties due to the scheduler. These scheduler dependent properties need not hold for **LEADS-TO** of both **PRG-1** and **PRG-2**.

Notice, however, that these peculiar **LEADS-TO** properties are not provable, neither in Mechanized Unity nor in Unity. However, in Unity, since **LEADS-TO** is defined to be only provable **LEADS-TO** properties, the Unity version of this candidate non-theorem is indeed valid for Unity programs. This observation validates Unity's choice of characterizing **LEADS-TO** as the strongest predicate satisfying the three characterizing axioms, since this ties **LEADS-TO** directly to provability.

8.4 Final Notes

The development of this proof system contributed to the enhancement of the Boyer-Moore logic and illuminated certain design decisions in Unity. This proof system provides a foundation for future research in the mechanical verification of concurrent programs.

Appendix A
Backquote

This appendix presents Boyer and Moore’s Common Lisp code defining Nqthm’s interpretation of backquote, which is consistent with the specification presented in [85].

After loading this code, Nqthm’s backquote is enabled by evaluating the form **(BACKQUOTE-SETTING ’NQTHM)**. The original interpretation is restored by evaluating the form **(BACKQUOTE-SETTING ’ORIG)**.

The setting **(BACKQUOTE-SETTING ’NQTHM)** is used for the proofs of the theorems in the other appendices.

```
(DEFPARAMETER *BACKQUOTE-COUNTER* 0)

(DEFPARAMETER *COMMA* (MAKE-SYMBOL "COMMA"))

(DEFPARAMETER *COMMA-ATSIGN* (MAKE-SYMBOL "COMMA-ATSIGN"))

(DEFUN BACKQUOTE (X)

; SEE THE FILE REMARKS.TEXT FOR SOME HISTORICAL INFORMATION REGARDING
; OUR IMPLEMENTATION OF BACKQUOTE.

  (COND ((SIMPLE-VECTOR-P X)
         (LIST ’APPLY ’(FUNCTION VECTOR)
               (BACKQUOTE (COERCE X ’LIST))))
        ((ATOM X) (LIST ’QUOTE X))
        ((EQ (CAR X) *COMMA*) (CADR X))
        ((EQ (CAR X) *COMMA-ATSIGN*)
         (ER SOFT NIL |‘, |IS| |AN| |ERROR| |.|))
        (T (BACKQUOTE-LST X))))

(DEFUN BACKQUOTE-LST (L)
  (COND ((ATOM L)
         (LIST ’QUOTE L))
        ((EQ (CAR L) *COMMA*)
         (CADR L))
        ((EQ (CAR L) *COMMA-ATSIGN*)
         (ER SOFT NIL |. , |IS| |ILLEGAL| |.|))
        ((AND (CONSP (CAR L))
              (EQ (CAAR L) *COMMA*))
         (LIST ’CONS
               (CADR (CAR L))
               (BACKQUOTE-LST (CDR L))))
        ((AND (CONSP (CAR L))
              (EQ (CAAR L) *COMMA-ATSIGN*))
         (LIST ’APPEND (CADR (CAR L)) (BACKQUOTE-LST (CDR L))))
        (T (LIST ’CONS
                 (BACKQUOTE (CAR L))
                 (BACKQUOTE-LST (CDR L))))))

(DEFUN BACKQUOTE-SETTING (ARG)

; BACKQUOTE-SETTING IS TO BE USED TO SET THE COMMA AND BACKQUOTE
; FUNCTIONS IN *READTABLE*.  IF ARG IS ’ORIG, THEN WE REVERT TO THE
; ORIGINAL SETTING.  IF ARG IS ’NQTHM, THEN WE USE THE NQTHM
; DEFINITIONS.

; WE USE THE NQTHM DEFINITIONS IN PROVE-FILE, AND WE RECOMMEND THAT
; ANY NQTHM USER WHO WANTS TO USE BACKQUOTE IN NQTHM FORMS ESTABLISH
; THE NQTHM SETTINGS IN THE DEFAULT TOP LEVEL READER BY INVOKING
; (BACKQUOTE-SETTINGS ’NQTHM) AT THE TOP LEVEL OF LISP.
```

```
(COND ((NOT (MEMBER ARG '(ORIG NQTHM)))
       (ER SOFT (ARG) |THE| |ARGUMENT| |TO| BACKQUOTE-SETTING |MUST|
           |BE| |THE| |SYMBOL| NQTHM |OR| |THE| |SYMBOL| ORIG |BUT|
           (!PPR ARG NIL) |IS| |NEITHER| |.|)))
(SET-MACRO-CHARACTER
 #\`
 (COND ((EQ ARG 'NQTHM)
        #'(LAMBDA (STREAM CHAR)
            (DECLARE (IGNORE CHAR))
            (LET ((*BACKQUOTE-COUNTER* (1+ *BACKQUOTE-COUNTER*)))
              (BACKQUOTE (READ STREAM T NIL T)))))
       (T (GET-MACRO-CHARACTER #\` (COPY-READTABLE NIL)))))
(SET-MACRO-CHARACTER
 #\,
 (COND ((EQ ARG 'NQTHM)
        #'(LAMBDA (STREAM CHAR)
            (DECLARE (IGNORE CHAR))
            (LET ((*BACKQUOTE-COUNTER* (1- *BACKQUOTE-COUNTER*)))
              (COND ((< *BACKQUOTE-COUNTER* 0)
                     (ER SOFT NIL |ILLEGAL| |COMMA| |ENCOUNTERED|
                         |BY| READ |.|)))
              (CASE (PEEK-CHAR NIL STREAM T NIL T)
                    ((#\ #\.) (READ-CHAR STREAM T NIL T)
                     (LIST *COMMA-ATSIGN* (READ STREAM T NIL T)))
                    (OTHERWISE (LIST *COMMA* (READ STREAM T NIL T)))))))
       (T (GET-MACRO-CHARACTER #\, (COPY-READTABLE NIL))))))
```

Appendix B
Proof System Events

This appendix contains the complete events list supporting the proof system described in chapters 2 and 3.

This event list constructs the proof system system on top of the naturals library [7], on top of the extended Boyer-Moore Prover.

```
(NOTE-LIB "NATURALS")

(PROVEALL "INTERPRETER" '(



;;; SEVERAL USEFUL THEOREMS

(PROVE-LEMMA EQUAL-IFF (REWRITE)
             (IMPLIES (AND (OR (TRUEP A)
                               (FALSEP A))
                           (OR (TRUEP B)
                               (FALSEP B)))
                      (EQUAL (EQUAL A B)
                             (IFF A B))))

(DISABLE EQUAL-IFF)

(DEFN LENGTH (LIST)
  (IF (LISTP LIST)
      (ADD1 (LENGTH (CDR LIST)))
    0))

(DEFN NTH (LIST N)
  (IF (ZEROP N)
      (CAR LIST)
    (NTH (CDR LIST) (SUB1 N))))

(DEFN POSITION (PRG E)
  (IF (LISTP PRG)
    (IF (EQUAL (CAR PRG) E)
        0
      (ADD1 (POSITION (CDR PRG) E)))
    0))

(PROVE-LEMMA NTH-MEMBER (REWRITE)
             (IMPLIES (LESSP N (LENGTH LIST))
                      (MEMBER (NTH LIST N) LIST)))


(PROVE-LEMMA LISTP-NOT-ZERO-LENGTH (REWRITE)
             (EQUAL (EQUAL (LENGTH LIST) 0)
                    (NOT (LISTP LIST))))


(PROVE-LEMMA POSITION-ZERO (REWRITE)
             (IMPLIES (NOT (MEMBER E PRG))
                      (EQUAL (POSITION PRG E)
                             (LENGTH PRG))))

(PROVE-LEMMA POSITION-LESSP (REWRITE)
             (IMPLIES (MEMBER E PRG)
                      (LESSP (POSITION PRG E)
                             (LENGTH PRG))))

(PROVE-LEMMA NTH-POSITION (REWRITE)
```

```
             (EQUAL (NTH PRG (POSITION PRG E))
                    (IF (MEMBER E PRG)
                        E
                      0)))

(PROVE-LEMMA POSITION-NTH (REWRITE)
              (IMPLIES (AND (MEMBER E LIST)
                            (NOT (EQUAL (NTH LIST N) E)))
                       (EQUAL (EQUAL (POSITION LIST E) N) F)))

(PROVE-LEMMA MEMBER-APPEND (REWRITE)
             (EQUAL (MEMBER E (APPEND J K))
                    (OR (MEMBER E J)
                        (MEMBER E K))))


(PROVE-LEMMA APPEND-IS-ASSOCIATIVE (REWRITE)
             (EQUAL (APPEND (APPEND X Y) Z)
                    (APPEND X (APPEND Y Z))))


;;; THE WITNESS FUNCTION FOR THE COMPUTATION


(DEFN MCHOOSE (PRG I)
  (NTH PRG (REMAINDER I (LENGTH PRG))))

(DEFN MNEXT (PRG E I)
  (PLUS I
        (IF (LESSP (POSITION PRG E)
                   (REMAINDER I (LENGTH PRG)))
            (PLUS (POSITION PRG E)
                  (DIFFERENCE (LENGTH PRG)
                              (REMAINDER I (LENGTH PRG))))
          (DIFFERENCE (POSITION PRG E)
                      (REMAINDER I (LENGTH PRG))))))

(PROVE-LEMMA MNEXT-FIXES (REWRITE)
             (IMPLIES (AND (MEMBER E PRG)
                           (NOT (NUMBERP I)))
                      (EQUAL (MNEXT PRG E I)
                             (MNEXT PRG E 0))))

(PROVE-LEMMA NUMBERP-MNEXT (REWRITE)
             (IMPLIES (MEMBER E PRG)
                      (NUMBERP (MNEXT PRG E I))))


(PROVE-LEMMA MCHOOSE-CHOOSES (REWRITE)
             (IMPLIES (LISTP PRG)
                      (MEMBER (MCHOOSE PRG I) PRG)))

(PROVE-LEMMA MCHOOSE-FIXES (REWRITE)
             (IMPLIES (AND (LISTP PRG)
                           (NOT (NUMBERP I)))
                      (EQUAL (MCHOOSE PRG I)
                             (MCHOOSE PRG 0))))

(PROVE-LEMMA MNEXT-CHOICE-1 (REWRITE)
             (IMPLIES (MEMBER E PRG)
                      (NOT (LESSP (MNEXT PRG E I) I))))


(PROVE-LEMMA MNEXT-CHOICE-2-SIMPLIFIED (REWRITE)
   (IMPLIES
    (LESSP E N)
    (EQUAL (REMAINDER (PLUS I
                           (IF (LESSP E (REMAINDER I N))
                               (PLUS E (DIFFERENCE N
```

```
                                                  (REMAINDER I N)))
                                (DIFFERENCE E (REMAINDER I N))))
                        N)
             (FIX E))))


(PROVE-LEMMA MNEXT-CHOICE-2 (REWRITE)
             (IMPLIES (MEMBER E PRG)
                      (EQUAL (MCHOOSE PRG (MNEXT PRG E I)) E))
             ((USE (MNEXT-CHOICE-2-SIMPLIFIED (I I)
                                              (N (LENGTH PRG))
                                              (E (POSITION PRG E))))))

(PROVE-LEMMA REMAINDER-OF-ADD1 (REWRITE)
             (IMPLIES (AND (LESSP X (REMAINDER A B))
                           (LESSP X (REMAINDER (ADD1 A) B)))
                      (EQUAL (REMAINDER (ADD1 A) B)
                             (ADD1 (REMAINDER A B)))))

(PROVE-LEMMA REMAINDER-OF-ADD1-1 (REWRITE)
             (IMPLIES (AND (NOT (LESSP X (REMAINDER A B)))
                           (LESSP X (REMAINDER (ADD1 A) B)))
                      (EQUAL (REMAINDER A B)
                             (FIX X))))

(PROVE-LEMMA REMAINDER-OF-ADD1-2 (REWRITE)
             (IMPLIES (AND (LESSP X (REMAINDER A B))
                           (NOT (LESSP X (REMAINDER (ADD1 A) B))))
                      (AND (EQUAL (REMAINDER (ADD1 A) B)
                                  0)
                           (EQUAL (REMAINDER A B)
                                  (SUB1 B)))))

(PROVE-LEMMA REMAINDER-OF-ADD1-3 (REWRITE)
             (IMPLIES (AND (NOT (LESSP X (REMAINDER A B)))
                           (LESSP X B)
                           (NUMBERP X)
                           (NOT (EQUAL X (REMAINDER A B)))
                           (NOT (LESSP X (REMAINDER (ADD1 A) B))))
                      (EQUAL (REMAINDER (ADD1 A) B)
                             (ADD1 (REMAINDER A B)))))

(PROVE-LEMMA REMAINDER-OF-ADD1-3-1 (REWRITE)
    (IMPLIES (AND (NOT (LESSP (POSITION PRG E)
                              (REMAINDER I (LENGTH PRG))))
                  (NOT (LESSP (POSITION PRG E)
                              (REMAINDER (ADD1 I) (LENGTH PRG))))
                  (NOT (EQUAL (NTH PRG (REMAINDER I (LENGTH PRG)))
                              E))
                  (MEMBER E PRG))
             (EQUAL (REMAINDER (ADD1 I) (LENGTH PRG))
                    (ADD1 (REMAINDER I (LENGTH PRG))))))

(DISABLE MCHOOSE)
(DISABLE MNEXT)

;;; THE STATEMENT INTERPRETER

(DEFN N (OLD NEW E)
  (APPLY$ (CAR E) (APPEND (LIST OLD NEW) (CDR E))))


(DEFN-SK EXISTS-SUCCESSOR (OLD E)
  (EXISTS NEW (N OLD NEW E))
  ((SUFFIX X)))

(PROVE-LEMMA EXISTS-SUCCESSOR-IMPLIES (REWRITE)
             (IMPLIES (EXISTS-SUCCESSOR OLD E)
                      (N OLD (NEWX E OLD) E)))
```

```
(PROVE-LEMMA PROVE-EXISTS-SUCCESSOR (REWRITE)
             (IMPLIES (N OLD NEW E)
                      (EXISTS-SUCCESSOR OLD E))
             ((DISABLE EXISTS-SUCCESSOR)
              (USE (EXISTS-SUCCESSOR))))

(DISABLE EXISTS-SUCCESSOR)

(DEFN MS (PRG I)
  (IF (ZEROP I)
      NIL
    (IF (N (MS PRG (SUB1 I))
           (NEWX (MCHOOSE PRG (SUB1 I))
                 (MS PRG (SUB1 I)))
           (MCHOOSE PRG (SUB1 I)))
        (NEWX (MCHOOSE PRG (SUB1 I))
              (MS PRG (SUB1 I)))
      (MS PRG (SUB1 I)))))


(PROVE-LEMMA MS-TRANSITION-SUCCESSFUL (REWRITE)
             (IMPLIES (AND (LISTP PRG)
                           (N (MS PRG I) NEW (MCHOOSE PRG I)))
                      (N (MS PRG I) (MS PRG (ADD1 I))
                         (MCHOOSE PRG I)))
             ((DISABLE N)))

(PROVE-LEMMA MS-TRANSITION-IDLE (REWRITE)
             (IMPLIES (AND (LISTP PRG)
                           (NOT (N (MS PRG I) (NEWX (MCHOOSE PRG I)
                                                         (MS PRG I))
                                   (MCHOOSE PRG I))))
                      (EQUAL (MS PRG (ADD1 I))
                             (MS PRG I)))
             ((DISABLE N)))


;;; CHARACTERIZING AN ARBITRARY COMPUTATION

(CONSTRAIN COMPUTATION (REWRITE)
           (AND (IMPLIES (LISTP PRG)
                         (MEMBER (CHOOSE PRG I) PRG))
                (IMPLIES (MEMBER E PRG)
                         (NOT (LESSP (NEXT PRG E I) I)))
                (IMPLIES (MEMBER E PRG)
                         (EQUAL (CHOOSE PRG (NEXT PRG E I)) E))
                (IMPLIES (AND (LISTP PRG)
                              (N (S PRG I) NEW (CHOOSE PRG I)))
                         (N (S PRG I) (S PRG (ADD1 I)) (CHOOSE PRG I)))
                (IMPLIES (AND (LISTP PRG)
                              (NOT (N (S PRG I)
                                      (NEWX (CHOOSE PRG I)
                                            (S PRG I))
                                      (CHOOSE PRG I))))
                         (EQUAL (S PRG (ADD1 I))
                                (S PRG I)))
                (IMPLIES (MEMBER E PRG)
                         (NUMBERP (NEXT PRG E I)))
                (IMPLIES (AND (LISTP PRG)
                              (NOT (NUMBERP I)))
                         (EQUAL (S PRG I)
                                (S PRG 0)))
                (IMPLIES (AND (LISTP PRG)
                              (NOT (NUMBERP I)))
                         (EQUAL (CHOOSE PRG I)
                                (CHOOSE PRG 0)))
                (IMPLIES (AND (MEMBER E PRG)
                              (NOT (NUMBERP I)))
                         (EQUAL (NEXT PRG E I)
```

```
                                (NEXT PRG E 0))))
           ((CHOOSE MCHOOSE)
            (NEXT MNEXT)
            (S MS))
           ((DISABLE N)))

(DISABLE MNEXT-FIXES)
(DISABLE NUMBERP-MNEXT)
(DISABLE MCHOOSE-CHOOSES)
(DISABLE MCHOOSE-FIXES)
(DISABLE MNEXT-CHOICE-1)
(DISABLE MNEXT-CHOICE-2-SIMPLIFIED)
(DISABLE MNEXT-CHOICE-2)
(DISABLE REMAINDER-OF-ADD1)
(DISABLE REMAINDER-OF-ADD1-1)
(DISABLE REMAINDER-OF-ADD1-2)
(DISABLE REMAINDER-OF-ADD1-3)
(DISABLE REMAINDER-OF-ADD1-3-1)

(DISABLE EXISTS-SUCCESSOR-IMPLIES)
(DISABLE PROVE-EXISTS-SUCCESSOR)
(DISABLE MS-TRANSITION-SUCCESSFUL)
(DISABLE MS-TRANSITION-IDLE)

(DISABLE N)

;;; THE PREDICATE INTERPRETER

(DEFN EVAL (PRED STATE)
  (EVAL$ T PRED (LIST (CONS 'STATE STATE))))

(PROVE-LEMMA EVAL-NOT (REWRITE)
             (EQUAL (EVAL (LIST 'NOT P) STATE)
                    (NOT (EVAL P STATE))))

(PROVE-LEMMA EVAL-AND (REWRITE)
             (EQUAL (EVAL (LIST 'AND P Q) STATE)
                    (AND (EVAL P STATE)
                         (EVAL Q STATE))))

(PROVE-LEMMA EVAL-OR (REWRITE)
             (EQUAL (EVAL (LIST 'OR P Q) STATE)
                    (OR (EVAL P STATE)
                        (EVAL Q STATE))))

(PROVE-LEMMA EVAL-IMPLIES (REWRITE)
             (EQUAL (EVAL (LIST 'IMPLIES P Q) STATE)
                    (IMPLIES (EVAL P STATE)
                             (EVAL Q STATE))))

(PROVE-LEMMA EVAL-IFF (REWRITE)
             (EQUAL (EVAL (LIST 'IFF P Q) STATE)
                    (IFF (EVAL P STATE)
                         (EVAL Q STATE))))

(PROVE-LEMMA EVAL-EQUAL (REWRITE)
             (EQUAL (EVAL (LIST 'EQUAL P Q) STATE)
                    (EQUAL (EVAL P STATE)
                           (EVAL Q STATE))))

(PROVE-LEMMA EVAL-TRUE (REWRITE)
             (EQUAL (EVAL '(TRUE) STATE)
                    T))

(PROVE-LEMMA EVAL-FALSE (REWRITE)
             (EQUAL (EVAL '(FALSE) STATE)
                    F))

(DISABLE EVAL)
```

```
;;; THE STABILITY OPERATOR:  UNLESS

(DEFN-SK UNLESS (P Q PRG)
  (FORALL (OLD NEW E)
          (IMPLIES (AND (MEMBER E PRG)
                        (AND (N OLD NEW E)
                             (EVAL (LIST 'AND P (LIST 'NOT Q)) OLD)))
                   (EVAL (LIST 'OR P Q) NEW)))
  ((SUFFIX U)))

(DISABLE UNLESS)

(PROVE-LEMMA PROVE-UNLESS (REWRITE)
             (IMPLIES (IMPLIES (AND (MEMBER (EU P PRG Q) PRG)
                                    (N (OLDU P PRG Q) (NEWU P PRG Q)
                                       (EU P PRG Q))
                                    (EVAL P (OLDU P PRG Q))
                                    (NOT (EVAL Q (OLDU P PRG Q))))
                               (OR (EVAL P (NEWU P PRG Q))
                                   (EVAL Q (NEWU P PRG Q))))
                  (UNLESS P Q PRG))
             ((USE (UNLESS))))

(PROVE-LEMMA UNLESS-IMPLIES (REWRITE)
             (IMPLIES (AND (UNLESS P Q PRG)
                           (MEMBER E PRG)
                           (N OLD NEW E)
                           (EVAL P OLD)
                           (NOT (EVAL Q OLD)))
                      (EVAL (LIST 'OR P Q) NEW))
             ((USE (UNLESS))))

(DISABLE PROVE-UNLESS)
(DISABLE UNLESS-IMPLIES)


;;; THE PROGRESS OPERATOR:  LEADS-TO

(DEFN-SK LEADS-TO (P Q PRG)
  (FORALL I (IMPLIES (EVAL P (S PRG I))
                     (EXISTS J
                             (AND (NOT (LESSP J I))
                                  (EVAL Q (S PRG J))))))
  ((SUFFIX LEADS)))

(DISABLE LEADS-TO)

(PROVE-LEMMA PROVE-LEADS-TO (REWRITE)
             (IMPLIES (IMPLIES (EVAL P (S PRG (ILEADS P PRG Q)))
                               (AND (NOT (LESSP J (ILEADS P PRG Q)))
                                    (EVAL Q (S PRG J))))
                      (LEADS-TO P Q PRG))
             ((DISABLE EVAL)
              (USE (LEADS-TO))))

(PROVE-LEMMA LEADS-TO-IMPLIES (REWRITE)
             (IMPLIES (AND (LEADS-TO P Q PRG)
                           (EVAL P (S PRG I)))
                      (AND (NOT (LESSP (JLEADS I PRG Q) I))
                           (EVAL Q (S PRG (JLEADS I PRG Q)))))
             ((DISABLE EVAL)
              (USE (LEADS-TO))))

(DISABLE PROVE-LEADS-TO)
(DISABLE LEADS-TO-IMPLIES)

;;; THE MOST EFFECTIVE TRACE

(DEFN-SK SCHEDULABLE (PRG)
```

```
  (FORALL I (EXISTS NEW (N (S PRG I) NEW (CHOOSE PRG I))))
  ((SUFFIX S)))

(DISABLE SCHEDULABLE)

(PROVE-LEMMA PROVE-SCHEDULABLE (REWRITE)
             (IMPLIES (N (S PRG (IS PRG)) NEW
                         (CHOOSE PRG (IS PRG)))
                      (SCHEDULABLE PRG))
             ((USE (SCHEDULABLE))))

(PROVE-LEMMA SCHEDULABLE-IMPLIES (REWRITE)
             (IMPLIES (SCHEDULABLE PRG)
                      (N (S PRG I)
                         (NEWS I PRG)
                         (CHOOSE PRG I)))
             ((USE (SCHEDULABLE))))

(DISABLE PROVE-SCHEDULABLE)
(DISABLE SCHEDULABLE-IMPLIES)

(PROVE-LEMMA SCHEDULABLE-IMPLIES-EFFECTIVE-COMPUTATION (REWRITE)
             (IMPLIES (AND (SCHEDULABLE PRG)
                           (LISTP PRG))
                      (N (S PRG I)
                         (S PRG (ADD1 I))
                         (CHOOSE PRG I)))
             ((INSTRUCTIONS PROMOTE
                            (REWRITE COMPUTATION (($NEW (NEWS I PRG))))
                            (REWRITE SCHEDULABLE-IMPLIES))))

(PROVE-LEMMA COMPUTATION-N (REWRITE)
    (IMPLIES (AND (SCHEDULABLE PRG)
                  (MEMBER E PRG))
             (N (S PRG (NEXT PRG E I))
                (S PRG (ADD1 (NEXT PRG E I)))
                E))
    ((INSTRUCTIONS PROMOTE (DIVE 3) (= (CHOOSE PRG (NEXT PRG E I))) TOP
        (REWRITE SCHEDULABLE-IMPLIES-EFFECTIVE-COMPUTATION) (DROP 1)
        PROVE)))

(PROVE-LEMMA EFFECTIVE-IDLE (REWRITE)
             (IMPLIES (LISTP PRG)
                      (OR (EQUAL (S PRG (ADD1 I)) (S PRG I))
                          (N (S PRG I) (S PRG (ADD1 I))
                             (CHOOSE PRG I))))
             ((USE (COMPUTATION (NEW (NEWX (CHOOSE PRG I) (S PRG I)))))))

(DISABLE EFFECTIVE-IDLE)

;;; LEADS-TO PROOF RULES

(PROVE-LEMMA LEADS-TO-TRANSITIVE (REWRITE)
             (IMPLIES (AND (LEADS-TO P Q PRG)
                           (LEADS-TO Q R PRG))
                      (LEADS-TO P R PRG))
             ((ENABLE LEADS-TO-IMPLIES)
              (USE (PROVE-LEADS-TO (P P)
                                   (Q R)
                                   (PRG PRG)
                                   (J (JLEADS (JLEADS (ILEADS P PRG R)
                                                      PRG Q)
                                              PRG R))))))

(DISABLE LEADS-TO-TRANSITIVE)

(PROVE-LEMMA LEADS-TO-TRANSITIVE-GENERAL (REWRITE)
   (IMPLIES (AND (LEADS-TO P-1 Q-1 PRG)
                 (LEADS-TO Q-2 R-1 PRG)
```

```
                 (IMPLIES (EVAL Q-1 (S PRG (JLEADS (ILEADS P PRG R)
                                                  PRG Q-1)))
                          (EVAL Q-2 (S PRG (JLEADS (ILEADS P PRG R)
                                                  PRG Q-1))))
                 (IMPLIES (EVAL P (S PRG (ILEADS P PRG R)))
                          (EVAL P-1 (S PRG (ILEADS P PRG R))))
                 (IMPLIES (EVAL R-1 (S PRG (JLEADS (JLEADS
                                                   (ILEADS P PRG R)
                                                   PRG Q-1)
                                                  PRG R-1)))
                          (EVAL R (S PRG (JLEADS (JLEADS
                                                 (ILEADS P PRG R)
                                                 PRG Q-1)
                                                PRG R-1)))))
            (LEADS-TO P R PRG))
             ((ENABLE LEADS-TO-IMPLIES)
              (USE (PROVE-LEADS-TO (P P)
                                   (Q R)
                                   (PRG PRG)
                                   (J (JLEADS (JLEADS (ILEADS P PRG R)
                                                      PRG Q-1)
                                              PRG R-1))))))

(DISABLE LEADS-TO-TRANSITIVE-GENERAL)

(PROVE-LEMMA Q-LEADS-TO-Q (REWRITE)
             (LEADS-TO Q Q PRG)
             ((USE (PROVE-LEADS-TO (P Q)
                                   (J (ILEADS Q PRG Q))))))

(DISABLE Q-LEADS-TO-Q)

(PROVE-LEMMA FALSE-LEADS-TO-ANYTHING (REWRITE)
             (LEADS-TO '(FALSE) P PRG)
             ((USE (PROVE-LEADS-TO (P '(FALSE)) (Q P) (PRG PRG)))))

(DISABLE FALSE-LEADS-TO-ANYTHING)

(PROVE-LEMMA P-IMPLIES-Q-LEADS-TO (REWRITE)
             (IMPLIES (IMPLIES (EVAL P (S PRG (ILEADS P PRG Q)))
                               (EVAL Q (S PRG (ILEADS P PRG Q))))
                      (LEADS-TO P Q PRG))
             ((USE (PROVE-LEADS-TO (P P) (Q Q) (PRG PRG)
                                   (J (ILEADS P PRG Q))))))

(DISABLE P-IMPLIES-Q-LEADS-TO)

(PROVE-LEMMA LEADS-TO-STRENGTHEN-LEFT (REWRITE)
             (IMPLIES (AND (IMPLIES (EVAL Q (S PRG (ILEADS Q PRG R)))
                                    (EVAL P (S PRG (ILEADS Q PRG R))))
                           (LEADS-TO P R PRG))
                      (LEADS-TO Q R PRG))
             ((USE (PROVE-LEADS-TO (P Q)
                                   (Q R)
                                   (PRG PRG)
                                   (J (JLEADS (ILEADS Q PRG R) PRG R)))
                   (LEADS-TO-IMPLIES (P P)
                                     (Q R)
                                     (PRG PRG)
                                     (I (ILEADS Q PRG R))))))

(DISABLE LEADS-TO-STRENGTHEN-LEFT)

(PROVE-LEMMA LEADS-TO-WEAKEN-RIGHT (REWRITE)
  (IMPLIES (AND (IMPLIES (EVAL Q (S PRG (JLEADS (ILEADS P PRG R)
                                                PRG Q)))
                         (EVAL R (S PRG (JLEADS (ILEADS P PRG R)
                                                PRG Q))))
                (LEADS-TO P Q PRG))
```

```
           (LEADS-TO P R PRG))
             ((USE (PROVE-LEADS-TO (P P)
                                   (Q R)
                                   (PRG PRG)
                                   (J (JLEADS (ILEADS P PRG R) PRG Q)))
                   (LEADS-TO-IMPLIES (P P)
                                     (Q Q)
                                     (PRG PRG)
                                     (I (ILEADS P PRG R))))))

(DISABLE LEADS-TO-WEAKEN-RIGHT)

(PROVE-LEMMA LEADS-TO-MODIFY-BOTH (REWRITE)
  (IMPLIES (AND (IMPLIES (EVAL P (S PRG (ILEADS P PRG Q)))
                         (EVAL P-1 (S PRG (ILEADS P PRG Q))))
                (IMPLIES (EVAL Q-1 (S PRG (JLEADS (ILEADS P PRG Q)
                                                  PRG Q-1)))
                         (EVAL Q (S PRG (JLEADS (ILEADS P PRG Q)
                                                PRG Q-1))))
                (LEADS-TO P-1 Q-1 PRG))
           (LEADS-TO P Q PRG))
 ((USE (PROVE-LEADS-TO (P P)
                       (Q Q)
                       (PRG PRG)
                       (J (JLEADS (ILEADS P PRG Q) PRG Q-1)))
       (LEADS-TO-IMPLIES (P P-1)
                         (Q Q-1)
                         (PRG PRG)
                         (I (ILEADS P PRG Q))))))

(DISABLE LEADS-TO-MODIFY-BOTH)

(PROVE-LEMMA DISJOIN-LEFT (REWRITE)
             (IMPLIES (AND (LEADS-TO P R PRG)
                           (LEADS-TO Q R PRG))
                      (LEADS-TO (LIST 'OR P Q) R PRG))
             ((USE (PROVE-LEADS-TO (P (LIST 'OR P Q))
                                   (Q R)
                                   (PRG PRG)
                                   (J (JLEADS (ILEADS (LIST 'OR P Q)
                                                      PRG R)
                                              PRG R)))
                   (LEADS-TO-IMPLIES (P P)
                                     (Q R)
                                     (PRG PRG)
                                     (I (ILEADS (LIST 'OR P Q) PRG R)))
                   (LEADS-TO-IMPLIES (P Q)
                                     (Q R)
                                     (PRG PRG)
                                     (I (ILEADS (LIST 'OR P Q) PRG R))))))

(DISABLE DISJOIN-LEFT)


(PROVE-LEMMA DISJOIN-LEFT-GENERAL (REWRITE)
             (IMPLIES (AND (LEADS-TO P-1 Q-1 PRG)
                           (LEADS-TO P-2 Q-2 PRG)
                           (IMPLIES (EVAL P (S PRG (ILEADS P PRG Q)))
                                    (EVAL (LIST 'OR P-1 P-2)
                                          (S PRG (ILEADS P PRG Q))))
                           (IMPLIES (EVAL Q-1 (S PRG
                                                 (JLEADS (ILEADS P PRG Q)
                                                         PRG Q-1)))
                                    (EVAL Q (S PRG
                                               (JLEADS (ILEADS P PRG Q)
                                                       PRG Q-1))))
                           (IMPLIES (EVAL Q-2 (S PRG
                                                 (JLEADS (ILEADS P PRG Q)
                                                         PRG Q-2)))
```

```
                                    (EVAL Q (S PRG
                                               (JLEADS (ILEADS P PRG Q)
                                                       PRG Q-2)))))
                      (LEADS-TO P Q PRG))
             ((USE (PROVE-LEADS-TO (P P)
                                   (Q Q)
                                   (J (JLEADS (ILEADS P PRG Q)
                                              PRG Q-1)))
                   (PROVE-LEADS-TO (P P)
                                   (Q Q)
                                   (J (JLEADS (ILEADS P PRG Q)
                                              PRG Q-2)))
                   (LEADS-TO-IMPLIES (P P-1)
                                     (Q Q-1)
                                     (PRG PRG)
                                     (I (ILEADS P PRG Q)))
                   (LEADS-TO-IMPLIES (P P-2)
                                     (Q Q-2)
                                     (PRG PRG)
                                     (I (ILEADS P PRG Q))))))

(DISABLE DISJOIN-LEFT-GENERAL)

(PROVE-LEMMA CANCELLATION-LEADS-TO (REWRITE)
  (IMPLIES (AND (LEADS-TO P (LIST 'OR Q B) PRG)
                (LEADS-TO B R PRG))
           (LEADS-TO P (LIST 'OR Q R) PRG))
  ((USE (PROVE-LEADS-TO (P P)
                        (Q (LIST 'OR Q R))
                        (PRG PRG)
                        (J (JLEADS (JLEADS (ILEADS P PRG
                                                   (LIST 'OR Q R))
                                           PRG (LIST 'OR Q B))
                                   PRG R)))
        (PROVE-LEADS-TO (P P)
                        (Q (LIST 'OR Q R))
                        (PRG PRG)
                        (J (JLEADS (ILEADS P PRG
                                           (LIST 'OR Q R))
                                   PRG (LIST 'OR Q B))))
        (LEADS-TO-IMPLIES (P P)
                          (Q (LIST 'OR Q B))
                          (PRG PRG)
                          (I (ILEADS P PRG (LIST 'OR Q R))))
        (LEADS-TO-IMPLIES (P B)
                          (Q R)
                          (PRG PRG)
                          (I (JLEADS (ILEADS P PRG
                                             (LIST 'OR Q R))
                                     PRG (LIST 'OR Q B)))))))

(DISABLE CANCELLATION-LEADS-TO)

(PROVE-LEMMA CANCELLATION-LEADS-TO-GENERAL (REWRITE)
  (IMPLIES (AND (LEADS-TO P-1 D PRG)
                (LEADS-TO B R-1 PRG)
                (IMPLIES (EVAL (LIST 'AND D (LIST 'NOT B))
                               (S PRG (JLEADS (ILEADS P PRG R)
                                              PRG D)))
                         (EVAL R (S PRG (JLEADS (ILEADS P PRG R)
                                                PRG D))))
                (IMPLIES (EVAL R-1
                               (S PRG
                                  (JLEADS (JLEADS (ILEADS P PRG R)
                                                  PRG D)
                                          PRG R-1)))
                         (EVAL R (S PRG
                                    (JLEADS (JLEADS (ILEADS P PRG R)
                                                    PRG D)
```

```
                                        PRG R-1))))
                (IMPLIES (EVAL P (S PRG (ILEADS P PRG R)))
                         (EVAL P-1 (S PRG (ILEADS P PRG R)))))
           (LEADS-TO P R PRG))
  ((USE (PROVE-LEADS-TO (P P)
                        (Q R)
                        (PRG PRG)
                        (J (JLEADS (ILEADS P PRG R)
                                   PRG D)))
        (PROVE-LEADS-TO (P P)
                        (Q R)
                        (PRG PRG)
                        (J (JLEADS (JLEADS (ILEADS P PRG R)
                                           PRG D)
                                   PRG R-1)))
        (LEADS-TO-IMPLIES (P P-1)
                          (Q D)
                          (PRG PRG)
                          (I (ILEADS P PRG R)))
        (LEADS-TO-IMPLIES (P B)
                          (Q R-1)
                          (PRG PRG)
                          (I (JLEADS (ILEADS P PRG R)
                                     PRG D))))))

(DISABLE CANCELLATION-LEADS-TO-GENERAL)

(DEFN ENSURES-INTERVAL (PRG Q TOP I)
  (IF (LESSP TOP I)
      (FIX TOP)
    (IF (EVAL Q (S PRG I))
        (FIX I)
      (ENSURES-INTERVAL PRG Q TOP (ADD1 I))))
  ((LESSP (DIFFERENCE (ADD1 TOP) I))))

(PROVE-LEMMA ENSURES-INTERVAL-FIXES (REWRITE)
             (IMPLIES (AND (LISTP PRG)
                           (NOT (NUMBERP I)))
                      (EQUAL (ENSURES-INTERVAL PRG Q TOP I)
                             (ENSURES-INTERVAL PRG Q TOP 0))))

(PROVE-LEMMA ENSURES-INTERVAL-BIGGER (REWRITE)
             (EQUAL (LESSP (ENSURES-INTERVAL PRG Q TOP I) I)
                    (LESSP TOP I))
             ((ENABLE EQUAL-IFF)))

(PROVE-LEMMA PSP-PROVES-SOMETHING (REWRITE)
             (IMPLIES (AND (LEADS-TO P Q PRG)
                           (UNLESS R B PRG)
                           (LISTP PRG)
                           (NUMBERP I)
                           (EVAL P (S PRG BASE))
                           (EVAL R (S PRG I))
                           (LESSP (JLEADS BASE PRG Q) TOP)
                           (NOT (LESSP (JLEADS BASE PRG Q) I))
                           (EQUAL FINAL (LIST 'OR Q B)))
                      (EVAL (LIST 'OR (LIST 'AND Q R) B)
                            (S PRG (ENSURES-INTERVAL PRG FINAL TOP I))))
     ((INSTRUCTIONS
       (INDUCT (ENSURES-INTERVAL PRG FINAL TOP I)) PROVE
       PROVE S SPLIT (CLAIM (EVAL Q (S PRG I)) 0) PROVE
       (CONTRADICT 14) (DIVE 2 2) (= (FIX (JLEADS BASE PRG Q))) UP
       (= (S PRG (JLEADS BASE PRG Q))) TOP
       (REWRITE LEADS-TO-IMPLIES)
       PROVE (CLAIM (EVAL B (S PRG (ADD1 I))) 0) PROVE
       (CLAIM (EVAL (LIST 'OR R B) (S PRG (ADD1 I))) 0) PROVE
       (CONTRADICT 14)
       (CLAIM (OR (EQUAL (S PRG (ADD1 I)) (S PRG I))
                  (N (S PRG I) (S PRG (ADD1 I)) (CHOOSE PRG I)))
```

```
            0)
       SPLIT
       (REWRITE UNLESS-IMPLIES
                (($PRG PRG) ($E (CHOOSE PRG I)) ($OLD (S PRG I))))
       (REWRITE COMPUTATION) PROVE PROVE (CONTRADICT 15) SPLIT
       (REWRITE COMPUTATION (($NEW (NEWX (CHOOSE PRG I) (S PRG I)))))
       PROVE)))

(DISABLE PSP-PROVES-SOMETHING)

(PROVE-LEMMA PSP (REWRITE)
    (IMPLIES (AND (LEADS-TO P Q PRG)
                  (UNLESS R B PRG)
                  (LISTP PRG))
             (LEADS-TO (LIST 'AND P R)
                       (LIST 'OR (LIST 'AND Q R) B)
                       PRG))
    ((INSTRUCTIONS PROMOTE
         (REWRITE PROVE-LEADS-TO
                  (($J (ENSURES-INTERVAL PRG (LIST 'OR Q B)
                           (ADD1 (JLEADS (ILEADS (LIST 'AND P R) PRG
                                                  (LIST 'OR (LIST 'AND Q R) B))
                                         PRG Q))
                           (ILEADS (LIST 'AND P R) PRG
                                   (LIST 'OR (LIST 'AND Q R) B))))))
         (USE-LEMMA PSP-PROVES-SOMETHING
             ((P P) (Q Q) (R R) (PRG PRG) (FINAL (LIST 'OR Q B))
              (TOP (ADD1 (JLEADS (ILEADS (LIST 'AND P R) PRG
                                         (LIST 'OR (LIST 'AND Q R) B))
                                 PRG Q)))
              (I (FIX (ILEADS (LIST 'AND P R) PRG
                              (LIST 'OR (LIST 'AND Q R) B))))
              (BASE (ILEADS (LIST 'AND P R) PRG
                            (LIST 'OR (LIST 'AND Q R) B)))))
         (USE-LEMMA LEADS-TO-IMPLIES
             ((P P) (Q Q) (PRG PRG)
              (I (ILEADS (LIST 'AND P R) PRG
                         (LIST 'OR (LIST 'AND Q R) B)))))
         PROVE)))

(DISABLE PSP)

(PROVE-LEMMA PSP-GENERAL (REWRITE)
    (IMPLIES
     (AND (LEADS-TO P Q PRG)
          (UNLESS R B PRG)
          (IMPLIES (EVAL PR (S PRG (ILEADS PR PRG QB)))
                   (EVAL (LIST 'AND P R)
                         (S PRG (ILEADS PR PRG QB))))
          (IMPLIES (EVAL (LIST 'OR (LIST 'AND Q R) B)
                         (S PRG (JLEADS (ILEADS PR PRG QB)
                                        PRG
                                        (LIST 'OR (LIST 'AND Q R) B))))
                   (EVAL QB (S PRG (JLEADS (ILEADS PR PRG QB)
                                           PRG
                                           (LIST 'OR (LIST 'AND Q R)
                                                 B)))))
          (LISTP PRG))
     (LEADS-TO PR QB PRG))
    ((ENABLE PSP)
     (USE (LEADS-TO-MODIFY-BOTH (P PR) (Q QB)
                                (P-1 (LIST 'AND P R))
                                (Q-1 (LIST 'OR (LIST 'AND Q R) B))))))

(DISABLE PSP-GENERAL)

(PROVE-LEMMA LEADS-TO-TRUE (REWRITE)
             (LEADS-TO P '(TRUE) PRG)
             ((USE (PROVE-LEADS-TO (P P) (Q '(TRUE))
```

```
                                    (J (ILEADS P PRG '(TRUE)))))))

;;; INITIAL CONDITION AND INVARIANTS

(DEFN INITIAL-CONDITION (IC PRG)
  (EVAL IC (S PRG 0)))

(DISABLE INITIAL-CONDITION)

(DEFN-SK INVARIANT (INV PRG)
  (FORALL I (EVAL INV (S PRG I)))
  ((SUFFIX I)))

(DISABLE INVARIANT)

(PROVE-LEMMA PROVE-INVARIANT (REWRITE)
             (IMPLIES (EVAL INV (S PRG (II INV PRG)))
                      (INVARIANT INV PRG))
             ((USE (INVARIANT))))

(PROVE-LEMMA INVARIANT-IMPLIES (REWRITE)
             (IMPLIES (INVARIANT INV PRG)
                      (EVAL INV (S PRG I)))
             ((USE (INVARIANT))))

(DISABLE PROVE-INVARIANT)

(PROVE-LEMMA INVARIANT-CONSEQUENCE (REWRITE)
             (IMPLIES (AND (INVARIANT P PRG)
                           (IMPLIES (EVAL P (S PRG (II Q PRG)))
                                    (EVAL Q (S PRG (II Q PRG)))))
                      (INVARIANT Q PRG))
             ((ENABLE PROVE-INVARIANT)))

(PROVE-LEMMA INVARIANTS-PERSIST-GENERAL (REWRITE)
             (IMPLIES (AND (UNLESS P '(FALSE) PRG)
                           (EVAL P (S PRG I))
                           (LISTP PRG)
                           (NOT (LESSP J I)))
                      (EVAL P (S PRG J)))
             ((INDUCT (PLUS J I))
              (USE (UNLESS-IMPLIES (P P)
                                   (Q '(FALSE))
                                   (OLD (S PRG (SUB1 J)))
                                   (NEW (S PRG J))
                                   (E (CHOOSE PRG (SUB1 J))))
                   (EFFECTIVE-IDLE (PRG PRG) (I (SUB1 J))))))

(DISABLE INVARIANTS-PERSIST-GENERAL)

(PROVE-LEMMA UNLESS-PROVES-INVARIANT (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION IC PRG)
                           (UNLESS P '(FALSE) PRG)
                           (IMPLIES (EVAL IC (S PRG 0))
                                    (EVAL P (S PRG 0)))
                           (LISTP PRG))
                      (INVARIANT P PRG))
             ((ENABLE PROVE-INVARIANT INITIAL-CONDITION
                      INVARIANTS-PERSIST-GENERAL)))

(PROVE-LEMMA LEADS-TO-FALSE-INVARIANT (REWRITE)
  (IMPLIES (AND (LEADS-TO P '(FALSE) PRG)
                (IMPLIES (EVAL (LIST 'NOT P) (S PRG (II INV PRG)))
                         (EVAL INV (S PRG (II INV PRG)))))
           (INVARIANT INV PRG))
             ((USE (LEADS-TO-IMPLIES (P P) (Q '(FALSE)) (PRG PRG)
                                     (I (II INV PRG)))
                   (PROVE-INVARIANT (INV INV) (PRG PRG)))))
```

```
;;; EVENTUAL STABILITY

(DEFN-SK EVENTUALLY-INVARIANT (R PRG)
  (EXISTS I (FORALL J (IMPLIES (NOT (LESSP J I))
                               (EVAL R (S PRG J)))))
  ((SUFFIX ES)))

(DISABLE EVENTUALLY-INVARIANT)

(PROVE-LEMMA PROVE-EVENTUALLY-INVARIANT (REWRITE)
             (IMPLIES (IMPLIES (NOT (LESSP (JES I PRG R) I))
                               (EVAL R (S PRG (JES I PRG R))))
                      (EVENTUALLY-INVARIANT R PRG))
             ((USE (EVENTUALLY-INVARIANT))))

(PROVE-LEMMA EVENTUALLY-INVARIANT-IMPLIES (REWRITE)
         (IMPLIES (AND (EVENTUALLY-INVARIANT R PRG)
                       (NOT (LESSP J (IES PRG R))))
                  (EVAL R (S PRG J)))
         ((USE (EVENTUALLY-INVARIANT))))

(DISABLE PROVE-EVENTUALLY-INVARIANT)
(DISABLE EVENTUALLY-INVARIANT-IMPLIES)

(PROVE-LEMMA NOT-LEADS-TO-PROVES-EVENTUALLY-INVARIANT (REWRITE)
   (IMPLIES (AND (NOT (LEADS-TO P NOT-R PRG))
                 (IMPLIES (NOT (EVAL NOT-R
                                     (S PRG (JES (ILEADS P PRG NOT-R)
                                                 PRG R))))
                          (EVAL R (S PRG (JES (ILEADS P PRG NOT-R)
                                              PRG R)))))
            (EVENTUALLY-INVARIANT R PRG))
             ((USE (PROVE-LEADS-TO (P P)
                                   (Q NOT-R)
                                   (PRG PRG)
                                   (J (JES (ILEADS P PRG NOT-R)
                                           PRG R)))
                   (PROVE-EVENTUALLY-INVARIANT (I (ILEADS P PRG NOT-R))
                                               (R R)
                                               (PRG PRG)))))

(DISABLE NOT-LEADS-TO-PROVES-EVENTUALLY-INVARIANT)

(PROVE-LEMMA NOT-EVENTUALLY-INVARIANT-PROVES-LEADS-TO (REWRITE)
  (IMPLIES (AND (NOT (EVENTUALLY-INVARIANT NOT-Q PRG))
                (IMPLIES (NOT (EVAL NOT-Q
                                    (S PRG (JES (ILEADS P PRG Q)
                                                PRG NOT-Q))))
                         (EVAL Q (S PRG (JES (ILEADS P PRG Q)
                                             PRG NOT-Q)))))
           (LEADS-TO P Q PRG))
  ((USE (NOT-LEADS-TO-PROVES-EVENTUALLY-INVARIANT
        (P P)
        (NOT-R Q)
        (R NOT-Q)
        (PRG PRG)))))

(DISABLE NOT-EVENTUALLY-INVARIANT-PROVES-LEADS-TO)

(PROVE-LEMMA TRUE-LEADS-TO-PROVES-NOT-EVENTUALLY-INVARIANT (REWRITE)
  (IMPLIES (AND (LEADS-TO '(TRUE) NOT-R PRG)
                (IMPLIES (EVAL NOT-R (S PRG (JLEADS (IES PRG R)
                                                    PRG NOT-R)))
                         (NOT (EVAL R (S PRG (JLEADS (IES PRG R)
                                                     PRG NOT-R))))))
           (NOT (EVENTUALLY-INVARIANT R PRG)))
 ((USE (LEADS-TO-IMPLIES (P '(TRUE)) (Q NOT-R) (PRG PRG)
                         (I (IES PRG R)))
```

```
      (EVENTUALLY-INVARIANT-IMPLIES (R R) (PRG PRG)
                                    (J (JLEADS (IES PRG R)
                                               PRG NOT-R))))))

(DISABLE TRUE-LEADS-TO-PROVES-NOT-EVENTUALLY-INVARIANT)

(PROVE-LEMMA EVENTUALLY-INVARIANT-PROVES-NOT-TRUE-LEADS-TO (REWRITE)
  (IMPLIES (AND (EVENTUALLY-INVARIANT NOT-Q PRG)
                (IMPLIES (EVAL NOT-Q (S PRG (JLEADS (IES PRG NOT-Q)
                                                    PRG Q)))
                         (NOT (EVAL Q (S PRG (JLEADS (IES PRG NOT-Q)
                                                     PRG Q))))))
           (NOT (LEADS-TO '(TRUE) Q PRG)))
  ((USE (TRUE-LEADS-TO-PROVES-NOT-EVENTUALLY-INVARIANT
        (R NOT-Q) (NOT-R Q) (PRG PRG)))))

(DISABLE EVENTUALLY-INVARIANT-PROVES-NOT-TRUE-LEADS-TO)

(PROVE-LEMMA EVENTUALLY-INVARIANT-WEAKEN (REWRITE)
  (IMPLIES (AND (EVENTUALLY-INVARIANT P PRG)
                (IMPLIES (EVAL P (S PRG (JES (IES PRG P) PRG R)))
                         (EVAL R (S PRG (JES (IES PRG P) PRG R)))))
           (EVENTUALLY-INVARIANT R PRG))
   ((USE (EVENTUALLY-INVARIANT-IMPLIES (R P) (PRG PRG)
                                      (J (JES (IES PRG P) PRG R)))
         (PROVE-EVENTUALLY-INVARIANT (R R) (PRG PRG)
                                    (I (IES PRG P))))))

(DISABLE EVENTUALLY-INVARIANT-WEAKEN)

(PROVE-LEMMA EVENTUALLY-INVARIANT-CONJUNCTION (REWRITE)
  (IMPLIES (AND (EVENTUALLY-INVARIANT P PRG)
                (EVENTUALLY-INVARIANT Q PRG)
                (IMPLIES (EVAL (LIST 'AND P Q)
                               (S PRG (JES (IF (LESSP (IES PRG P)
                                                      (IES PRG Q))
                                               (IES PRG Q)
                                             (IES PRG P))
                                           PRG R)))
                         (EVAL R (S PRG
                                    (JES (IF (LESSP (IES PRG P)
                                                    (IES PRG Q))
                                             (IES PRG Q)
                                           (IES PRG P))
                                         PRG R)))))
           (EVENTUALLY-INVARIANT R PRG))
   ((USE (PROVE-EVENTUALLY-INVARIANT (R R) (PRG PRG)
                                    (I (IF (LESSP (IES PRG P)
                                                  (IES PRG Q))
                                           (IES PRG Q)
                                         (IES PRG P))))
         (EVENTUALLY-INVARIANT-IMPLIES (R P) (PRG PRG)
                                      (J (JES (IF (LESSP (IES PRG P)
                                                         (IES PRG Q))
                                                  (IES PRG Q)
                                                (IES PRG P))
                                              PRG R)))
         (EVENTUALLY-INVARIANT-IMPLIES (R Q) (PRG PRG)
                                      (J (JES (IF (LESSP (IES PRG P)
                                                         (IES PRG Q))
                                                  (IES PRG Q)
                                                (IES PRG P))
                                              PRG R))))))

(DISABLE EVENTUALLY-INVARIANT-CONJUNCTION)

(PROVE-LEMMA EVENTUALLY-INVARIANT-FALSE (REWRITE)
  (IMPLIES (AND (LEADS-TO P Q PRG)
                (IMPLIES (EVAL Q (S PRG (JLEADS (IES PRG P)
```

```
                                                PRG Q)))
                          (NOT (EVAL P (S PRG (JLEADS (IES PRG P)
                                                     PRG Q))))))
            (NOT (EVENTUALLY-INVARIANT P PRG)))
   ((USE (LEADS-TO-IMPLIES (P P) (Q Q) (PRG PRG)
                           (I (IES PRG P)))
         (EVENTUALLY-INVARIANT-IMPLIES (R P) (PRG PRG)
                                       (J (JLEADS (IES PRG P)
                                                  PRG Q)))
         (EVENTUALLY-INVARIANT-IMPLIES (R P) (PRG PRG) (IN IN)
                                       (J (IES PRG P))))))

(DISABLE EVENTUALLY-INVARIANT-FALSE)

(PROVE-LEMMA STABLE-OCCURS-PROVES-EVENTUALLY-INVARIANT (REWRITE)
             (IMPLIES (AND (LISTP PRG)
                           (UNLESS P '(FALSE) PRG)
                           (LEADS-TO '(TRUE) P PRG))
                      (EVENTUALLY-INVARIANT P PRG))
    ((INSTRUCTIONS PROMOTE
         (REWRITE PROVE-EVENTUALLY-INVARIANT (($I (JLEADS 0 PRG P))))
         PROMOTE
         (REWRITE INVARIANTS-PERSIST-GENERAL (($I (JLEADS 0 PRG P))))
         (REWRITE LEADS-TO-IMPLIES (($P '(TRUE))))
         (BASH (DISABLE EVAL)))))

(DISABLE STABLE-OCCURS-PROVES-EVENTUALLY-INVARIANT)



;;; THE BASIC ENSURES OPERATOR

(DEFN-SK ENSURES (P Q PRG)
  (EXISTS E
          (AND (MEMBER E PRG)
               (FORALL (OLD NEW)
                       (IMPLIES
                        (AND (N OLD NEW E)
                             (EVAL (LIST 'AND P (LIST 'NOT Q)) OLD))
                        (EVAL Q NEW)))))
  ((SUFFIX E)))

(DISABLE ENSURES)

(PROVE-LEMMA PROVE-ENSURES (REWRITE)
             (IMPLIES (AND (MEMBER E PRG)
                           (IMPLIES (AND (N (OLDE E P Q) (NEWE E P Q) E)
                                         (EVAL P (OLDE E P Q))
                                         (NOT (EVAL Q (OLDE E P Q))))
                                    (EVAL Q (NEWE E P Q))))
                  (ENSURES P Q PRG))
             ((USE (ENSURES))))

(PROVE-LEMMA ENSURES-IMPLIES (REWRITE)
             (AND (IMPLIES (ENSURES P Q PRG)
                           (MEMBER (EE P PRG Q) PRG))
                  (IMPLIES (AND (ENSURES P Q PRG)
                                (N OLD NEW (EE P PRG Q))
                                (EVAL P OLD)
                                (NOT (EVAL Q OLD)))
                           (EVAL Q NEW)))
             ((USE (ENSURES))))

(DISABLE PROVE-ENSURES)
(DISABLE ENSURES-IMPLIES)

(PROVE-LEMMA ENSURES-PROVES-SOMETHING (REWRITE)
             (IMPLIES (AND (ENSURES P Q PRG)
                           (UNLESS P Q PRG)
```

```
                           (SCHEDULABLE PRG)
                           (LESSP (NEXT PRG (EE P PRG Q) BASE) TOP)
                           (NOT (LESSP (NEXT PRG (EE P PRG Q) BASE) I))
                           (NOT (LESSP I BASE))
                           (NUMBERP I)
                           (EVAL P (S PRG I)))
                      (EVAL Q (S PRG (ENSURES-INTERVAL PRG Q TOP I))))
    ((INSTRUCTIONS PROMOTE
         (CLAIM (MEMBER (EE P PRG Q) PRG) ((ENABLE ENSURES-IMPLIES)))
         (CLAIM (LISTP PRG)) (INDUCT (ENSURES-INTERVAL PRG Q TOP I))
         PROVE PROVE S SPLIT (DIVE 2 2) X X TOP
         (CLAIM (EVAL Q (S PRG (ADD1 I))) 0) PROVE (CONTRADICT 14)
         (REWRITE ENSURES-IMPLIES (($P P) ($PRG PRG) ($OLD (S PRG I))))
         PROVE PROVE PROVE
         (CLAIM (EVAL (LIST 'OR P Q) (S PRG (ADD1 I))) 0) PROVE
         (CONTRADICT 16)
         (REWRITE UNLESS-IMPLIES
                  (($PRG PRG) ($E (CHOOSE PRG I)) ($OLD (S PRG I))))
         PROVE PROVE)))

(PROVE-LEMMA THE-INTERVAL-OF-ENSURES (REWRITE)
             (IMPLIES (AND (ENSURES P Q PRG)
                           (UNLESS P Q PRG)
                           (SCHEDULABLE PRG)
                           (EVAL P (S PRG I)))
                      (EVAL Q
                            (S PRG (ENSURES-INTERVAL
                                    PRG Q
                                    (ADD1 (NEXT PRG (EE P PRG Q) I)) I))))
    ((INSTRUCTIONS PROMOTE (CLAIM (NUMBERP I) 0)
         (REWRITE ENSURES-PROVES-SOMETHING (($P P) ($BASE I))) PROVE
         (PROVE (ENABLE ENSURES-IMPLIES)) PROVE (DIVE 2 2)
         (REWRITE ENSURES-INTERVAL-FIXES) TOP
         (REWRITE ENSURES-PROVES-SOMETHING (($P P) ($BASE 0)))
         (PROVE (ENABLE ENSURES-IMPLIES)) PROVE PROVE (DEMOTE 4)
         (DIVE 1 2) (REWRITE COMPUTATION) TOP S
         (CLAIM (MEMBER (EE P PRG Q) PRG) ((ENABLE ENSURES-IMPLIES)))
         PROVE
         (CLAIM (MEMBER (EE P PRG Q) PRG) ((ENABLE ENSURES-IMPLIES)))
         PROVE)))

(PROVE-LEMMA ENSURES-PROVES-LEADS-TO (REWRITE)
    (IMPLIES (AND (SCHEDULABLE PRG)
                  (UNLESS P Q PRG)
                  (ENSURES P Q PRG))
             (LEADS-TO P Q PRG))
    ((USE (PROVE-LEADS-TO (J (ENSURES-INTERVAL
                               PRG Q
                               (ADD1 (NEXT PRG (EE P PRG Q)
                                           (ILEADS P PRG Q)))
                               (ILEADS P PRG Q)))
                          (P P) (Q Q) (PRG PRG)))
     (ENABLE ENSURES-IMPLIES)))

(DISABLE ENSURES-PROVES-SOMETHING)
(DISABLE THE-INTERVAL-OF-ENSURES)

(DISABLE ENSURES-PROVES-LEADS-TO)

;;; TOTAL PROGRAMS, STATEMENTS ARE ASSIGNMENTS

(DEFN-SK TOTAL (PRG)
  (FORALL E (IMPLIES (MEMBER E PRG)
                     (FORALL OLD
                             (EXISTS NEW (N OLD NEW E)))))
  ((SUFFIX T)))

(DISABLE TOTAL)
```

```
(PROVE-LEMMA PROVE-TOTAL (REWRITE)
             (IMPLIES (IMPLIES (MEMBER (ET PRG) PRG)
                                 (N (OLDT PRG) NEW (ET PRG)))
                      (TOTAL PRG))
             ((USE (TOTAL))))

(PROVE-LEMMA TOTAL-IMPLIES (REWRITE)
           (IMPLIES (AND (TOTAL PRG)
                         (MEMBER E PRG))
                    (N OLD (NEWT E OLD) E))
           ((USE (TOTAL))))

(DISABLE PROVE-TOTAL)
(DISABLE TOTAL-IMPLIES)

(PROVE-LEMMA TOTAL-IMPLIES-SCHEDULABLE (REWRITE)
             (IMPLIES (AND (TOTAL PRG)
                           (LISTP PRG))
                      (SCHEDULABLE PRG))
             ((USE (PROVE-SCHEDULABLE (NEW (NEWT (CHOOSE PRG (IS PRG))
                                                 (S PRG (IS PRG))))))
              (ENABLE TOTAL-IMPLIES)))


;;; ENABLED TRANSITIONS

(DEFN-SK ENABLING-CONDITION (C E PRG)
  (AND (MEMBER E PRG)
       (FORALL (OLD NEW)
               (IMPLIES (N OLD NEW E)
                        (EVAL C OLD)))
       (FORALL OLD (IMPLIES (EVAL C OLD)
                            (EXISTS NEW (N OLD NEW E)))))
  ((SUFFIX C)))

(DISABLE ENABLING-CONDITION)

(PROVE-LEMMA PROVE-ENABLING-CONDITION (REWRITE)
             (IMPLIES (AND (MEMBER E PRG)
                           (IMPLIES (N (OLDC C E) (NEWC C E) E)
                                    (EVAL C (OLDC C E)))
                           (IMPLIES (EVAL C (OLDC-1 C E))
                                    (N (OLDC-1 C E) NEW E)))
                      (ENABLING-CONDITION C E PRG))
             ((USE (ENABLING-CONDITION))))

(PROVE-LEMMA ENABLING-CONDITION-IMPLIES (REWRITE)
             (AND (IMPLIES (ENABLING-CONDITION C E PRG)
                           (MEMBER E PRG))
                  (IMPLIES (AND (ENABLING-CONDITION C E PRG)
                                (N OLD NEW E))
                           (EVAL C OLD))
                  (IMPLIES (AND (ENABLING-CONDITION C E PRG)
                                (EVAL C OLD))
                           (N OLD (NEWC-1 E OLD) E)))
             ((USE (ENABLING-CONDITION))))

(DISABLE PROVE-ENABLING-CONDITION)
(DISABLE ENABLING-CONDITION-IMPLIES)

;;; ENSURES WITH ENABLING

(DEFN-SK E-ENSURES (P Q C PRG)
  (EXISTS E
          (AND (MEMBER E PRG)
               (ENABLING-CONDITION C E PRG)
               (FORALL (OLD NEW)
                       (IMPLIES
                        (AND (N OLD NEW E)
```

```
                             (EVAL (LIST 'AND P (LIST 'NOT Q)) OLD))
                        (EVAL Q NEW)))))
  ((SUFFIX EE)))

(DISABLE E-ENSURES)

(PROVE-LEMMA PROVE-E-ENSURES (REWRITE)
  (IMPLIES (AND (MEMBER E PRG)
                (ENABLING-CONDITION C E PRG)
                (IMPLIES (AND (N (OLDEE E P Q) (NEWEE E P Q) E)
                              (EVAL (LIST 'AND P (LIST 'NOT Q))
                                    (OLDEE E P Q)))
                         (EVAL Q (NEWEE E P Q))))
           (E-ENSURES P Q C PRG))
  ((USE (E-ENSURES))))

(DISABLE PROVE-E-ENSURES)

(DEFN-SK E-ENSURES-ENABLING (P Q C PRG)
  (EXISTS E
          (AND
           (MEMBER E PRG)
           (FORALL (OLD NEW)
                   (AND (IMPLIES (N OLD NEW E)
                                 (EVAL C OLD))
                        (IMPLIES (AND (N OLD NEW E)
                                      (EVAL (LIST 'AND P (LIST 'NOT Q))
                                            OLD))
                                 (EVAL Q NEW))))
           (FORALL OLD (IMPLIES (EVAL C OLD)
                                (EXISTS NEW (N OLD NEW E))))))
  ((SUFFIX EEE)))

(DISABLE E-ENSURES-ENABLING)

(PROVE-LEMMA HELP-PROVE-E-ENSURES (REWRITE)
             (IMPLIES (AND (MEMBER E PRG)
                           (IMPLIES (N (OLDEEE C E P Q)
                                       (NEWEEE C E P Q)
                                       E)
                                    (AND (EVAL C (OLDEEE C E P Q))
                                         (EVAL Q (NEWEEE C E P Q))))
                           (IMPLIES (EVAL C (OLDEEE-1 C E))
                                    (N (OLDEEE-1 C E) NEW E))
                           (IMPLIES (EVAL C (OLDEEE C E P Q))
                                    (EVAL (LIST 'AND P (LIST 'NOT Q))
                                          (OLDEEE C E P Q))))
                      (E-ENSURES P Q C PRG))
  ((USE (PROVE-E-ENSURES (E (EEEE C P PRG Q)))
        (PROVE-ENABLING-CONDITION
         (E (EEEE C P PRG Q))
         (NEW (NEWEEE-1
               C
               (OLDC-1 C (EEEE C P PRG Q)) P PRG Q)))
        (E-ENSURES-ENABLING (OLD (OLDEE (EEEE C P PRG Q) P Q))
                            (NEW (NEWEE (EEEE C P PRG Q) P Q)))
        (E-ENSURES-ENABLING (OLD (OLDC C (EEEE C P PRG Q)))
                            (NEW (NEWC C (EEEE C P PRG Q))))
        (E-ENSURES-ENABLING (OLD (OLDC-1 C (EEEE C P PRG Q)))))))

(PROVE-LEMMA E-ENSURES-IMPLIES (REWRITE)
             (AND (IMPLIES (E-ENSURES P Q C PRG)
                           (MEMBER (EEE C P PRG Q) PRG))
                  (IMPLIES (E-ENSURES P Q C PRG)
                           (ENABLING-CONDITION C
                                               (EEE C P PRG Q)
                                               PRG))
                  (IMPLIES (AND (E-ENSURES P Q C PRG)
                                (N OLD NEW (EEE C P PRG Q))
```

```
                                (EVAL (LIST 'AND P (LIST 'NOT Q))
                                      OLD))
                           (EVAL Q NEW)))
             ((USE (E-ENSURES))))

(DISABLE E-ENSURES-IMPLIES)
(DISABLE HELP-PROVE-E-ENSURES)


;;; UNION THEOREMS

(PROVE-LEMMA TOTAL-UNION-1 (REWRITE)
             (IMPLIES (TOTAL (APPEND PRG-1 PRG-2))
                      (AND (TOTAL PRG-1)
                           (TOTAL PRG-2)))
             ((USE (PROVE-TOTAL (PRG PRG-1)
                                (NEW (NEWT (ET PRG-1)
                                            (OLDT PRG-1))))
                   (PROVE-TOTAL (PRG PRG-2)
                                (NEW (NEWT (ET PRG-2)
                                           (OLDT PRG-2)))))
              (ENABLE TOTAL-IMPLIES)))

(PROVE-LEMMA TOTAL-UNION-2 (REWRITE)
    (IMPLIES (AND (TOTAL PRG-1) (TOTAL PRG-2))
             (TOTAL (APPEND PRG-1 PRG-2)))
    ((INSTRUCTIONS
         (USE-LEMMA PROVE-TOTAL
             ((PRG (APPEND PRG-1 PRG-2))
              (NEW (NEWT (ET (APPEND PRG-1 PRG-2))
                         (OLDT (APPEND PRG-1 PRG-2))))))
         (DEMOTE 1) (DIVE 1) (DIVE 1) (DIVE 1) (REWRITE MEMBER-APPEND)
         TOP S SPLIT (CONTRADICT 2) (REWRITE TOTAL-IMPLIES)
         (CONTRADICT 3) (REWRITE TOTAL-IMPLIES))))

(PROVE-LEMMA TOTAL-UNION (REWRITE)
             (EQUAL (TOTAL (APPEND PRG-1 PRG-2))
                    (AND (TOTAL PRG-1)
                         (TOTAL PRG-2)))
             ((ENABLE EQUAL-IFF)))


(DISABLE TOTAL-UNION-1)
(DISABLE TOTAL-UNION-2)


(PROVE-LEMMA UNLESS-UNION-1 (REWRITE)
    (IMPLIES (UNLESS P Q (APPEND PRG-1 PRG-2))
             (AND (UNLESS P Q PRG-1) (UNLESS P Q PRG-2)))
    ((INSTRUCTIONS SPLIT (REWRITE PROVE-UNLESS) PROMOTE
         (USE-LEMMA UNLESS-IMPLIES
             ((E (EU P PRG-1 Q)) (PRG (APPEND PRG-1 PRG-2))
              (OLD (OLDU P PRG-1 Q)) (NEW (NEWU P PRG-1 Q))))
         PROVE (REWRITE PROVE-UNLESS) PROMOTE
         (USE-LEMMA UNLESS-IMPLIES
             ((E (EU P PRG-2 Q)) (PRG (APPEND PRG-1 PRG-2))
              (OLD (OLDU P PRG-2 Q)) (NEW (NEWU P PRG-2 Q))))
         PROVE)))

(PROVE-LEMMA UNLESS-UNION-2 (REWRITE)
    (IMPLIES (AND (UNLESS P Q PRG-1) (UNLESS P Q PRG-2))
             (UNLESS P Q (APPEND PRG-1 PRG-2)))
    ((INSTRUCTIONS PROMOTE (REWRITE PROVE-UNLESS)
         (DIVE 1) (DIVE 1) (REWRITE MEMBER-APPEND) TOP PROMOTE
         (USE-LEMMA UNLESS-IMPLIES
             ((E (EU P (APPEND PRG-1 PRG-2) Q)) (P P) (Q Q)
              (OLD (OLDU P (APPEND PRG-1 PRG-2) Q))
              (NEW (NEWU P (APPEND PRG-1 PRG-2) Q)) (PRG PRG-1)))
         (USE-LEMMA UNLESS-IMPLIES
```

```
              ((E (EU P (APPEND PRG-1 PRG-2) Q)) (P P) (Q Q)
               (OLD (OLDU P (APPEND PRG-1 PRG-2) Q))
               (NEW (NEWU P (APPEND PRG-1 PRG-2) Q)) (PRG PRG-2)))
          PROVE)))

(PROVE-LEMMA UNLESS-UNION (REWRITE)
             (EQUAL (UNLESS P Q (APPEND PRG-1 PRG-2))
                    (AND (UNLESS P Q PRG-1)
                         (UNLESS P Q PRG-2)))
             ((ENABLE EQUAL-IFF)))

(DISABLE UNLESS-UNION-1)
(DISABLE UNLESS-UNION-2)

(PROVE-LEMMA ENSURES-UNION-1 (REWRITE)
    (IMPLIES (ENSURES P Q (APPEND PRG-1 PRG-2))
             (OR (ENSURES P Q PRG-1) (ENSURES P Q PRG-2)))
    ((INSTRUCTIONS PROMOTE
         (CLAIM (MEMBER (EE P (APPEND PRG-1 PRG-2) Q)
                        (APPEND PRG-1 PRG-2))
                0)
         (DEMOTE 2) (DIVE 1) (REWRITE MEMBER-APPEND) TOP SPLIT
         (REWRITE PROVE-ENSURES (($E (EE P (APPEND PRG-1 PRG-2) Q))))
         PROMOTE (REWRITE ENSURES-IMPLIES)
         (REWRITE PROVE-ENSURES (($E (EE P (APPEND PRG-1 PRG-2) Q))))
         PROMOTE (REWRITE ENSURES-IMPLIES) (CONTRADICT 3)
         (REWRITE PROVE-ENSURES (($E (EE P (APPEND PRG-1 PRG-2) Q))))
         PROMOTE (REWRITE ENSURES-IMPLIES) (CONTRADICT 2)
         (REWRITE ENSURES-IMPLIES))))

(PROVE-LEMMA ENSURES-UNION-2 (REWRITE)
    (IMPLIES (OR (ENSURES P Q PRG-1) (ENSURES P Q PRG-2))
             (ENSURES P Q (APPEND PRG-1 PRG-2)))
   ((INSTRUCTIONS SPLIT (REWRITE PROVE-ENSURES (($E (EE P PRG-2 Q))))
         (USE-LEMMA ENSURES-IMPLIES ((E (EE P PRG-2 Q))))
         (PROVE (ENABLE ENSURES-IMPLIES)) PROMOTE
         (REWRITE ENSURES-IMPLIES)
         (REWRITE PROVE-ENSURES (($E (EE P PRG-1 Q))))
         (PROVE (ENABLE ENSURES-IMPLIES))
         (PROVE (ENABLE ENSURES-IMPLIES)))))


(DISABLE ENSURES-UNION-1)
(DISABLE ENSURES-UNION-2)

(PROVE-LEMMA ENSURES-UNION (REWRITE)
             (EQUAL (ENSURES P Q (APPEND PRG-1 PRG-2))
                    (OR (ENSURES P Q PRG-1)
                        (ENSURES P Q PRG-2)))
             ((ENABLE EQUAL-IFF)
              (USE (ENSURES-UNION-1)
                   (ENSURES-UNION-2))))



;;; HELP PROVE TOTAL, UNLESS, AND ENSURES.


(DEFN TOTAL-SUFFICIENT (STATEMENT PROGRAM OLD NEW)
  (IMPLIES (MEMBER STATEMENT PROGRAM)
           (N OLD NEW STATEMENT)))

(PROVE-LEMMA HELP-PROVE-TOTAL (REWRITE)
             (IMPLIES (TOTAL-SUFFICIENT (ET PRG)
                                        PRG
                                        (OLDT PRG)
                                        NEW)
                      (TOTAL PRG))
             ((USE (PROVE-TOTAL))))
```

```
(DEFN UNLESS-SUFFICIENT (STATEMENT PROGRAM OLD NEW P Q)
  (IMPLIES (AND (MEMBER STATEMENT PROGRAM)
                (N OLD NEW STATEMENT)
                (EVAL P OLD)
                (NOT (EVAL Q OLD)))
           (EVAL (LIST 'OR P Q) NEW)))

(PROVE-LEMMA HELP-PROVE-UNLESS (REWRITE)
             (IMPLIES (UNLESS-SUFFICIENT (EU P PRG Q)
                                         PRG
                                         (OLDU P PRG Q)
                                         (NEWU P PRG Q)
                                         P Q)
                    (UNLESS P Q PRG))
             ((USE (PROVE-UNLESS))))

(DEFN ENSURES-KEY (STATEMENT PROGRAM OLD NEW P Q)
  (AND (MEMBER STATEMENT PROGRAM)
       (IMPLIES (AND (N OLD NEW STATEMENT)
                     (EVAL P OLD)
                     (NOT (EVAL Q OLD)))
                (EVAL Q NEW))))

(PROVE-LEMMA HELP-PROVE-ENSURES (REWRITE)
             (IMPLIES (ENSURES-KEY STATEMENT PRG
                                   (OLDE STATEMENT P Q)
                                   (NEWE STATEMENT P Q)
                                   P Q)
                      (ENSURES P Q PRG))
             ((ENABLE PROVE-ENSURES)))

(DEFN ENSURES-REST (STATEMENT KEY PROGRAM OLD NEW P Q)
  (IMPLIES (AND (MEMBER STATEMENT PROGRAM)
                (NOT (EQUAL STATEMENT KEY))
                (N OLD NEW STATEMENT)
                (EVAL P OLD)
                (NOT (EVAL Q OLD)))
           (EVAL P NEW)))

(PROVE-LEMMA HELP-PROVE-UNLESS-ENSURES (REWRITE)
             (IMPLIES (AND (ENSURES-KEY STATEMENT PRG
                                        (OLDU P PRG Q)
                                        (NEWU P PRG Q)
                                        P Q)
                           (ENSURES-REST (EU P PRG Q) STATEMENT PRG
                                         (OLDU P PRG Q) (NEWU P PRG Q)
                                         P Q))
                      (UNLESS P Q PRG))
             ((USE (HELP-PROVE-UNLESS))))


;;; STRENGTHENING AND WEAKENING UNLESS AND ENSURES

(DEFN-SK STRONGER-P (P Q)
  (FORALL STATE (IMPLIES (EVAL P STATE)
                         (EVAL Q STATE)))
  ((SUFFIX S)))

(DISABLE STRONGER-P)

(PROVE-LEMMA STRONGER-P-IMPLIES (REWRITE)
             (IMPLIES (AND (STRONGER-P P Q)
                           (EVAL P STATE))
                      (EVAL Q STATE))
             ((USE (STRONGER-P))))

(PROVE-LEMMA STRONGER-P-REWRITE (REWRITE)
             (EQUAL (STRONGER-P P Q)
                    (IMPLIES (EVAL P (STATES P Q))
```

```
                              (EVAL Q (STATES P Q))))
             ((USE (STRONGER-P (STATE (STATES P Q))))))

(DISABLE STRONGER-P-IMPLIES)

(DEFN-SK EQUAL-P (P Q)
  (FORALL STATE (EQUAL (EVAL P STATE)
                       (EVAL Q STATE)))
  ((SUFFIX E)))

(DISABLE EQUAL-P)

(PROVE-LEMMA EQUAL-P-IMPLIES (REWRITE)
             (IMPLIES (EQUAL-P P Q)
                      (EQUAL (EVAL P STATE)
                             (EVAL Q STATE)))
             ((USE (EQUAL-P))))

(DISABLE EQUAL-P-IMPLIES)

(PROVE-LEMMA EQUAL-P-REWRITE (REWRITE)
             (EQUAL (EQUAL-P P Q)
                    (EQUAL (EVAL P (STATEE P Q))
                           (EVAL Q (STATEE P Q))))
             ((USE (EQUAL-P (STATE (STATEE P Q))))))

(PROVE-LEMMA EQUAL-P-COMMUTATIVE (REWRITE)
             (EQUAL (EQUAL-P P Q)
                    (EQUAL-P Q P))
             ((USE (EQUAL-P-IMPLIES (P P) (Q Q) (STATE (STATEE Q P)))
                   (EQUAL-P-IMPLIES (P Q) (Q P) (STATE (STATEE P Q))))))

(PROVE-LEMMA ENSURES-STRENGTHEN-LEFT (REWRITE)
             (IMPLIES (AND (ENSURES Q R PRG)
                           (STRONGER-P P Q))
                      (ENSURES P R PRG))
             ((USE (PROVE-ENSURES (P P) (Q R) (PRG PRG)
                                  (E (EE Q PRG R)))
                   (ENSURES-IMPLIES (P Q) (Q R) (PRG PRG)
                                    (OLD (OLDE (EE Q PRG R) P R))
                                    (NEW (NEWE (EE Q PRG R) P R)))
                   (STRONGER-P-IMPLIES (P P) (Q Q)
                                       (STATE (OLDE (EE Q PRG R) P R))))
              (DISABLE STRONGER-P-REWRITE)))

(DISABLE ENSURES-STRENGTHEN-LEFT)

(PROVE-LEMMA ENSURES-WEAKEN-RIGHT (REWRITE)
             (IMPLIES (AND (ENSURES P Q PRG)
                           (STRONGER-P Q R))
                      (ENSURES P R PRG))
             ((USE (PROVE-ENSURES (P P) (Q R) (PRG PRG)
                                  (E (EE P PRG Q)))
                   (ENSURES-IMPLIES (P P) (Q Q) (PRG PRG)
                                    (OLD (OLDE (EE P PRG Q) P R))
                                    (NEW (NEWE (EE P PRG Q) P R)))
                   (STRONGER-P-IMPLIES (P Q) (Q R)
                                       (STATE (NEWE (EE P PRG Q) P R)))
                   (STRONGER-P-IMPLIES (P Q) (Q R)
                                       (STATE (OLDE (EE P PRG Q) P R))))
              (DISABLE STRONGER-P-REWRITE)))

(DISABLE ENSURES-WEAKEN-RIGHT)

(PROVE-LEMMA UNLESS-WEAKEN-RIGHT (REWRITE)
             (IMPLIES (AND (UNLESS P Q PRG)
                           (STRONGER-P Q R))
                      (UNLESS P R PRG))
             ((USE (PROVE-UNLESS (P P) (Q R) (PRG PRG))
```

```
                  (UNLESS-IMPLIES (P P) (Q Q) (PRG PRG)
                                  (E (EU P PRG R))
                                  (OLD (OLDU P PRG R))
                                  (NEW (NEWU P PRG R)))
                  (STRONGER-P-IMPLIES (P Q) (Q R)
                                      (STATE (OLDU P PRG R)))
                  (STRONGER-P-IMPLIES (P Q) (Q R)
                                      (STATE (NEWU P PRG R))))
             (DISABLE STRONGER-P-REWRITE)))

(DISABLE UNLESS-WEAKEN-RIGHT)

(PROVE-LEMMA UNLESS-EQUAL-P (REWRITE)
             (IMPLIES (EQUAL-P P Q)
                      (EQUAL (UNLESS P R PRG)
                             (UNLESS Q R PRG)))
             ((ENABLE PROVE-UNLESS)
              (DISABLE HELP-PROVE-UNLESS)
              (USE (UNLESS-IMPLIES (P P) (Q R)
                                   (OLD (OLDU Q PRG R))
                                   (NEW (NEWU Q PRG R))
                                   (E (EU Q PRG R)))
                   (EQUAL-P-IMPLIES (P Q) (Q P)
                                    (STATE (OLDU Q PRG R)))
                   (EQUAL-P-IMPLIES (P Q) (Q P)
                                    (STATE (NEWU Q PRG R)))
                   (UNLESS-IMPLIES (P Q) (Q R)
                                   (OLD (OLDU P PRG R))
                                   (NEW (NEWU P PRG R))
                                   (E (EU P PRG R)))
                   (EQUAL-P-IMPLIES (P P) (Q Q)
                                    (STATE (OLDU P PRG R)))
                   (EQUAL-P-IMPLIES (P P) (Q Q)
                                    (STATE (NEWU P PRG R))))))

(PROVE-LEMMA UNLESS-CONJUNCTION (REWRITE)
             (IMPLIES (AND (UNLESS P-1 Q PRG)
                           (UNLESS P-2 Q PRG)
                           (EQUAL-P P (LIST 'AND P-1 P-2)))
                      (UNLESS P Q PRG))
             ((ENABLE PROVE-UNLESS)
              (DISABLE HELP-PROVE-UNLESS)
              (USE (UNLESS-IMPLIES (P P-1) (Q Q)
                                   (OLD (OLDU P PRG Q))
                                   (NEW (NEWU P PRG Q))
                                   (E (EU P PRG Q)))
                   (UNLESS-IMPLIES (P P-2) (Q Q)
                                   (OLD (OLDU P PRG Q))
                                   (NEW (NEWU P PRG Q))
                                   (E (EU P PRG Q)))
                   (EQUAL-P-IMPLIES (P P) (Q (LIST 'AND P-1 P-2))
                                    (STATE (OLDU P PRG Q)))
                   (EQUAL-P-IMPLIES (P P) (Q (LIST 'AND P-1 P-2))
                                    (STATE (NEWU P PRG Q))))))

(PROVE-LEMMA UNLESS-DISJUNCTION (REWRITE)
             (IMPLIES (AND (UNLESS P-1 Q PRG)
                           (UNLESS P-2 Q PRG)
                           (EQUAL-P P (LIST 'OR P-1 P-2)))
                      (UNLESS P Q PRG))
             ((ENABLE PROVE-UNLESS)
              (DISABLE HELP-PROVE-UNLESS)
              (USE (UNLESS-IMPLIES (P P-1) (Q Q)
                                   (OLD (OLDU P PRG Q))
                                   (NEW (NEWU P PRG Q))
                                   (E (EU P PRG Q)))
                   (UNLESS-IMPLIES (P P-2) (Q Q)
                                   (OLD (OLDU P PRG Q))
                                   (NEW (NEWU P PRG Q))
```

```
                         (E (EU P PRG Q)))
                (EQUAL-P-IMPLIES (P P) (Q (LIST 'OR P-1 P-2))
                                 (STATE (OLDU P PRG Q)))
                (EQUAL-P-IMPLIES (P P) (Q (LIST 'OR P-1 P-2))
                                 (STATE (NEWU P PRG Q))))))


;;; FAIRNESS THEOREMS


(PROVE-LEMMA UNCONDITIONAL-FAIRNESS (REWRITE)
             (IMPLIES (AND (UNLESS P Q PRG)
                           (ENSURES P Q PRG)
                           (TOTAL PRG))
                      (LEADS-TO P Q PRG))
    ((INSTRUCTIONS PROMOTE (REWRITE ENSURES-PROVES-LEADS-TO)
         (REWRITE TOTAL-IMPLIES-SCHEDULABLE)
         (CLAIM (MEMBER (EE P PRG Q) PRG) ((ENABLE ENSURES-IMPLIES)))
         PROVE)))

(DISABLE UNCONDITIONAL-FAIRNESS)

(PROVE-LEMMA UNCONDITIONAL-FAIRNESS-GENERAL (REWRITE)
  (IMPLIES (AND (UNLESS P-1 Q-1 PRG)
                (ENSURES P-2 Q-2 PRG)
                (IMPLIES (EVAL P (S PRG (ILEADS P PRG Q)))
                         (EVAL P-1 (S PRG (ILEADS P PRG Q))))
                (STRONGER-P P-1 P-2)
                (IMPLIES (EVAL Q-1 (S PRG (JLEADS (ILEADS P-1 PRG Q)
                                                  PRG Q-1)))
                         (EVAL Q (S PRG (JLEADS (ILEADS P-1 PRG Q)
                                                PRG Q-1))))
                (STRONGER-P Q-2 Q-1)
                (TOTAL PRG))
           (LEADS-TO P Q PRG))
  ((USE (LEADS-TO-STRENGTHEN-LEFT (Q P) (R Q) (P P-1) (PRG PRG))
        (LEADS-TO-WEAKEN-RIGHT (P P-1) (R Q) (Q Q-1) (PRG PRG))
        (ENSURES-STRENGTHEN-LEFT (P P-1) (R Q-2) (Q P-2) (PRG PRG))
        (ENSURES-WEAKEN-RIGHT (P P-1) (R Q-1) (Q Q-2))
        (UNCONDITIONAL-FAIRNESS (P P-1) (Q Q-1) (PRG PRG)))))

(DISABLE UNCONDITIONAL-FAIRNESS-GENERAL)


(CONSTRAIN STRONG-FAIRNESS (REWRITE)
           (IMPLIES (AND (UNLESS P Q PRG)
                         (E-ENSURES P Q C PRG)
                         (LEADS-TO P (LIST 'OR Q C) PRG)
                         (STRONGLY-FAIR PRG))
                    (LEADS-TO P Q PRG))
           ((STRONGLY-FAIR SCHEDULABLE))
           ((USE (ENSURES-PROVES-LEADS-TO)
                 (PROVE-ENSURES (E (EEE C P PRG Q)))
                 (E-ENSURES-IMPLIES (OLD (OLDE (EEE C P PRG Q) P Q))
                                    (NEW (NEWE (EEE C P PRG Q) P Q))))))

(DISABLE STRONG-FAIRNESS)

(PROVE-LEMMA STRONG-FAIRNESS-GENERAL (REWRITE)
  (IMPLIES (AND (UNLESS P-1 Q-1 PRG)
                (E-ENSURES P-2 Q-2 C PRG)
                (LEADS-TO P (LIST 'OR Q C) PRG)
                (IMPLIES (EVAL P (S PRG (ILEADS P PRG Q)))
                         (EVAL P-1 (S PRG (ILEADS P PRG Q))))
                (IMPLIES (EVAL P-1 (S PRG
                                      (ILEADS P-1 PRG
                                              (LIST 'OR Q-1 C))))
                         (EVAL P (S PRG (ILEADS P-1 PRG
                                                (LIST 'OR Q-1 C)))))
```

```
                (STRONGER-P P-1 P-2)
                (IMPLIES (EVAL Q-1 (S PRG (JLEADS (ILEADS P-1 PRG Q)
                                                  PRG Q-1)))
                         (EVAL Q (S PRG (JLEADS (ILEADS P-1 PRG Q)
                                                PRG Q-1))))
                (IMPLIES (EVAL (LIST 'OR Q C)
                               (S PRG (JLEADS (ILEADS
                                               P PRG
                                               (LIST 'OR Q-1 C))
                                              PRG (LIST 'OR Q C))))
                         (EVAL (LIST 'OR Q-1 C)
                               (S PRG (JLEADS (ILEADS
                                               P PRG
                                               (LIST 'OR Q-1 C))
                                              PRG (LIST 'OR Q C)))))
                (STRONGER-P Q-2 Q-1)
                (STRONGLY-FAIR PRG))
           (LEADS-TO P Q PRG))
  ((USE (PROVE-E-ENSURES (E (EEE C P-2 PRG Q-2))
                         (P P-1) (Q Q-1) (PRG PRG))
        (E-ENSURES-IMPLIES (P P-2) (Q Q-2) (C C) (PRG PRG)
                           (OLD (OLDEE (EEE C P-2 PRG Q-2)
                                       P-1 Q-1))
                           (NEW (NEWEE (EEE C P-2 PRG Q-2)
                                       P-1 Q-1)))
        (LEADS-TO-STRENGTHEN-LEFT (Q P) (R Q) (P P-1) (PRG PRG))
        (LEADS-TO-STRENGTHEN-LEFT (Q P-1) (R (LIST 'OR Q-1 C))
                                  (P P) (PRG PRG))
        (LEADS-TO-WEAKEN-RIGHT (P P-1) (R Q) (Q Q-1) (PRG PRG))
        (LEADS-TO-WEAKEN-RIGHT (P P) (R (LIST 'OR Q-1 C))
                               (Q (LIST 'OR Q C)) (PRG PRG))
        (STRONGER-P-IMPLIES (P P-1) (Q P-2)
                            (STATE (OLDEE (EEE C P-2 PRG Q-2)
                                          P-1 Q-1)))
        (STRONGER-P-IMPLIES (P Q-2) (Q Q-1)
                            (STATE (OLDEE (EEE C P-2 PRG Q-2)
                                          P-1 Q-1)))
        (STRONGER-P-IMPLIES (P Q-2) (Q Q-1)
                            (STATE (NEWEE (EEE C P-2 PRG Q-2)
                                          P-1 Q-1)))
        (STRONG-FAIRNESS (P P-1) (Q Q-1) (C C) (PRG PRG)))))


(DISABLE STRONG-FAIRNESS-GENERAL)

(DEFN WFW (I J P Q C PRG)
  (IF (LESSP I J)
      (IF (EVAL Q (S PRG I))
          I
        (IF (EVAL P (S PRG I))
            (IF (EVAL C (S PRG I))
                (WFW (ADD1 I) J P Q C PRG)
              I)
          I))
    (FIX I))
  ((LESSP (DIFFERENCE J I))))

(PROVE-LEMMA WFW-BIGGER (REWRITE)
             (NOT (LESSP (WFW I J P Q C PRG) I)))

(PROVE-LEMMA ABOUT-WFW (REWRITE)
  (IMPLIES (AND (UNLESS P Q PRG)
                (LISTP PRG)
                (EVAL P (S PRG I))
                (NOT (LESSP J I)))
           (OR (EVAL Q (S PRG (WFW I J P Q C PRG)))
               (AND (EVAL P (S PRG (WFW I J P Q C PRG)))
                    (NOT (EVAL Q (S PRG (WFW I J P Q C PRG))))
                    (NOT (EVAL C (S  PRG (WFW I J P Q C PRG)))))
```

```
               (AND (EQUAL (WFW I J P Q C PRG) (FIX J))
                    (OR (EVAL P (S PRG J))
                        (EVAL Q (S PRG J))))))
  ((INSTRUCTIONS
    (INDUCT (WFW I J P Q C PRG)) PROVE
    (CLAIM (EVAL Q (S PRG (ADD1 I))) 0) PROVE
    (CLAIM (EVAL (LIST 'OR P Q) (S PRG (ADD1 I))) 0) PROVE PROMOTE
    PROMOTE (CONTRADICT 2)
    (USE-LEMMA EFFECTIVE-IDLE ((PRG PRG) (I I)))
    (USE-LEMMA UNLESS-IMPLIES
               ((PRG PRG) (OLD (S PRG I)) (NEW (S PRG (ADD1 I)))
                (E (CHOOSE PRG I))))
    PROVE PROVE PROVE PROVE)))

(DEFN WITNESS (P Q C PRG)
  (WFW (ILEADS P PRG Q)
       (NEXT PRG (EEE C P PRG Q)
             (ILEADS P PRG Q))
       P Q C PRG))

(PROVE-LEMMA WEAK-FAIRNESS (REWRITE)
             (IMPLIES (AND (UNLESS P Q PRG)
                           (E-ENSURES P Q C PRG)
                           (IMPLIES (EVAL (LIST 'AND P (LIST 'NOT Q))
                                          (S PRG (WITNESS P Q C PRG)))
                                    (EVAL C (S PRG (WITNESS P Q C PRG)))))
                      (LEADS-TO P Q PRG))
    ((INSTRUCTIONS
      PROMOTE
      (CLAIM (MEMBER (EEE C P PRG Q) PRG)
             ((ENABLE E-ENSURES-IMPLIES)))
      (CLAIM (LISTP PRG))
      (USE-LEMMA ABOUT-WFW
                 ((P P) (Q Q) (C C) (PRG PRG) (I (ILEADS P PRG Q))
                  (J (NEXT PRG (EEE C P PRG Q) (ILEADS P PRG Q)))))
      (REWRITE PROVE-LEADS-TO
               (($J (IF (EVAL Q (S PRG (WITNESS P Q C PRG)))
                        (WITNESS P Q C PRG)
                        (ADD1 (WITNESS P Q C PRG))))))
      PROMOTE (DIVE 1) (= T) UP (DEMOTE 6) (DIVE 1) (DIVE 1) (= T)
      TOP PROMOTE S-PROP (DIVE 3) (DIVE 1)
      (REWRITE E-ENSURES-IMPLIES
               (($P P) ($C C) ($PRG PRG)
                ($OLD (S PRG (WITNESS P Q C PRG)))))
      TOP S (DIVE 3) (= (CHOOSE PRG (WITNESS P Q C PRG))) UP
      (REWRITE COMPUTATION
               (($NEW (NEWC-1 (EEE C P PRG Q)
                              (S PRG (WITNESS P Q C PRG))))))
      (DIVE 3) (= (EEE C P PRG Q)) UP
      (REWRITE ENABLING-CONDITION-IMPLIES (($C C) ($PRG PRG)))
      (REWRITE E-ENSURES-IMPLIES) PROVE PROVE)))

(DISABLE WEAK-FAIRNESS)

(PROVE-LEMMA WEAK-FAIRNESS-GENERAL (REWRITE)
  (IMPLIES (AND (UNLESS P-1 Q-1 PRG)
                (E-ENSURES P-2 Q-2 C PRG)
                (IMPLIES (EVAL (LIST 'AND P-1 (LIST 'NOT Q-1))
                               (S PRG (WITNESS P-1 Q-1 C PRG)))
                         (EVAL C (S PRG (WITNESS P-1 Q-1 C PRG))))
                (IMPLIES (EVAL P (S PRG (ILEADS P PRG Q)))
                         (EVAL P-1 (S PRG (ILEADS P PRG Q))))
                (STRONGER-P P-1 P-2)
                (IMPLIES (EVAL Q-1 (S PRG (JLEADS (ILEADS P-1 PRG Q)
                                                  PRG Q-1)))
                         (EVAL Q (S PRG (JLEADS (ILEADS P-1 PRG Q)
                                                PRG Q-1))))
                (STRONGER-P Q-2 Q-1))
           (LEADS-TO P Q PRG))
```

```
   ((USE (PROVE-E-ENSURES (E (EEE C P-2 PRG Q-2))
                          (P P-1) (Q Q-1) (PRG PRG))
         (E-ENSURES-IMPLIES (P P-2) (Q Q-2) (C C) (PRG PRG)
                            (OLD (OLDEE (EEE C P-2 PRG Q-2)
                                        P-1 Q-1))
                            (NEW (NEWEE (EEE C P-2 PRG Q-2)
                                        P-1 Q-1)))
         (LEADS-TO-STRENGTHEN-LEFT (Q P) (R Q) (P P-1) (PRG PRG))
         (LEADS-TO-WEAKEN-RIGHT (P P-1) (R Q) (Q Q-1) (PRG PRG))
         (STRONGER-P-IMPLIES (P P-1) (Q P-2)
                             (STATE (OLDEE (EEE C P-2 PRG Q-2)
                                           P-1 Q-1)))
         (STRONGER-P-IMPLIES (P Q-2) (Q Q-1)
                             (STATE (OLDEE (EEE C P-2 PRG Q-2)
                                           P-1 Q-1)))
         (STRONGER-P-IMPLIES (P Q-2) (Q Q-1)
                             (STATE (NEWEE (EEE C P-2 PRG Q-2)
                                           P-1 Q-1)))
         (WEAK-FAIRNESS (P P-1) (Q Q-1) (C C) (PRG PRG)))))

(DISABLE WEAK-FAIRNESS-GENERAL)
(DISABLE WITNESS)


(PROVE-LEMMA DEADLOCK-FREEDOM-WITNESS (REWRITE)
  (IMPLIES (AND (UNLESS INV '(FALSE) PRG)
                (ENABLING-CONDITION C E PRG)
                (IMPLIES (EVAL INV
                               (S PRG
                                  (NEXT PRG E
                                        (ILEADS INV PRG
                                                '(FALSE)))))
                         (NOT (EVAL C
                                    (S PRG
                                       (NEXT PRG E
                                             (ILEADS INV PRG
                                                     '(FALSE)))))))
                (SCHEDULABLE PRG))
           (LEADS-TO INV '(FALSE) PRG))
    ((INSTRUCTIONS PROMOTE
         (CLAIM (MEMBER E PRG) ((ENABLE ENABLING-CONDITION-IMPLIES)))
         (REWRITE PROVE-LEADS-TO) (CONTRADICT 3) (DIVE 1) (DIVE 1)
         (REWRITE INVARIANTS-PERSIST-GENERAL
                  (($I (ILEADS INV PRG '(FALSE)))))
         UP (DIVE 2) (DIVE 1)
         (REWRITE ENABLING-CONDITION-IMPLIES
                  (($E E) ($PRG PRG)
                   ($NEW (S PRG
                            (ADD1 (NEXT PRG E
                                        (ILEADS INV PRG '(FALSE))))))))
         TOP S (REWRITE COMPUTATION-N) PROVE PROVE PROVE)))

(DISABLE DEADLOCK-FREEDOM-WITNESS)


(CONSTRAIN DEADLOCK-FREEDOM (REWRITE)
  (IMPLIES (AND (UNLESS INV '(FALSE) PRG)
                (ENABLING-CONDITION C E PRG)
                (IMPLIES (EVAL INV
                               (S PRG
                                  (NEXT PRG E
                                        (ILEADS INV PRG
                                                '(FALSE)))))
                         (NOT (EVAL C
                                    (S PRG
                                       (NEXT PRG E
                                             (ILEADS INV PRG
                                                     '(FALSE)))))))
                (DEADLOCK-FREE PRG))
```

```
           (LEADS-TO INV '(FALSE) PRG))
  ((DEADLOCK-FREE SCHEDULABLE))
  ((USE (DEADLOCK-FREEDOM-WITNESS))))

(PROVE-LEMMA DEADLOCK-FREEDOM-GENERAL (REWRITE)
  (IMPLIES (AND (UNLESS INV '(FALSE) PRG)
                (ENABLING-CONDITION C E PRG)
                (IMPLIES (EVAL INV
                               (S PRG
                                  (NEXT PRG E
                                        (ILEADS INV PRG
                                                '(FALSE)))))
                         (NOT (EVAL C
                                    (S PRG
                                       (NEXT PRG E
                                             (ILEADS INV PRG
                                                     '(FALSE)))))))
                (IMPLIES (NOT (EVAL INV (S PRG (II P PRG))))
                         (EVAL P (S PRG (II P PRG))))
                (DEADLOCK-FREE PRG))
           (INVARIANT P PRG))
   ((INSTRUCTIONS PROMOTE
         (REWRITE INVARIANT-CONSEQUENCE (($P (LIST 'NOT INV))))
         (REWRITE LEADS-TO-FALSE-INVARIANT (($P INV)))
         (REWRITE DEADLOCK-FREEDOM) S (DROP 1 2 3 5) PROVE)))

(ENABLE EVAL)
(ENABLE N)

;;; SOME HELPFUL DEFINITIONS AND THEOREMS

(DEFN UPDATE-ASSOC (KEY VALUE ALIST)
  (IF (LISTP ALIST)
      (IF (EQUAL (CAAR ALIST) KEY)
          (CONS (CONS KEY VALUE) (CDR ALIST))
          (CONS (CAR ALIST)
                (UPDATE-ASSOC KEY VALUE (CDR ALIST))))
      (LIST (CONS KEY VALUE))))

(PROVE-LEMMA SIMPLIFY-ASSOC (REWRITE)
             (EQUAL (ASSOC KEY-1 (UPDATE-ASSOC KEY-2 VALUE ALIST))
                    (IF (EQUAL KEY-1 KEY-2)
                        (CONS KEY-1 VALUE)
                        (ASSOC KEY-1 ALIST))))

(DEFN ADD1-MOD (N X)
  (IF (LESSP (ADD1 X) N)
      (ADD1 X)
    0))

(DEFN SUB1-MOD (N X)
  (IF (LESSP X N)
      (IF (ZEROP X)
          (SUB1 N)
        (SUB1 X))
      0))

(DEFN UC (OLD NEW KEYS EXCPT)
  (IF (LISTP KEYS)
      (IF (MEMBER (CAR KEYS) EXCPT)
          (UC OLD NEW (CDR KEYS) EXCPT)
        (IF (EQUAL (ASSOC (CAR KEYS) OLD)
                   (ASSOC (CAR KEYS) NEW))
            (UC OLD NEW (CDR KEYS) EXCPT)
          F))
    T))


(PROVE-LEMMA UC-BASIC-PROPERTY (REWRITE)
```

```
             (IMPLIES (AND (UC OLD NEW KEYS EXCPT)
                           (MEMBER KEY KEYS)
                           (NOT (MEMBER KEY EXCPT)))
                      (EQUAL (EQUAL (ASSOC KEY OLD)
                                    (ASSOC KEY NEW))
                             T)))

(PROVE-LEMMA UC-COMMUTATIVE (REWRITE)
             (EQUAL (UC OLD NEW KEYS EXCPT)
                    (UC NEW OLD KEYS EXCPT)))

(PROVE-LEMMA UC-REFLEXIVE (REWRITE)
             (UC LIST LIST KEYS EXCPT))

(PROVE-LEMMA UC-OF-UPDATE-ASSOC (REWRITE)
             (EQUAL (UC LIST-1
                        (UPDATE-ASSOC KEY VALUE LIST-2)
                        KEYS EXCPT)
                    (IF (MEMBER KEY EXCPT)
                        (UC LIST-1 LIST-2 KEYS EXCPT)
                      (IF (MEMBER KEY KEYS)
                          (AND (EQUAL (ASSOC KEY LIST-1)
                                      (CONS KEY VALUE))
                               (UC LIST-1 LIST-2 KEYS (CONS KEY EXCPT)))
                        (UC LIST-1 LIST-2 KEYS EXCPT)))))


(PROVE-LEMMA STRIP-CARS-APPEND (REWRITE)
             (EQUAL (STRIP-CARS (APPEND A B))
                    (APPEND (STRIP-CARS A)
                            (STRIP-CARS B))))

(PROVE-LEMMA UC-APPEND (REWRITE)
             (EQUAL (UC OLD NEW (APPEND A B) EXCPT)
                    (AND (UC OLD NEW A EXCPT)
                         (UC OLD NEW B EXCPT))))

(PROVE-LEMMA UC-COMMUTATIVE-2 (REWRITE)
             (EQUAL (UC OLD NEW (APPEND A B) EXCPT)
                    (UC OLD NEW (APPEND B A) EXCPT)))

(DISABLE UC-APPEND)

(PROVE-LEMMA KEY-NOT-MEMBER-STRIP-CARS (REWRITE)
             (IMPLIES (NOT (MEMBER KEY (STRIP-CARS ALIST)))
                      (EQUAL (ASSOC KEY ALIST)
                             F)))

(PROVE-LEMMA UC-PROPERTY (REWRITE)
  (IMPLIES (AND (UC OLD NEW (APPEND (STRIP-CARS OLD)
                                    (STRIP-CARS NEW))
                    EXCPT)
                (NOT (MEMBER KEY EXCPT)))
           (EQUAL (EQUAL (ASSOC KEY OLD)
                         (ASSOC KEY NEW))
                  T))
  ((DISABLE UC-BASIC-PROPERTY)
   (USE (UC-BASIC-PROPERTY (KEYS (STRIP-CARS (APPEND OLD NEW)))))))

(PROVE-LEMMA ABOUT-UC (REWRITE)
             (IMPLIES (AND (UC A B (APPEND (STRIP-CARS A)
                                           (STRIP-CARS B))
                               EXCPT)
                           (NOT (MEMBER KEY EXCPT)))
                      (EQUAL (ASSOC KEY A)
                             (ASSOC KEY B)))
             ((DISABLE UC-PROPERTY)
              (USE (UC-PROPERTY (OLD A) (NEW B)))))
```

```
(DEFN CHANGED (OLD NEW EXCPT)
  (UC OLD NEW (STRIP-CARS (APPEND OLD NEW)) EXCPT))


))
```

Appendix C
Mutual Exclusion Events

This appendix contains the complete events list supporting the proof of mutual exclusion described in chapter 4.

This event list constructs the proof of mutual exclusion on top of the library created by the events listed in Appendix B.

```
(NOTE-LIB "INTERPRETER")

(PROVEALL "ME" '(


;;;THIS FILES DEFINES A TOKEN PASSING SOLUTION TO THE N PROCESSOR
;;;MUTUAL EXCLUSION PROBLEM.

(DEFN STATUS (STATE INDEX)
  (CDR (ASSOC (CONS 'ME INDEX) STATE)))

(DEFN WAIT (STATE INDEX)
  (EQUAL (STATUS STATE INDEX) 'WAIT))

(DEFN NON-CRITICAL (STATE INDEX)
  (EQUAL (STATUS STATE INDEX) 'NON-CRITICAL))

(DEFN CRITICAL (STATE INDEX)
  (AND (NOT (NON-CRITICAL STATE INDEX))
       (NOT (WAIT STATE INDEX))))

(DEFN TICKS (STATE INDEX)
  (FIX (STATUS STATE INDEX)))

(DEFN CRITICAL-TICKS (STATE INDEX TICKS)
  (AND (CRITICAL STATE INDEX)
       (EQUAL (TICKS STATE INDEX)
              (FIX TICKS))))

(DEFN CHANNEL (STATE INDEX)
  (CDR (ASSOC (CONS 'C INDEX) STATE)))

(DEFN TOKEN (STATE INDEX)
  (LISTP (CHANNEL STATE INDEX)))

(DEFN ME (OLD NEW INDEX SIZE)
  (IF (NON-CRITICAL OLD INDEX)
      (IF (NON-CRITICAL NEW INDEX)
          (IF (TOKEN OLD INDEX)
              (IF (EQUAL (ADD1-MOD SIZE INDEX) INDEX)
                  (AND (EQUAL (LENGTH (CHANNEL NEW INDEX))
                              (LENGTH (CHANNEL OLD INDEX)))
                       (CHANGED OLD NEW
                                (LIST (CONS 'C INDEX))))
                (AND (EQUAL (CHANNEL NEW INDEX)
                            (CDR (CHANNEL OLD INDEX)))
                     (EQUAL (LENGTH (CHANNEL NEW (ADD1-MOD SIZE INDEX)))
                            (ADD1 (LENGTH (CHANNEL OLD
                                                   (ADD1-MOD SIZE
                                                             INDEX)))))
                     (CHANGED OLD NEW
                              (LIST (CONS 'C INDEX)
                                    (CONS 'C (ADD1-MOD SIZE INDEX))))))
            (CHANGED OLD NEW NIL))
        (AND (WAIT NEW INDEX)
```

```
             (CHANGED OLD NEW (LIST (CONS 'ME INDEX)))))
    (IF (WAIT OLD INDEX)
        (IF (TOKEN OLD INDEX)
            (AND (EQUAL (CHANNEL NEW INDEX)
                        (CDR (CHANNEL OLD INDEX)))
                 (CRITICAL NEW INDEX)
                 (CHANGED OLD NEW (LIST (CONS 'ME INDEX)
                                        (CONS 'C INDEX))))
          (CHANGED OLD NEW NIL))
      (OR (AND (LESSP (TICKS NEW INDEX) (TICKS OLD INDEX))
               (CRITICAL NEW INDEX)
               (CHANGED OLD NEW (LIST (CONS 'ME INDEX))))
          (AND (NON-CRITICAL NEW INDEX)
               (EQUAL (LENGTH (CHANNEL NEW (ADD1-MOD SIZE INDEX)))
                      (ADD1 (LENGTH (CHANNEL OLD (ADD1-MOD SIZE INDEX)))))
               (CHANGED OLD NEW
                        (LIST (CONS 'C (ADD1-MOD SIZE INDEX))
                              (CONS 'ME INDEX))))))))

(DEFN ME-FUNCTION (INDEX SIZE STATE)
  (IF (NON-CRITICAL STATE INDEX)
      (IF (TOKEN STATE INDEX)
          (UPDATE-ASSOC (CONS 'C (ADD1-MOD SIZE INDEX))
                        (CONS 'TOKEN
                              (CHANNEL (UPDATE-ASSOC
                                        (CONS 'C INDEX)
                                        (CDR (CHANNEL STATE INDEX))
                                        STATE)
                                       (ADD1-MOD SIZE INDEX)))
                        (UPDATE-ASSOC (CONS 'C INDEX)
                                      (CDR (CHANNEL STATE INDEX))
                                      STATE))
          STATE)
      (IF (WAIT STATE INDEX)
          (IF (TOKEN STATE INDEX)
              (UPDATE-ASSOC (CONS 'C INDEX)
                            (CDR (CHANNEL STATE INDEX))
                            (UPDATE-ASSOC (CONS 'ME INDEX)
                                          0
                                          STATE))
            STATE)
        (IF (ZEROP (TICKS STATE INDEX))
            (UPDATE-ASSOC (CONS 'C (ADD1-MOD SIZE INDEX))
                          (CONS 'TOKEN (CHANNEL STATE (ADD1-MOD SIZE
                                                                INDEX)))
                          (UPDATE-ASSOC (CONS 'ME INDEX)
                                        'NON-CRITICAL
                                        STATE))
          (UPDATE-ASSOC (CONS 'ME INDEX)
                        (SUB1 (TICKS STATE INDEX))
                        STATE)))))

(PROVE-LEMMA TOTAL-ME (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE))
                      (ME OLD (ME-FUNCTION INDEX SIZE OLD) INDEX SIZE)))

(DEFN PROGRAM (INDEX SIZE)
  (IF (ZEROP INDEX)
      NIL
    (CONS (LIST 'ME (SUB1 INDEX) SIZE)
          (PROGRAM (SUB1 INDEX) SIZE))))

(DEFN ME-PRG (SIZE)
  (PROGRAM SIZE SIZE))

(PROVE-LEMMA MEMBER-PROGRAM (REWRITE)
             (EQUAL (MEMBER STATEMENT (PROGRAM INDEX SIZE))
                    (IF (ZEROP INDEX)
```

```
                        F
                      (AND (EQUAL (CAR STATEMENT) 'ME)
                           (NUMBERP (CADR STATEMENT))
                           (LESSP (CADR STATEMENT) INDEX)
                           (EQUAL (CADDR STATEMENT) SIZE)
                           (EQUAL (CDDDR STATEMENT) NIL))))
             ((EXPAND (PROGRAM 1 SIZE)
                      (PROGRAM 0 SIZE))))

(PROVE-LEMMA MEMBER-ME-PRG (REWRITE)
             (EQUAL (MEMBER STATEMENT (ME-PRG SIZE))
                    (IF (ZEROP SIZE)
                        F
                      (AND (EQUAL (CAR STATEMENT) 'ME)
                           (NUMBERP (CADR STATEMENT))
                           (LESSP (CADR STATEMENT) SIZE)
                           (EQUAL (CADDR STATEMENT) SIZE)
                           (EQUAL (CDDDR STATEMENT) NIL)))))

(DISABLE ME-PRG)

(PROVE-LEMMA TOTAL-PRG (REWRITE)
   (TOTAL (ME-PRG SIZE))
   ((DISABLE PROVE-TOTAL ME-FUNCTION ME)
    (USE (PROVE-TOTAL (PRG (ME-PRG SIZE))
                      (NEW (ME-FUNCTION (CADR (ET (ME-PRG SIZE)))
                                        (CADDR (ET (ME-PRG SIZE)))
                                        (OLDT (ME-PRG SIZE)))))))))

(DEFN WEIGHT-OF-TRIPLE (STATE INDEX SIZE)
  (PLUS (IF (CRITICAL STATE INDEX)
            1 0)
        (LENGTH (CHANNEL STATE INDEX))
        (LENGTH (CHANNEL STATE (ADD1-MOD SIZE INDEX)))))


(PROVE-LEMMA WEIGHT-OF-TRIPLE-PRESERVED (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP 1 SIZE)
                           (LESSP INDEX SIZE)
                           (ME OLD NEW INDEX SIZE))
                      (EQUAL (WEIGHT-OF-TRIPLE NEW INDEX SIZE)
                             (WEIGHT-OF-TRIPLE OLD INDEX SIZE))))


(DEFN WEIGHT-BUT-TRIPLE (STATE INDEX SIZE N)
  (IF (ZEROP N)
      0
    (IF (EQUAL (SUB1 N) INDEX)
        (WEIGHT-BUT-TRIPLE STATE INDEX SIZE (SUB1 N))
      (PLUS (IF (CRITICAL STATE (SUB1 N))
                1 0)
            (IF (EQUAL (SUB1 N) (ADD1-MOD SIZE INDEX))
                0
              (LENGTH (CHANNEL STATE (SUB1 N))))
            (WEIGHT-BUT-TRIPLE STATE INDEX SIZE (SUB1 N))))))

(PROVE-LEMMA WEIGHT-BUT-TRIPLE-PRESERVED (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE)
                           (ME OLD NEW INDEX SIZE))
                      (EQUAL (WEIGHT-BUT-TRIPLE NEW INDEX SIZE N)
                             (WEIGHT-BUT-TRIPLE OLD INDEX SIZE N))))

(DEFN WEIGHT (STATE SIZE)
  (IF (ZEROP SIZE)
      0
    (PLUS (IF (CRITICAL STATE (SUB1 SIZE))
              1 0)
```

```
             (LENGTH (CHANNEL STATE (SUB1 SIZE)))
             (WEIGHT STATE (SUB1 SIZE)))))

(PROVE-LEMMA WEIGHT-IS-SUM (REWRITE)
             (IMPLIES
              (AND (NUMBERP INDEX)
                   (LESSP INDEX SIZE)
                   (LESSP 1 SIZE))
              (EQUAL (WEIGHT STATE N)
                     (PLUS (IF (LESSP INDEX N)
                               (PLUS (IF (CRITICAL STATE INDEX)
                                         1 0)
                                     (LENGTH (CHANNEL STATE INDEX)))
                             0)
                           (IF (LESSP (ADD1-MOD SIZE INDEX) N)
                               (LENGTH (CHANNEL STATE (ADD1-MOD SIZE
                                                       INDEX)))
                             0)
                           (WEIGHT-BUT-TRIPLE STATE INDEX SIZE N)))))

(PROVE-LEMMA WEIGHT-PRESERVED-1 (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE)
                           (LESSP 1 SIZE)
                           (ME OLD NEW INDEX SIZE))
                      (EQUAL (WEIGHT NEW SIZE)
                             (WEIGHT OLD SIZE)))
             ((DISABLE ME WEIGHT-OF-TRIPLE-PRESERVED)
              (USE (WEIGHT-OF-TRIPLE-PRESERVED))))

(PROVE-LEMMA WEIGHT-PRESERVED-2 (REWRITE)
             (IMPLIES (ME OLD NEW 0 1)
                      (EQUAL (WEIGHT NEW 1)
                             (WEIGHT OLD 1))))

(PROVE-LEMMA ABOUT-SIZE-INDEX ()
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE))
                      (OR (AND (EQUAL INDEX 0)
                               (EQUAL SIZE 1))
                          (LESSP 1 SIZE))))

(PROVE-LEMMA WEIGHT-PRESERVED (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE)
                           (ME OLD NEW INDEX SIZE))
                      (EQUAL (WEIGHT NEW SIZE)
                             (WEIGHT OLD SIZE)))
             ((DISABLE WEIGHT-IS-SUM ME)
              (USE (ABOUT-SIZE-INDEX))))

(DISABLE WEIGHT-IS-SUM)
(DISABLE WEIGHT-PRESERVED-1)
(DISABLE WEIGHT-PRESERVED-2)
(DISABLE WEIGHT-OF-TRIPLE-PRESERVED)
(DISABLE WEIGHT-BUT-TRIPLE-PRESERVED)
(DISABLE TOTAL-ME)

(DEFN MUTUAL-EXCLUSIONP (STATE SIZE)
  (EQUAL (WEIGHT STATE SIZE) 1))


(PROVE-LEMMA MUTUAL-EXCLUSIONP-UNLESS-SUFFICIENT (REWRITE)
             (UNLESS-SUFFICIENT STATEMENT
                                (ME-PRG SIZE)
                                OLD NEW
                                ‘(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                                ’(FALSE))
             ((DISABLE ME)))
```

```
(DISABLE WEIGHT-PRESERVED)

(PROVE-LEMMA MUTUAL-EXCLUSIONP-PRG (REWRITE)
             (UNLESS '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                     '(FALSE)
                     (ME-PRG SIZE))
             ((DISABLE ME-PRG UNLESS-SUFFICIENT)))

(PROVE-LEMMA WAIT-UNLESS-CRITICAL-1 ()
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE)
                           (ME OLD NEW INDEX SIZE)
                           (WAIT OLD INDEX))
                      (OR (WAIT NEW INDEX)
                          (CRITICAL NEW INDEX))))

(PROVE-LEMMA WAIT-UNLESS-CRITICAL-2 ()
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE)
                           (NUMBERP INDEX1)
                           (LESSP INDEX1 SIZE)
                           (NOT (EQUAL INDEX INDEX1))
                           (ME OLD NEW INDEX1 SIZE)
                           (WAIT OLD INDEX))
                      (OR (WAIT NEW INDEX)
                          (CRITICAL NEW INDEX))))

(PROVE-LEMMA WAIT-UNLESS-CRITICAL-UNLESS-SUFFICIENT (REWRITE)
   (IMPLIES (AND (NUMBERP INDEX)
                 (LESSP INDEX SIZE))
            (UNLESS-SUFFICIENT STATEMENT
                               (ME-PRG SIZE)
                               OLD
                               NEW
                               '(WAIT STATE (QUOTE ,INDEX))
                               '(CRITICAL STATE (QUOTE ,INDEX))))
   ((DISABLE ME)
    (USE (WAIT-UNLESS-CRITICAL-1)
         (WAIT-UNLESS-CRITICAL-2 (INDEX1 (CADR STATEMENT))))))

(PROVE-LEMMA WAIT-UNLESS-CRITICAL (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE)
                           (NOT (ZEROP SIZE)))
                      (UNLESS '(WAIT STATE (QUOTE ,INDEX))
                              '(CRITICAL STATE (QUOTE ,INDEX))
                              (ME-PRG SIZE)))
             ((DISABLE ME
                       WAIT CRITICAL
                       UNLESS-SUFFICIENT)))

(PROVE-LEMMA CELCOR-ENSURES-KEY (REWRITE)
   (IMPLIES (AND (NUMBERP INDEX)
                 (LESSP INDEX SIZE))
            (ENSURES-KEY (LIST 'ME INDEX SIZE)
                         (ME-PRG SIZE)
                         OLD NEW
                         '(AND (CRITICAL STATE (QUOTE ,INDEX))
                               (LESSP (TICKS STATE (QUOTE ,INDEX))
                                      (QUOTE ,(ADD1 TICKS))))
                         '(OR (AND (CRITICAL STATE (QUOTE ,INDEX))
                                   (LESSP (TICKS STATE
                                                 (QUOTE ,INDEX))
                                          (QUOTE ,TICKS)))
                              (TOKEN STATE
                                     (QUOTE ,(ADD1-MOD SIZE
                                                       INDEX)))))))

(PROVE-LEMMA CELCOR-ENSURES-REST (REWRITE)
```

```
   (IMPLIES (AND (NUMBERP INDEX)
                 (LESSP INDEX SIZE))
            (ENSURES-REST STATEMENT
                          (LIST 'ME INDEX SIZE)
                          (ME-PRG SIZE)
                          OLD NEW
                          `(AND (CRITICAL STATE (QUOTE ,INDEX))
                                (LESSP (TICKS STATE (QUOTE ,INDEX))
                                       (QUOTE ,(ADD1 TICKS))))
                          Q)))

(PROVE-LEMMA CRITICAL-ENSURES-LESS-CRITICAL-OR-RIGHT (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (ENSURES `(AND (CRITICAL STATE (QUOTE ,INDEX))
                            (LESSP (TICKS STATE (QUOTE ,INDEX))
                                   (QUOTE ,(ADD1 TICKS))))
                      `(OR (AND (CRITICAL STATE (QUOTE ,INDEX))
                                (LESSP (TICKS STATE (QUOTE ,INDEX))
                                       (QUOTE ,TICKS)))
                           (TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX))))
                      (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE
                   (REWRITE HELP-PROVE-ENSURES
                            (($STATEMENT (LIST 'ME INDEX SIZE))))
                   (REWRITE CELCOR-ENSURES-KEY))))

(PROVE-LEMMA CRITICAL-UNLESS-LESS-CRITICAL-OR-RIGHT (REWRITE)
   (IMPLIES
    (AND (NUMBERP INDEX)
         (LESSP INDEX SIZE))
    (UNLESS `(AND (CRITICAL STATE (QUOTE ,INDEX))
                  (LESSP (TICKS STATE (QUOTE ,INDEX))
                         (QUOTE ,(ADD1 TICKS))))
             `(OR (AND (CRITICAL STATE (QUOTE ,INDEX))
                       (LESSP (TICKS STATE (QUOTE ,INDEX))
                              (QUOTE ,TICKS)))
                  (TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX))))
             (ME-PRG SIZE)))
   ((INSTRUCTIONS
     PROMOTE
     (REWRITE HELP-PROVE-UNLESS-ENSURES
              (($STATEMENT (LIST 'ME INDEX SIZE))))
     (REWRITE CELCOR-ENSURES-KEY) (REWRITE CELCOR-ENSURES-REST))))

(PROVE-LEMMA WLCEC-ENSURES-KEY (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE))
                      (ENSURES-KEY (LIST 'ME INDEX SIZE)
                                   (ME-PRG SIZE)
                                   OLD NEW
                                   `(AND (WAIT STATE (QUOTE ,INDEX))
                                         (TOKEN STATE (QUOTE ,INDEX)))
                                   `(CRITICAL STATE (QUOTE ,INDEX)))))

(PROVE-LEMMA WLCEC-ENSURES-REST (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
                      (ENSURES-REST STATEMENT (LIST 'ME INDEX SIZE)
                                    (ME-PRG SIZE) OLD NEW
                                    `(AND (WAIT STATE (QUOTE ,INDEX))
                                          (TOKEN STATE (QUOTE ,INDEX)))
                                    Q))
             ((INSTRUCTIONS
               (CLAIM (EQUAL (ADD1-MOD SIZE (CADR STATEMENT)) INDEX) 0)
               PROVE PROVE)))


(PROVE-LEMMA WAIT-AND-LEFT-CHANNEL-ENSURES-CRITIAL (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (ENSURES `(AND (WAIT STATE (QUOTE ,INDEX))
```

```
                           (TOKEN STATE (QUOTE ,INDEX)))
                     '(CRITICAL STATE (QUOTE ,INDEX))
                     (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE HELP-PROVE-ENSURES (($STATEMENT (LIST 'ME INDEX SIZE))))
         (REWRITE WLCEC-ENSURES-KEY))))

(PROVE-LEMMA WAIT-AND-LEFT-CHANNEL-UNLESS-CRITIAL (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (UNLESS '(AND (WAIT STATE (QUOTE ,INDEX))
                          (TOKEN STATE (QUOTE ,INDEX)))
                    '(CRITICAL STATE (QUOTE ,INDEX))
                    (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE HELP-PROVE-UNLESS-ENSURES
                  (($STATEMENT (LIST 'ME INDEX SIZE))))
         (REWRITE WLCEC-ENSURES-KEY) (REWRITE WLCEC-ENSURES-REST))))

(PROVE-LEMMA NCLCEWLCORC-ENSURES-KEY (REWRITE)
  (IMPLIES (AND (NUMBERP INDEX)
                (LESSP INDEX SIZE))
           (ENSURES-KEY (LIST 'ME INDEX SIZE)
                        (ME-PRG SIZE)
                        OLD NEW
                        '(AND (NON-CRITICAL STATE (QUOTE ,INDEX))
                              (TOKEN STATE (QUOTE ,INDEX)))
                        '(OR (AND (WAIT STATE (QUOTE ,INDEX))
                                  (TOKEN STATE (QUOTE ,INDEX)))
                             (TOKEN STATE
                                    (QUOTE ,(ADD1-MOD SIZE
                                                      INDEX)))))))

(PROVE-LEMMA NCLCEWLCORC-ENSURES-REST (REWRITE)
   (IMPLIES (AND (NUMBERP INDEX)
                 (LESSP INDEX SIZE))
            (ENSURES-REST STATEMENT
                          (LIST 'ME INDEX SIZE)
                          (ME-PRG SIZE)
                          OLD NEW
                          '(AND (NON-CRITICAL STATE (QUOTE ,INDEX))
                                (TOKEN STATE (QUOTE ,INDEX)))
                          Q))
   ((INSTRUCTIONS
     (CLAIM (EQUAL (ADD1-MOD SIZE (CADR STATEMENT)) INDEX) 0)
     PROVE PROVE)))


(PROVE-LEMMA NON-CRITICAL-LEFT-ENSURES-WAIT-LEFT-OR-RIGHT (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (ENSURES '(AND (NON-CRITICAL STATE (QUOTE ,INDEX))
                           (TOKEN STATE (QUOTE ,INDEX)))
                     '(OR (AND (WAIT STATE (QUOTE ,INDEX))
                               (TOKEN STATE (QUOTE ,INDEX)))
                          (TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX))))
                     (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE HELP-PROVE-ENSURES (($STATEMENT (LIST 'ME INDEX SIZE))))
         (REWRITE NCLCEWLCORC-ENSURES-KEY))))

(PROVE-LEMMA NON-CRITICAL-LEFT-UNLESS-WAIT-LEFT-OR-RIGHT (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (UNLESS '(AND (NON-CRITICAL STATE (QUOTE ,INDEX))
                          (TOKEN STATE (QUOTE ,INDEX)))
                    '(OR (AND (WAIT STATE (QUOTE ,INDEX))
                              (TOKEN STATE (QUOTE ,INDEX)))
                         (TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX))))
                    (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE HELP-PROVE-UNLESS-ENSURES
```

```
                  (($STATEMENT (LIST 'ME INDEX SIZE))))
         (REWRITE NCLCEWLCORC-ENSURES-KEY)
         (REWRITE NCLCEWLCORC-ENSURES-REST))))

(PROVE-LEMMA LISTP-ME-PRG (REWRITE)
             (IMPLIES (NOT (ZEROP SIZE))
                      (LISTP (ME-PRG SIZE)))
             ((ENABLE ME-PRG)))

(PROVE-LEMMA MUTUAL-EXCLUSIONP-IS-INVARIANT (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                            '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                            (ME-PRG SIZE))
                           (NOT (ZEROP SIZE)))
                      (INVARIANT '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                                 (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE UNLESS-PROVES-INVARIANT
                  (($IC (LIST 'MUTUAL-EXCLUSIONP 'STATE
                              (LIST 'QUOTE SIZE)))))
         (REWRITE MUTUAL-EXCLUSIONP-PRG) S (REWRITE LISTP-ME-PRG))))

(PROVE-LEMMA CRITICAL-LEADS-TO-LESS-CRITICAL-OR-RIGHT (REWRITE)
   (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
            (LEADS-TO '(AND (CRITICAL STATE (QUOTE ,INDEX))
                            (LESSP (TICKS STATE (QUOTE ,INDEX))
                                   (QUOTE ,(ADD1 TICKS))))
                      '(OR (AND (CRITICAL STATE (QUOTE ,INDEX))
                                (LESSP (TICKS STATE (QUOTE ,INDEX))
                                       (QUOTE ,TICKS)))
                           (TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX))))
                      (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE (REWRITE UNCONDITIONAL-FAIRNESS)
         (REWRITE CRITICAL-UNLESS-LESS-CRITICAL-OR-RIGHT)
         (REWRITE CRITICAL-ENSURES-LESS-CRITICAL-OR-RIGHT)
         (REWRITE TOTAL-PRG))))

(PROVE-LEMMA CRITICAL-TICKS-LEADS-TO-RIGHT (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (LEADS-TO '(AND (CRITICAL STATE (QUOTE ,INDEX))
                             (LESSP (TICKS STATE (QUOTE ,INDEX))
                                    (QUOTE ,(ADD1 TICKS))))
                       '(TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX)))
                       (ME-PRG SIZE)))
    ((INSTRUCTIONS
      (INDUCT (PLUS TICKS J)) PROMOTE PROMOTE
      (REWRITE LEADS-TO-WEAKEN-RIGHT
               (($Q (LIST 'OR
                          (LIST 'AND
                                (LIST 'CRITICAL 'STATE
                                      (LIST 'QUOTE INDEX))
                                (LIST 'LESSP
                                      (LIST 'TICKS 'STATE
                                            (LIST 'QUOTE INDEX))
                                      (LIST 'QUOTE TICKS)))
                          (LIST 'TOKEN 'STATE
                                (LIST 'QUOTE (ADD1-MOD SIZE INDEX)))))))
      PROVE (REWRITE CRITICAL-LEADS-TO-LESS-CRITICAL-OR-RIGHT)
      PROMOTE PROMOTE (DEMOTE 2) (DIVE 1) (DIVE 1) S UP S TOP
      PROMOTE
      (REWRITE LEADS-TO-WEAKEN-RIGHT
               (($Q (LIST 'OR
                          (LIST 'TOKEN 'STATE
                                (LIST 'QUOTE (ADD1-MOD SIZE INDEX)))
                          (LIST 'TOKEN 'STATE
                                (LIST 'QUOTE (ADD1-MOD SIZE INDEX)))))))
      PROVE
      (REWRITE CANCELLATION-LEADS-TO
               (($B (LIST 'AND
```

```
                         (LIST 'CRITICAL 'STATE
                               (LIST 'QUOTE INDEX))
                         (LIST 'LESSP
                               (LIST 'TICKS 'STATE
                                     (LIST 'QUOTE INDEX))
                               (LIST 'QUOTE (ADD1 (SUB1 TICKS))))))))
    (REWRITE LEADS-TO-WEAKEN-RIGHT
             (($Q (LIST 'OR
                         (LIST 'AND
                               (LIST 'CRITICAL 'STATE
                                     (LIST 'QUOTE INDEX))
                               (LIST 'LESSP
                                     (LIST 'TICKS 'STATE
                                           (LIST 'QUOTE INDEX))
                                     (LIST 'QUOTE (ADD1 (SUB1 TICKS)))))
                         (LIST 'TOKEN 'STATE
                               (LIST 'QUOTE (ADD1-MOD SIZE INDEX)))))))
    PROVE (DIVE 2) (DIVE 2) (DIVE 1) (DIVE 2) (DIVE 2) (DIVE 1)
    (DIVE 2) (DIVE 2) (DIVE 1) (DIVE 2) (DIVE 1) S TOP
    (REWRITE CRITICAL-LEADS-TO-LESS-CRITICAL-OR-RIGHT) (DEMOTE 4)
    S)))


(PROVE-LEMMA CRITICAL-LEADS-TO-RIGHT (REWRITE)
   (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
            (LEADS-TO `(CRITICAL STATE (QUOTE ,INDEX))
                      `(TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX)))
                      (ME-PRG SIZE)))
 ((DISABLE ADD1-MOD)
  (USE
   (LEADS-TO-STRENGTHEN-LEFT
    (Q (LIST 'CRITICAL 'STATE (LIST 'QUOTE INDEX)))
    (R (LIST 'TOKEN 'STATE (LIST 'QUOTE (ADD1-MOD SIZE INDEX))))
    (P (LIST
        'AND
        (LIST 'CRITICAL
              'STATE
              (LIST 'QUOTE INDEX))
        (LIST 'LESSP
              (LIST 'TICKS
                    'STATE
                    (LIST 'QUOTE INDEX))
              (LIST 'QUOTE
                    (ADD1 (TICKS
                           (S (ME-PRG SIZE)
                              (ILEADS
                               (LIST 'CRITICAL
                                     'STATE
                                     (LIST 'QUOTE INDEX))
                               (ME-PRG SIZE)
                               (LIST 'TOKEN
                                     'STATE
                                     (LIST 'QUOTE
                                           (ADD1-MOD SIZE INDEX)))))
                           INDEX))))))
    (PRG (ME-PRG SIZE))))))

(PROVE-LEMMA WAIT-LEFT-LEADS-TO-CRITICAL (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (LEADS-TO `(AND (WAIT STATE (QUOTE ,INDEX))
                             (TOKEN STATE (QUOTE ,INDEX)))
                       `(CRITICAL STATE (QUOTE ,INDEX))
                       (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE (REWRITE UNCONDITIONAL-FAIRNESS)
         (REWRITE WAIT-AND-LEFT-CHANNEL-UNLESS-CRITIAL)
         (REWRITE WAIT-AND-LEFT-CHANNEL-ENSURES-CRITIAL)
         (REWRITE TOTAL-PRG))))

(PROVE-LEMMA NON-CRITICAL-LEFT-LEADS-TO-WAIT-LEFT-OR-RIGHT (REWRITE)
```

```
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (LEADS-TO ‘(AND (NON-CRITICAL STATE (QUOTE ,INDEX))
                             (TOKEN STATE (QUOTE ,INDEX)))
                       ‘(OR (AND (WAIT STATE (QUOTE ,INDEX))
                                 (TOKEN STATE (QUOTE ,INDEX)))
                            (TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX))))
                       (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE (REWRITE UNCONDITIONAL-FAIRNESS)
         (REWRITE NON-CRITICAL-LEFT-UNLESS-WAIT-LEFT-OR-RIGHT)
         (REWRITE NON-CRITICAL-LEFT-ENSURES-WAIT-LEFT-OR-RIGHT)
         (REWRITE TOTAL-PRG))))

(PROVE-LEMMA WAIT-LEFT-LEADS-TO-RIGHT (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (LEADS-TO ‘(AND (WAIT STATE (QUOTE ,INDEX))
                             (TOKEN STATE (QUOTE ,INDEX)))
                       ‘(TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX)))
                       (ME-PRG SIZE)))
    ((DISABLE ADD1-MOD LEADS-TO-TRANSITIVE)
     (USE (LEADS-TO-TRANSITIVE
           (PRG (ME-PRG SIZE))
           (P (LIST ’AND
                    (LIST ’WAIT ’STATE (LIST ’QUOTE INDEX))
                    (LIST ’TOKEN ’STATE
                          (LIST ’QUOTE INDEX))))
           (Q (LIST ’CRITICAL ’STATE
                    (LIST ’QUOTE INDEX)))
           (R (LIST ’TOKEN ’STATE
                    (LIST ’QUOTE (ADD1-MOD SIZE
                                           INDEX))))))))

(PROVE-LEMMA NON-CRITICAL-LEFT-LEADS-TO-RIGHT (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE))
             (LEADS-TO ‘(AND (NON-CRITICAL STATE (QUOTE ,INDEX))
                             (TOKEN STATE (QUOTE ,INDEX)))
                       ‘(TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX)))
                       (ME-PRG SIZE)))
    ((DISABLE ADD1-MOD EVAL)
     (USE (CANCELLATION-LEADS-TO
           (P (LIST ’AND
                    (LIST ’NON-CRITICAL ’STATE
                          (LIST ’QUOTE INDEX))
                    (LIST ’TOKEN ’STATE
                          (LIST ’QUOTE INDEX))))
           (B (LIST ’AND
                    (LIST ’WAIT ’STATE
                          (LIST ’QUOTE INDEX))
                    (LIST ’TOKEN ’STATE
                          (LIST ’QUOTE INDEX))))
           (Q (LIST ’TOKEN ’STATE
                    (LIST ’QUOTE (ADD1-MOD SIZE INDEX))))
           (R (LIST ’TOKEN ’STATE
                    (LIST ’QUOTE (ADD1-MOD SIZE INDEX))))
           (PRG (ME-PRG SIZE)))
          (LEADS-TO-WEAKEN-RIGHT
           (PRG (ME-PRG SIZE))
           (P (LIST ’AND
                    (LIST ’NON-CRITICAL ’STATE
                          (LIST ’QUOTE INDEX))
                    (LIST ’TOKEN ’STATE
                          (LIST ’QUOTE INDEX))))
           (Q (LIST ’OR
                    (LIST ’AND
                          (LIST ’WAIT ’STATE
                                (LIST ’QUOTE INDEX))
                          (LIST ’TOKEN ’STATE
                                (LIST ’QUOTE INDEX)))
                    (LIST ’TOKEN ’STATE
                          (LIST ’QUOTE (ADD1-MOD
```

```
                                   SIZE INDEX)))))
           (R (LIST 'OR
                    (LIST 'TOKEN 'STATE
                          (LIST 'QUOTE (ADD1-MOD
                                        SIZE INDEX)))
                    (LIST 'AND
                          (LIST 'WAIT 'STATE
                                (LIST 'QUOTE INDEX))
                          (LIST 'TOKEN 'STATE
                                (LIST 'QUOTE INDEX))))))
          (LEADS-TO-WEAKEN-RIGHT
           (PRG (ME-PRG SIZE))
           (P (LIST 'AND
                    (LIST 'NON-CRITICAL 'STATE
                          (LIST 'QUOTE INDEX))
                    (LIST 'TOKEN 'STATE
                          (LIST 'QUOTE INDEX))))
           (Q (LIST 'OR
                    (LIST 'TOKEN 'STATE
                          (LIST 'QUOTE
                                (ADD1-MOD SIZE INDEX)))
                    (LIST 'TOKEN 'STATE
                          (LIST 'QUOTE
                                (ADD1-MOD SIZE INDEX)))))
           (R (LIST 'TOKEN 'STATE
                    (LIST 'QUOTE
                          (ADD1-MOD SIZE INDEX))))))))

(PROVE-LEMMA CRITICAL-WAIT-NON-CRITICAL (REWRITE)
             (OR (CRITICAL STATE INDEX)
                 (WAIT STATE INDEX)
                 (NON-CRITICAL STATE INDEX)))

(DEFN TOKENS (STATE SIZE)
  (IF (ZEROP SIZE)
      NIL
    (IF (TOKEN STATE (SUB1 SIZE))
        (CONS (SUB1 SIZE)
              (TOKENS STATE (SUB1 SIZE)))
      (TOKENS STATE (SUB1 SIZE)))))

(DEFN CRITICALS (STATE SIZE)
  (IF (ZEROP SIZE)
      NIL
    (IF (CRITICAL STATE (SUB1 SIZE))
        (CONS (SUB1 SIZE)
              (CRITICALS STATE (SUB1 SIZE)))
      (CRITICALS STATE (SUB1 SIZE)))))

(PROVE-LEMMA EQUAL-PLUS-1 (REWRITE)
             (EQUAL (EQUAL (PLUS A B) 1)
                    (OR (AND (EQUAL A 1)
                             (ZEROP B))
                        (AND (EQUAL B 1)
                             (ZEROP A)))))

(PROVE-LEMMA ABOUT-MUTUAL-EXCLUSIONP ()
             (IF (ZEROP (WEIGHT STATE SIZE))
                 (AND (EQUAL (TOKENS STATE SIZE) NIL)
                      (EQUAL (CRITICALS STATE SIZE) NIL))
               (IF (EQUAL (WEIGHT STATE SIZE)
                          1)
                   (OR (AND (EQUAL (LENGTH (TOKENS STATE SIZE)) 1)
                            (EQUAL (CRITICALS STATE SIZE) NIL))
                       (AND (EQUAL (LENGTH (CRITICALS STATE SIZE)) 1)
                            (EQUAL (TOKENS STATE SIZE) NIL)))
                 T))
             ((INDUCT (WEIGHT STATE SIZE))))
```

```
(DISABLE EQUAL-PLUS-1)

(PROVE-LEMMA NUMBERP-CAR-TOKENS (REWRITE)
             (NUMBERP (CAR (TOKENS STATE SIZE))))

(PROVE-LEMMA NUMBERP-CAR-CRITICALS (REWRITE)
             (NUMBERP (CAR (CRITICALS STATE SIZE))))

(PROVE-LEMMA LESSP-CAR-TOKENS (REWRITE)
             (EQUAL (LESSP (CAR (TOKENS STATE SIZE)) SIZE)
                    (NOT (ZEROP SIZE))))

(PROVE-LEMMA LESSP-CAR-CRITICALS (REWRITE)
             (EQUAL (LESSP (CAR (CRITICALS STATE SIZE)) SIZE)
                    (NOT (ZEROP SIZE))))

(PROVE-LEMMA MUTUAL-EXCLUSIONP-REQUIRES-STATE ()
             (IMPLIES (MUTUAL-EXCLUSIONP STATE SIZE)
                      (NOT (ZEROP SIZE))))

(PROVE-LEMMA LESSP-CAR-TOKENS-AND-CRITICALS (REWRITE)
             (IMPLIES (MUTUAL-EXCLUSIONP STATE SIZE)
                      (AND (LESSP (CAR (TOKENS STATE SIZE)) SIZE)
                           (LESSP (CAR (CRITICALS STATE SIZE)) SIZE)))
             ((USE (MUTUAL-EXCLUSIONP-REQUIRES-STATE))))

(PROVE-LEMMA ABOUT-MEMBER-LIST-LENGTH-ONE (REWRITE)
             (IMPLIES (EQUAL (LENGTH LIST) 1)
                      (EQUAL (MEMBER X LIST)
                             (EQUAL X (CAR LIST)))))

(PROVE-LEMMA ABOUT-MEMBER-CAR-TOKENS-CRITICALS (REWRITE)
    (IMPLIES
     (MUTUAL-EXCLUSIONP STATE SIZE)
     (AND (EQUAL (MEMBER INDEX (TOKENS STATE SIZE))
                 (AND (LISTP (TOKENS STATE SIZE))
                      (EQUAL INDEX (CAR (TOKENS STATE SIZE)))))
          (EQUAL (MEMBER INDEX (CRITICALS STATE SIZE))
                 (AND (LISTP (CRITICALS STATE SIZE))
                      (EQUAL INDEX (CAR (CRITICALS STATE SIZE)))))))
    ((USE (ABOUT-MUTUAL-EXCLUSIONP))))

(PROVE-LEMMA NOT-MEMBER-TOKENS (REWRITE)
             (IMPLIES (NOT (LESSP INDEX SIZE))
                      (NOT (MEMBER INDEX (TOKENS STATE SIZE))))
             ((INDUCT (TOKENS STATE SIZE))))

(PROVE-LEMMA NOT-MEMBER-CRITICALS (REWRITE)
             (IMPLIES (NOT (LESSP INDEX SIZE))
                      (NOT (MEMBER INDEX (CRITICALS STATE SIZE))))
             ((INDUCT (CRITICALS STATE SIZE))))

(PROVE-LEMMA TOKEN-EQUALS (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE))
                      (EQUAL (TOKEN STATE INDEX)
                             (MEMBER INDEX (TOKENS STATE SIZE))))
             ((INDUCT (TOKENS STATE SIZE))))

(PROVE-LEMMA CRITICAL-EQUALS (REWRITE)
             (IMPLIES (AND (NUMBERP INDEX)
                           (LESSP INDEX SIZE))
                      (EQUAL (CRITICAL STATE INDEX)
                             (MEMBER INDEX (CRITICALS STATE SIZE))))
             ((INDUCT (CRITICALS STATE SIZE))))

(PROVE-LEMMA MTOKEN-EQUALS (REWRITE)
   (IMPLIES (AND (MUTUAL-EXCLUSIONP STATE SIZE)
                 (NUMBERP INDEX)
```

```
                  (LESSP INDEX SIZE))
             (EQUAL (TOKEN STATE INDEX)
                    (AND (LISTP (TOKENS STATE SIZE))
                         (EQUAL INDEX (CAR (TOKENS STATE SIZE))))))
              ((DISABLE TOKEN-EQUALS)
               (USE (TOKEN-EQUALS))))

(PROVE-LEMMA MCRITICAL-EQUALS (REWRITE)
  (IMPLIES (AND (MUTUAL-EXCLUSIONP STATE SIZE)
                (NUMBERP INDEX)
                (LESSP INDEX SIZE))
           (EQUAL (CRITICAL STATE INDEX)
                  (AND (LISTP (CRITICALS STATE SIZE))
                       (EQUAL INDEX (CAR (CRITICALS STATE SIZE))))))
 ((DISABLE CRITICAL-EQUALS)
  (USE (CRITICAL-EQUALS))))


(PROVE-LEMMA MUTUAL-EXCLUSIONP-IMPLIES (REWRITE)
             (IMPLIES (MUTUAL-EXCLUSIONP STATE SIZE)
                      (IFF (LISTP (TOKENS STATE SIZE))
                           (NOT (LISTP (CRITICALS STATE SIZE)))))
             ((USE (ABOUT-MUTUAL-EXCLUSIONP))))

(DISABLE LESSP-CAR-CRITICALS)
(DISABLE LESSP-CAR-TOKENS)
(DISABLE ABOUT-MEMBER-LIST-LENGTH-ONE)
(DISABLE NOT-MEMBER-TOKENS)
(DISABLE NOT-MEMBER-CRITICALS)

(DISABLE TOKEN)
(DISABLE CRITICAL)
(DISABLE NON-CRITICAL)
(DISABLE WAIT)
(DISABLE MUTUAL-EXCLUSIONP)

(PROVE-LEMMA MUTUAL-EXCLUSIONP-IMPLIES-PRG (REWRITE)
             (IMPLIES (MUTUAL-EXCLUSIONP STATE SIZE)
                      (LISTP (ME-PRG SIZE)))
             ((USE (MUTUAL-EXCLUSIONP-REQUIRES-STATE))
              (ENABLE ME-PRG)))


(PROVE-LEMMA LEFT-LEADS-TO-RIGHT (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE)
                  (INITIAL-CONDITION
                   '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                   (ME-PRG SIZE)))
             (LEADS-TO '(TOKEN STATE (QUOTE ,INDEX))
                       '(TOKEN STATE (QUOTE ,(ADD1-MOD SIZE INDEX)))
                       (ME-PRG SIZE)))
    ((INSTRUCTIONS (DISABLE ADD1-MOD) PROMOTE
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'OR
                             (LIST 'AND
                                   (LIST 'WAIT 'STATE
                                    (LIST 'QUOTE INDEX))
                                   (LIST 'TOKEN 'STATE
                                    (LIST 'QUOTE INDEX)))
                             (LIST 'AND
                                   (LIST 'NON-CRITICAL 'STATE
                                    (LIST 'QUOTE INDEX))
                                   (LIST 'TOKEN 'STATE
                                    (LIST 'QUOTE INDEX))))))
         (CLAIM (EVAL (LIST 'MUTUAL-EXCLUSIONP 'STATE
                            (LIST 'QUOTE SIZE))
                      (S (ME-PRG SIZE)
                         (ILEADS (LIST 'TOKEN 'STATE
                                       (LIST 'QUOTE INDEX))
```

```
                                    (ME-PRG SIZE)
                                    (LIST 'TOKEN 'STATE
                                          (LIST 'QUOTE
                                           (ADD1-MOD SIZE INDEX))))))
                      0)
               (USE-LEMMA CRITICAL-WAIT-NON-CRITICAL
                   ((STATE (S (ME-PRG SIZE)
                              (ILEADS (LIST 'TOKEN 'STATE
                                            (LIST 'QUOTE INDEX))
                                      (ME-PRG SIZE)
                                      (LIST 'TOKEN 'STATE
                                            (LIST 'QUOTE
                                             (ADD1-MOD SIZE INDEX))))))
                    (INDEX INDEX)))
               (BASH (DISABLE ADD1-MOD)) (CONTRADICT 4)
               (REWRITE INVARIANT-IMPLIES)
               (REWRITE MUTUAL-EXCLUSIONP-IS-INVARIANT) PROVE PROVE
               (REWRITE DISJOIN-LEFT) (REWRITE WAIT-LEFT-LEADS-TO-RIGHT)
               (REWRITE NON-CRITICAL-LEFT-LEADS-TO-RIGHT))))

      (DEFN WALK-RING (A B SIZE)
        (IF (NUMBERP A)
            (IF (NUMBERP B)
                (IF (LESSP A SIZE)
                    (IF (LESSP B SIZE)
                        (IF (EQUAL A B)
                            T
                          (WALK-RING (ADD1-MOD SIZE A) B SIZE))
                      T)
                  T)
              T)
          T)
        ((LESSP (IF (NOT (LESSP B A))
                    (DIFFERENCE B A)
                  (PLUS (ADD1 B) (DIFFERENCE SIZE A))))))


      (PROVE-LEMMA ANY-LEFT-LEADS-TO-RIGHT (REWRITE)
          (IMPLIES (AND (NUMBERP LEFT) (LESSP LEFT SIZE) (NUMBERP RIGHT)
                        (LESSP RIGHT SIZE)
                        (INITIAL-CONDITION
                         `(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                         (ME-PRG SIZE)))
                   (LEADS-TO `(TOKEN STATE (QUOTE ,LEFT))
                             `(TOKEN STATE (QUOTE ,RIGHT))
                             (ME-PRG SIZE)))
          ((INSTRUCTIONS (INDUCT (WALK-RING LEFT RIGHT SIZE)) PROMOTE PROMOTE
               (DIVE 1) (DIVE 2) (DIVE 2) (DIVE 1) (DIVE 2) (DIVE 1) = TOP
               (REWRITE Q-LEADS-TO-Q) PROMOTE PROMOTE (DEMOTE 6) (DIVE 1)
               (DIVE 1) (= T) TOP SPLIT
               (REWRITE LEADS-TO-TRANSITIVE
                        (($Q (LIST 'TOKEN 'STATE
                                   (LIST 'QUOTE (ADD1-MOD SIZE LEFT))))))
               (REWRITE LEFT-LEADS-TO-RIGHT) S S S S)))

      (PROVE-LEMMA LESSP-ADD1-MOD-SIZE (REWRITE)
                   (IMPLIES (MUTUAL-EXCLUSIONP STATE SIZE)
                            (LESSP (ADD1-MOD SIZE N) SIZE))
                   ((USE (MUTUAL-EXCLUSIONP-REQUIRES-STATE))))

      (PROVE-LEMMA ANY-CRITICAL-LEADS-TO-RIGHT (REWRITE)
          (IMPLIES (AND (NUMBERP LEFT) (LESSP LEFT SIZE) (NUMBERP RIGHT)
                        (LESSP RIGHT SIZE)
                        (INITIAL-CONDITION
                         `(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                         (ME-PRG SIZE)))
                   (LEADS-TO `(CRITICAL STATE (QUOTE ,LEFT))
                             `(TOKEN STATE (QUOTE ,RIGHT))
                             (ME-PRG SIZE)))
```

```
    ((INSTRUCTIONS
      (DISABLE ADD1-MOD) PROMOTE
      (REWRITE LEADS-TO-TRANSITIVE
               (($Q (LIST 'TOKEN 'STATE
                          (LIST 'QUOTE (ADD1-MOD SIZE LEFT))))))
      (REWRITE CRITICAL-LEADS-TO-RIGHT)
      (REWRITE ANY-LEFT-LEADS-TO-RIGHT) PROVE)))

(PROVE-LEMMA ANY-LEADS-TO-RIGHT (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE)
                  (INITIAL-CONDITION
                   `(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                   (ME-PRG SIZE)))
             (LEADS-TO '(TRUE)
                       `(TOKEN STATE (QUOTE ,INDEX))
                       (ME-PRG SIZE)))
    ((INSTRUCTIONS (DISABLE ADD1-MOD) PROMOTE
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'OR
                             (LIST 'TOKEN 'STATE
                                   (LIST 'QUOTE
                                    (CAR
                                     (TOKENS
                                      (S (ME-PRG SIZE)
                                       (ILEADS '(TRUE) (ME-PRG SIZE)
                                        (LIST 'TOKEN 'STATE
                                         (LIST 'QUOTE INDEX))))
                                      SIZE))))
                             (LIST 'CRITICAL 'STATE
                                   (LIST 'QUOTE
                                    (CAR
                                     (CRITICALS
                                      (S (ME-PRG SIZE)
                                       (ILEADS '(TRUE) (ME-PRG SIZE)
                                        (LIST 'TOKEN 'STATE
                                         (LIST 'QUOTE INDEX))))
                                      SIZE)))))))
         PROMOTE
         (CLAIM (EVAL (LIST 'MUTUAL-EXCLUSIONP 'STATE
                            (LIST 'QUOTE SIZE))
                      (S (ME-PRG SIZE)
                         (ILEADS '(TRUE) (ME-PRG SIZE)
                                 (LIST 'TOKEN 'STATE
                                       (LIST 'QUOTE INDEX)))))
                0)
         (BASH (DISABLE ADD1-MOD)) (CONTRADICT 5)
         (REWRITE INVARIANT-IMPLIES)
         (REWRITE MUTUAL-EXCLUSIONP-IS-INVARIANT) PROVE PROVE
         (REWRITE DISJOIN-LEFT) (REWRITE ANY-LEFT-LEADS-TO-RIGHT)
         (REWRITE NUMBERP-CAR-TOKENS) (REWRITE LESSP-CAR-TOKENS) PROVE
         (REWRITE ANY-CRITICAL-LEADS-TO-RIGHT)
         (REWRITE NUMBERP-CAR-CRITICALS) (REWRITE LESSP-CAR-CRITICALS)
         PROVE)))

(PROVE-LEMMA ME-PRG-NON-ZERO-SIZE (REWRITE)
             (EQUAL (LISTP (ME-PRG SIZE)) (NOT (ZEROP SIZE)))
             ((ENABLE ME-PRG)))

(PROVE-LEMMA WAIT-LEADS-TO-LEFT-WAIT-OR-CRITICAL (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE)
                  (INITIAL-CONDITION
                   `(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                   (ME-PRG SIZE)))
             (LEADS-TO `(AND (TRUE)
                             (WAIT STATE (QUOTE ,INDEX)))
                       `(OR (AND (TOKEN STATE (QUOTE ,INDEX))
                                 (WAIT STATE (QUOTE ,INDEX)))
                            (CRITICAL STATE (QUOTE ,INDEX)))
                       (ME-PRG SIZE)))
```

```
    ((INSTRUCTIONS PROMOTE (REWRITE PSP) (REWRITE ANY-LEADS-TO-RIGHT)
         (REWRITE WAIT-UNLESS-CRITICAL) BASH BASH
         (REWRITE ME-PRG-NON-ZERO-SIZE) PROVE)))

(PROVE-LEMMA WAIT-LEADS-TO-CRITICAL (REWRITE)
    (IMPLIES (AND (NUMBERP INDEX) (LESSP INDEX SIZE)
                  (INITIAL-CONDITION
                   '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
                   (ME-PRG SIZE)))
             (LEADS-TO '(WAIT STATE (QUOTE ,INDEX))
                       '(CRITICAL STATE (QUOTE ,INDEX))
                       (ME-PRG SIZE)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE LEADS-TO-WEAKEN-RIGHT
                  (($Q (LIST 'OR
                             (LIST 'CRITICAL 'STATE
                                   (LIST 'QUOTE INDEX))
                             (LIST 'CRITICAL 'STATE
                                   (LIST 'QUOTE INDEX))))))
         BASH
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'AND '(TRUE)
                             (LIST 'WAIT 'STATE (LIST 'QUOTE INDEX))))))
         BASH
         (REWRITE CANCELLATION-LEADS-TO
                  (($B (LIST 'AND
                             (LIST 'TOKEN 'STATE (LIST 'QUOTE INDEX))
                             (LIST 'WAIT 'STATE (LIST 'QUOTE INDEX))))))
         (REWRITE LEADS-TO-WEAKEN-RIGHT
                  (($Q (LIST 'OR
                             (LIST 'AND
                                   (LIST 'TOKEN 'STATE
                                    (LIST 'QUOTE INDEX))
                                   (LIST 'WAIT 'STATE
                                    (LIST 'QUOTE INDEX)))
                             (LIST 'CRITICAL 'STATE
                                   (LIST 'QUOTE INDEX))))))
         BASH (REWRITE WAIT-LEADS-TO-LEFT-WAIT-OR-CRITICAL)
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'AND
                             (LIST 'WAIT 'STATE (LIST 'QUOTE INDEX))
                             (LIST 'TOKEN 'STATE (LIST 'QUOTE INDEX))))))
         BASH (REWRITE WAIT-LEFT-LEADS-TO-CRITICAL))))


))
```

Appendix D
Min Tree Value Events

This appendix contains the complete events list supporting the proof of the minimum tree value algorithm described in chapter 5.

This event list constructs the proof of the minimun tree value algorithm on top of the proof system presented in Appendix B.

```
(NOTE-LIB "INTERPRETER")


(PROVEALL "MIN" '(


;;;LIST OPERATIONS

(PROVE-LEMMA CAR-APPEND (REWRITE)
             (EQUAL (CAR (APPEND A B))
                    (IF (LISTP A)
                        (CAR A)
                      (CAR B))))

(PROVE-LEMMA LISTP-APPEND (REWRITE)
             (EQUAL (LISTP (APPEND A B))
                    (OR (LISTP A)
                        (LISTP B))))

(PROVE-LEMMA LENGTH-APPEND (REWRITE)
             (EQUAL (LENGTH (APPEND A B))
                    (PLUS (LENGTH A) (LENGTH B))))


(DEFN PLISTP (LIST)
  (IF (LISTP LIST)
      (PLISTP (CDR LIST))
    (EQUAL LIST NIL)))

(PROVE-LEMMA PLISTP-APPEND-PLISTP (REWRITE)
             (EQUAL (PLISTP (APPEND A B))
                    (PLISTP B)))



(PROVE-LEMMA APPEND-PLISTP-NIL (REWRITE)
             (EQUAL (EQUAL (APPEND A NIL) A)
                    (PLISTP A)))

(PROVE-LEMMA NOT-LESSP-COUNT-APPEND (REWRITE)
             (NOT (LESSP (PLUS (COUNT X) (COUNT Y))
                         (COUNT (APPEND X Y)))))

(DEFN ALL-NUMBERPS (LIST)
  (IF (LISTP LIST)
      (IF (NUMBERP (CAR LIST))
          (ALL-NUMBERPS (CDR LIST))
        F)
    T))


(PROVE-LEMMA ALL-NUMBERPS-IMPLIES (REWRITE)
             (IMPLIES (AND (ALL-NUMBERPS LIST)
                           (MEMBER E LIST))
```

```
                      (NUMBERP E)))



;;;SET OPERATIONS

(DEFN SETP (LIST)
  (IF (LISTP LIST)
      (IF (MEMBER (CAR LIST) (CDR LIST))
          F
        (SETP (CDR LIST)))
    T))

(PROVE-LEMMA SETP-APPEND (REWRITE)
             (IMPLIES (OR (NOT (SETP A))
                          (NOT (SETP B)))
                      (NOT (SETP (APPEND A B)))))

(PROVE-LEMMA SETP-MEMBER (REWRITE)
             (IMPLIES (AND (MEMBER X A)
                           (MEMBER X B))
                      (NOT (SETP (APPEND A B)))))

(PROVE-LEMMA SETP-APPEND-CONS (REWRITE)
             (EQUAL (SETP (APPEND A (CONS X B)))
                    (SETP (CONS X (APPEND A B)))))

(PROVE-LEMMA SETP-APPEND-NOT-LISTP (REWRITE)
             (IMPLIES (NOT (LISTP B))
                      (EQUAL (SETP (APPEND A B))
                             (SETP A))))

(PROVE-LEMMA SETP-APPEND-CANONICALIZE (REWRITE)
             (EQUAL (SETP (APPEND A B))
                    (SETP (APPEND B A))))

(PROVE-LEMMA SETP-MEMBER-1 (REWRITE)
             (IMPLIES (AND (SETP (APPEND A B))
                           (MEMBER X B))
                      (NOT (MEMBER X A))))

(PROVE-LEMMA SETP-MEMBER-2 (REWRITE)
             (IMPLIES (AND (SETP (APPEND A B))
                           (MEMBER X A))
                      (NOT (MEMBER X B))))



;;;SUBSET OPERATIONS

(DEFN SUBLISTP (SUB LIST)
  (IF (LISTP SUB)
      (AND (MEMBER (CAR SUB) LIST)
           (SUBLISTP (CDR SUB) LIST))
    T))

(PROVE-LEMMA SUBLISTP-APPEND (REWRITE)
             (EQUAL (SUBLISTP (APPEND A B) LIST)
                    (AND (SUBLISTP A LIST)
                         (SUBLISTP B LIST))))

(PROVE-LEMMA MEMBER-OF-SUBLISTP-IS-MEMBER (REWRITE)
             (IMPLIES (AND (MEMBER A B)
                           (SUBLISTP B C))
                      (MEMBER A C)))

(PROVE-LEMMA SUBLISTP-OF-SUBLISTP-IS-SUBLISTP (REWRITE)
             (IMPLIES (AND (SUBLISTP A B)
```

```
                          (SUBLISTP B C))
                     (SUBLISTP A C)))

(PROVE-LEMMA SUBLISTP-NORMALIZE (REWRITE)
             (IMPLIES (NOT (PLISTP B))
                      (EQUAL (SUBLISTP A B)
                             (SUBLISTP A (APPEND B NIL)))))

(DEFN SEI (A B)
  (IF (LISTP A)
      (SEI (CDR A) (APPEND B (LIST (CAR A))))
    T))

(PROVE-LEMMA SUBLISTP-EASY (REWRITE)
             (SUBLISTP A (APPEND B A))
             ((INDUCT (SEI A B))))

(PROVE-LEMMA SUBLISTP-REFLEXIVE (REWRITE)
             (SUBLISTP A A)
             ((USE (SUBLISTP-EASY (B NIL)))))

(PROVE-LEMMA SUBLISTP-IN-APPEND (REWRITE)
             (IMPLIES (OR (SUBLISTP X A)
                          (SUBLISTP X B))
                      (SUBLISTP X (APPEND A B))))

(PROVE-LEMMA SUBLISTP-IN-CONS (REWRITE)
             (IMPLIES (SUBLISTP A Y)
                      (SUBLISTP A (CONS X Y))))



;;;TREE OPERATIONS

(DEFN NODES-REC (FLAG TREE)
  (IF (LISTP TREE)
      (IF (EQUAL FLAG 'TREE)
          (CONS (CAR TREE)
                (NODES-REC 'FOREST (CDR TREE)))
        (APPEND (NODES-REC 'TREE (CAR TREE))
                (NODES-REC 'FOREST (CDR TREE))))
    NIL))

(DEFN NODES (TREE)
  (NODES-REC 'TREE TREE))

(DEFN ROOTS (FOREST)
  (IF (LISTP FOREST)
      (CONS (CAAR FOREST)
            (ROOTS (CDR FOREST)))
    FOREST))

(DEFN CHILDREN-REC (FLAG NODE TREE)
  (IF (LISTP TREE)
      (IF (EQUAL FLAG 'TREE)
          (IF (EQUAL (CAR TREE) NODE)
              (APPEND (ROOTS (CDR TREE))
                      (CHILDREN-REC 'FOREST NODE (CDR TREE)))
            (CHILDREN-REC 'FOREST NODE (CDR TREE)))
        (APPEND (CHILDREN-REC 'TREE NODE (CAR TREE))
                (CHILDREN-REC 'FOREST NODE (CDR TREE))))
    NIL))

(DEFN CHILDREN (NODE TREE)
  (CHILDREN-REC 'TREE NODE TREE))

(DEFN PARENT-REC (FLAG NODE TREE)
  (IF (LISTP TREE)
```

```
    (IF (EQUAL FLAG 'TREE)
        (IF (MEMBER NODE (ROOTS (CDR TREE)))
            (CONS (CAR TREE)
                  (PARENT-REC 'FOREST NODE (CDR TREE)))
          (PARENT-REC 'FOREST NODE (CDR TREE)))
      (APPEND (PARENT-REC 'TREE NODE (CAR TREE))
              (PARENT-REC 'FOREST NODE (CDR TREE))))
  NIL))

(DEFN PARENT (NODE TREE)
  (CAR (PARENT-REC 'TREE NODE TREE)))

(DEFN PROPER-TREE (FLAG TREE)
  (IF (EQUAL FLAG 'TREE)
      (IF (LISTP TREE)
          (PROPER-TREE 'FOREST (CDR TREE))
        F)
    (IF (LISTP TREE)
        (AND (PROPER-TREE 'TREE (CAR TREE))
             (PROPER-TREE 'FOREST (CDR TREE)))
      (EQUAL TREE NIL))))

(PROVE-LEMMA CANONICALIZE-NODES-REC-FLAG ()
             (EQUAL (NODES-REC FLAG TREE)
                    (IF (EQUAL FLAG 'TREE)
                        (NODES-REC 'TREE TREE)
                      (NODES-REC 'FOREST TREE))))

(PROVE-LEMMA CANONICALIZE-PROPER-TREE-FLAG ()
             (EQUAL (PROPER-TREE FLAG TREE)
                    (IF (EQUAL FLAG 'TREE)
                        (PROPER-TREE 'TREE TREE)
                      (PROPER-TREE 'FOREST TREE))))

(PROVE-LEMMA CANONICALIZE-PARENT-REC-FLAG ()
             (EQUAL (PARENT-REC FLAG CHILD TREE)
                    (IF (EQUAL FLAG 'TREE)
                        (PARENT-REC 'TREE CHILD TREE)
                      (PARENT-REC 'FOREST CHILD TREE))))

(PROVE-LEMMA CANONICALIZE-CHILDREN-REC-FLAG ()
             (EQUAL (CHILDREN-REC FLAG PARENT TREE)
                    (IF (EQUAL FLAG 'TREE)
                        (CHILDREN-REC 'TREE PARENT TREE)
                      (CHILDREN-REC 'FOREST PARENT TREE))))

(PROVE-LEMMA NOT-FLAG-TREE (REWRITE)
             (IMPLIES (AND (NOT (EQUAL FLAG 'TREE))
                           (NOT (EQUAL FLAG 'FOREST)))
                      (AND (EQUAL (NODES-REC FLAG TREE)
                                  (NODES-REC 'FOREST TREE))
                           (EQUAL (PROPER-TREE FLAG TREE)
                                  (PROPER-TREE 'FOREST TREE))
                           (EQUAL (PARENT-REC FLAG CHILD TREE)
                                  (PARENT-REC 'FOREST CHILD TREE))
                           (EQUAL (CHILDREN-REC FLAG PARENT TREE)
                                  (CHILDREN-REC 'FOREST PARENT TREE))))
             ((USE (CANONICALIZE-NODES-REC-FLAG)
                   (CANONICALIZE-PROPER-TREE-FLAG)
                   (CANONICALIZE-PARENT-REC-FLAG)
                   (CANONICALIZE-CHILDREN-REC-FLAG))))

(PROVE-LEMMA PARENT-REC-CHILDREN-REC (REWRITE)
             (EQUAL (MEMBER CHILD (CHILDREN-REC FLAG PARENT TREE))
                    (MEMBER PARENT (PARENT-REC FLAG CHILD TREE))))

(DISABLE PARENT-REC-CHILDREN-REC)

(PROVE-LEMMA PLISTP-CHILDREN-REC (REWRITE)
```

```
             (PLISTP (CHILDREN-REC FLAG PARENT TREE)))

(PROVE-LEMMA PLISTP-PARENT-REC (REWRITE)
             (PLISTP (PARENT-REC FLAG CHILD TREE)))

(PROVE-LEMMA PLISTP-ROOTS (REWRITE)
             (IMPLIES (PROPER-TREE 'FOREST FOREST)
                      (PLISTP (ROOTS FOREST))))

(PROVE-LEMMA MEMBER-ROOTS-MEMBER-FOREST (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'FOREST FOREST)
                           (MEMBER NODE (ROOTS FOREST)))
                      (MEMBER NODE (NODES-REC 'FOREST FOREST))))

(PROVE-LEMMA NOT-MEMBER-NO-PARENT (REWRITE)
             (IMPLIES (AND (PROPER-TREE FLAG TREE)
                           (NOT (MEMBER NODE (NODES-REC FLAG TREE))))
                      (EQUAL (PARENT-REC FLAG NODE TREE)
                             NIL)))

(PROVE-LEMMA MEMBER-CHILD-TREE (REWRITE)
             (IMPLIES (AND (PROPER-TREE FLAG TREE)
                           (MEMBER CHILD (CHILDREN-REC FLAG NODE TREE)))
                      (MEMBER CHILD (NODES-REC FLAG TREE))))

(PROVE-LEMMA SETP-TREE-UNIQUE-PARENT (REWRITE)
  (IMPLIES (AND (PROPER-TREE FLAG TREE)
                (SETP (NODES-REC FLAG TREE)))
           (EQUAL (PARENT-REC FLAG CHILD TREE)
                  (IF (MEMBER CHILD (NODES-REC FLAG TREE))
                      (IF (OR (AND (EQUAL FLAG 'TREE)
                                   (EQUAL (CAR TREE) CHILD))
                              (AND (NOT (EQUAL FLAG 'TREE))
                                   (MEMBER CHILD (ROOTS TREE))))
                          NIL
                        (LIST (CAR (PARENT-REC FLAG CHILD TREE))))
                    NIL))))

(DISABLE SETP-TREE-UNIQUE-PARENT)

(PROVE-LEMMA MEMBER-PARENT-PARENT (REWRITE)
             (IMPLIES (AND (PROPER-TREE FLAG TREE)
                           (SETP (NODES-REC FLAG TREE))
                           (MEMBER PARENT (PARENT-REC FLAG CHILD TREE)))
                      (EQUAL (PARENT-REC FLAG CHILD TREE)
                             (LIST PARENT)))
    ((INSTRUCTIONS PROMOTE (DIVE 1) (REWRITE SETP-TREE-UNIQUE-PARENT)
         TOP (DEMOTE 3) (DIVE 1) (DIVE 2)
         (REWRITE SETP-TREE-UNIQUE-PARENT) TOP PROVE)))

(PROVE-LEMMA PARENT-OF-CHILD (REWRITE)
             (IMPLIES (AND (PROPER-TREE FLAG TREE)
                           (SETP (NODES-REC FLAG TREE))
                           (MEMBER CHILD (CHILDREN-REC FLAG PARENT TREE)))
                      (EQUAL (PARENT-REC FLAG CHILD TREE)
                             (LIST PARENT)))
             ((ENABLE PARENT-REC-CHILDREN-REC)))

(PROVE-LEMMA MEMBER-PARENT-MEMBER-TREE (REWRITE)
             (IMPLIES (MEMBER PARENT (PARENT-REC FLAG CHILD TREE))
                      (MEMBER PARENT (NODES-REC FLAG TREE))))

(PROVE-LEMMA NODE-THAT-HAS-CHILD-IS-IN-TREE (REWRITE)
             (IMPLIES (LISTP (CHILDREN-REC FLAG PARENT TREE))
                      (MEMBER PARENT (NODES-REC FLAG TREE))))

(PROVE-LEMMA NODE-THAT-HAS-PARENT-IS-IN-TREE (REWRITE)
             (IMPLIES (AND (PROPER-TREE FLAG TREE)
                           (LISTP (PARENT-REC FLAG CHILD TREE)))
```

```
                      (MEMBER CHILD (NODES-REC FLAG TREE))))

(PROVE-LEMMA SUBLISTP-CHILDREN-GENERALIZED (REWRITE)
  (IMPLIES (AND (PROPER-TREE FLAG TREE)
                (SUBLISTP CHILDREN (CHILDREN-REC FLAG PARENT TREE)))
           (SUBLISTP CHILDREN
                     (NODES-REC FLAG TREE)))
  ((INDUCT (LENGTH CHILDREN))))

(DISABLE SUBLISTP-CHILDREN-GENERALIZED)

(PROVE-LEMMA SUBLISTP-CHILDREN (REWRITE)
             (IMPLIES (PROPER-TREE FLAG TREE)
                      (SUBLISTP (CHILDREN-REC FLAG PARENT TREE)
                               (NODES-REC FLAG TREE)))
             ((USE (SUBLISTP-CHILDREN-GENERALIZED
                    (CHILDREN (CHILDREN-REC FLAG PARENT TREE))))))

(DEFN SUBTREEP (FLAG SUBTREE TREE)
  (IF (AND (LISTP TREE)
           (LISTP SUBTREE))
      (IF (EQUAL FLAG 'TREE)
          (IF (EQUAL SUBTREE TREE)
              T
            (SUBTREEP 'FOREST SUBTREE (CDR TREE)))
        (IF (SUBTREEP 'TREE SUBTREE (CAR TREE))
            T
          (SUBTREEP 'FOREST SUBTREE (CDR TREE))))
    F))

(DEFN SUBTREES (FLAG TREE)
  (IF (LISTP TREE)
      (IF (EQUAL FLAG 'TREE)
          (CONS TREE (SUBTREES 'FOREST (CDR TREE)))
        (APPEND (SUBTREES 'TREE (CAR TREE))
                (SUBTREES 'FOREST (CDR TREE))))
    NIL))

(PROVE-LEMMA SUBTREEP-SUBTREES (REWRITE)
             (IMPLIES (MEMBER SUBTREE (SUBTREES FLAG TREE))
                      (SUBTREEP FLAG SUBTREE TREE)))

(DEFN NEXT-LEVEL (SUBTREES)
  (IF (LISTP SUBTREES)
      (APPEND (CDAR SUBTREES)
              (NEXT-LEVEL (CDR SUBTREES)))
    SUBTREES))

(PROVE-LEMMA NODES-REC-FOREST-APPEND (REWRITE)
             (EQUAL (NODES-REC 'FOREST (APPEND A B))
                    (APPEND (NODES-REC 'FOREST A)
                            (NODES-REC 'FOREST B))))

(PROVE-LEMMA NEXT-LEVEL-REDUCES-COUNT (REWRITE)
             (IMPLIES (LISTP SUBTREES)
                      (LESSP (COUNT (NEXT-LEVEL SUBTREES))
                             (COUNT SUBTREES))))

(PROVE-LEMMA NEXT-LEVEL-OF-TREE-IN-SUBTREES (REWRITE)
             (IMPLIES (PROPER-TREE 'FOREST FOREST)
                      (SUBLISTP FOREST (SUBTREES 'FOREST FOREST)))
             ((EXPAND (SUBTREES 'TREE (CAR FOREST)))))

(PROVE-LEMMA SUBTREES-OF-SUBTREE-IN-COMPLETE-SUBTREES (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'TREE SUBTREE)
                           (MEMBER SUBTREE (SUBTREES FLAG TREE)))
                      (SUBLISTP (SUBTREES 'TREE SUBTREE)
                                (SUBTREES FLAG TREE))))
```

```
(PROVE-LEMMA SUBTREES-OF-SUBTREES-IN-COMPLETE-SUBTREES (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'FOREST SUBTREES)
                           (SUBLISTP SUBTREES (SUBTREES FLAG TREE)))
                      (SUBLISTP (SUBTREES 'FOREST SUBTREES)
                                (SUBTREES FLAG TREE)))
             ((INDUCT (LENGTH SUBTREES))))

(PROVE-LEMMA NEXT-LEVEL-IN-SUBTREES-FOREST (REWRITE)
             (IMPLIES (PROPER-TREE 'FOREST SUBTREES)
                      (SUBLISTP (NEXT-LEVEL SUBTREES)
                                (SUBTREES 'FOREST SUBTREES))))

(PROVE-LEMMA NEXT-LEVEL-OF-SUBTREES-IN-COMPLETE-SUBTREES (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'FOREST SUBTREES)
                           (SUBLISTP SUBTREES (SUBTREES FLAG TREE)))
                      (SUBLISTP (NEXT-LEVEL SUBTREES)
                                (SUBTREES FLAG TREE)))
             ((USE (SUBLISTP-OF-SUBLISTP-IS-SUBLISTP
                    (A (NEXT-LEVEL SUBTREES))
                    (B (SUBTREES 'FOREST SUBTREES))
                    (C (SUBTREES FLAG TREE))))))

(PROVE-LEMMA PROPER-TREE-OF-APPEND (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'FOREST A)
                           (PROPER-TREE 'FOREST B))
                      (PROPER-TREE 'FOREST (APPEND A B))))

(PROVE-LEMMA PROPER-TREE-NEXT-LEVEL-OF-PROPER-TREE (REWRITE)
             (IMPLIES (PROPER-TREE 'FOREST SUBTREES)
                      (PROPER-TREE 'FOREST (NEXT-LEVEL SUBTREES))))

(PROVE-LEMMA NOT-MEMBER-SUBTREES (REWRITE)
             (IMPLIES (NOT (MEMBER ROOT (NODES-REC FLAG TREE)))
                      (NOT (MEMBER (CONS ROOT FOREST)
                                   (SUBTREES FLAG TREE)))))

(PROVE-LEMMA NOT-MEMBER-NO-CHILDREN (REWRITE)
             (IMPLIES (NOT (MEMBER PARENT (NODES-REC FLAG TREE)))
                      (EQUAL (CHILDREN-REC FLAG PARENT TREE)
                             NIL)))

(PROVE-LEMMA NO-CHILDREN-IN-REST-OF-FOREST (REWRITE)
             (IMPLIES (AND (SETP (APPEND (NODES-REC 'TREE TREE)
                                         (NODES-REC 'FOREST FOREST)))
                           (MEMBER PARENT (NODES-REC 'TREE TREE)))
                      (EQUAL (CHILDREN-REC 'FOREST PARENT FOREST)
                             NIL)))

(PROVE-LEMMA NO-CHILDREN-IN-REST-OF-TREE (REWRITE)
             (IMPLIES (AND (SETP (APPEND (NODES-REC 'TREE TREE)
                                         (NODES-REC 'FOREST FOREST)))
                           (MEMBER PARENT (NODES-REC 'FOREST FOREST)))
                      (EQUAL (CHILDREN-REC 'TREE PARENT TREE)
                             NIL)))

(PROVE-LEMMA MEMBER-SUBTREE-MEMBER-TREE (REWRITE)
             (IMPLIES (MEMBER (CONS ROOT FOREST) (SUBTREES FLAG TREE))
                      (MEMBER ROOT (NODES-REC FLAG TREE))))

(PROVE-LEMMA CHILDREN-OF-SETP-TREE (REWRITE)
   (IMPLIES (AND (SETP (NODES-REC FLAG TREE))
                 (PROPER-TREE FLAG TREE)
                 (MEMBER (CONS ROOT FOREST) (SUBTREES FLAG TREE)))
            (EQUAL (CHILDREN-REC FLAG ROOT TREE)
                   (ROOTS FOREST)))
   ((INDUCT (CHILDREN-REC FLAG ROOT TREE))))

(PROVE-LEMMA NODE-HAS-PARENT (REWRITE)
             (IMPLIES (AND (MEMBER NODE (NODES-REC FLAG TREE))
```

```
                              (PROPER-TREE FLAG TREE)
                              (IF (EQUAL FLAG 'TREE)
                                  (NOT (EQUAL NODE (CAR TREE)))
                                (NOT (MEMBER NODE (ROOTS TREE)))))
                         (MEMBER (CAR (PARENT-REC FLAG NODE TREE))
                                 (NODES-REC FLAG TREE))))


(PROVE-LEMMA PARENT-IS-NOT-ITSELF-GENERALIZED (REWRITE)
   (IMPLIES (AND (SETP (NODES-REC FLAG TREE))
                 (PROPER-TREE FLAG TREE)
                 (LISTP (PARENT-REC FLAG CHILD TREE)))
            (NOT (EQUAL CHILD (CAR (PARENT-REC FLAG CHILD TREE))))))

(PROVE-LEMMA PARENT-IS-NOT-ITSELF (REWRITE)
  (IMPLIES (AND (SETP (NODES-REC 'TREE TREE))
                (PROPER-TREE 'TREE TREE)
                (MEMBER CHILD (CDR (NODES-REC 'TREE TREE))))
           (NOT (EQUAL CHILD (CAR (PARENT-REC 'TREE CHILD TREE)))))
  ((USE (SETP-TREE-UNIQUE-PARENT
         (CHILD CHILD) (TREE TREE) (FLAG 'TREE))
        (PARENT-IS-NOT-ITSELF-GENERALIZED
         (FLAG 'TREE)))
   (DISABLE PARENT-IS-NOT-ITSELF-GENERALIZED)))

(PROVE-LEMMA LISTP-PARENT-REC-EQUALS (REWRITE)
             (IMPLIES (AND (SETP (NODES-REC FLAG TREE))
                           (PROPER-TREE FLAG TREE))
                      (EQUAL (LISTP (PARENT-REC FLAG CHILD TREE))
                             (AND (MEMBER CHILD (NODES-REC FLAG TREE))
                                  (IF (EQUAL FLAG 'TREE)
                                      (NOT (EQUAL CHILD (CAR TREE)))
                                    (NOT (MEMBER CHILD (ROOTS TREE)))))))
             ((USE (SETP-TREE-UNIQUE-PARENT))))


(PROVE-LEMMA PARENT-IS-NOT-CHILD (REWRITE)
             (IMPLIES (AND (SETP (NODES-REC FLAG TREE))
                           (PROPER-TREE FLAG TREE)
                           (LISTP (PARENT-REC FLAG CHILD TREE)))
                      (NOT (MEMBER (CAR (PARENT-REC FLAG CHILD TREE))
                                   (CHILDREN-REC FLAG CHILD TREE)))))

(PROVE-LEMMA PARENT-NOT-IN-CHILDREN (REWRITE)
  (IMPLIES (AND (SETP (NODES-REC 'TREE TREE))
                (PROPER-TREE 'TREE TREE)
                (MEMBER PARENT (CDR (NODES-REC 'TREE TREE))))
           (NOT (MEMBER PARENT (CHILDREN-REC 'TREE PARENT TREE))))
    ((INSTRUCTIONS PROMOTE (DIVE 1) (REWRITE PARENT-REC-CHILDREN-REC)
         (DIVE 2) (REWRITE SETP-TREE-UNIQUE-PARENT) (DIVE 1) (DIVE 2) X
         TOP BASH)))



;;; VARIABLES AND CHANNEL OPERATIONS

(DEFN VALUE (KEY STATE)
  (CDR (ASSOC KEY STATE)))

(DEFN CHANNEL (NAME STATE)
  (VALUE NAME STATE))

(DEFN EMPTY (NAME STATE)
  (NOT (LISTP (CHANNEL NAME STATE))))

(DEFN HEAD (NAME STATE)
  (CAR (CHANNEL NAME STATE)))
```

```
(DEFN SEND (CHANNEL MESSAGE STATE)
  (APPEND (CHANNEL CHANNEL STATE)
          (LIST MESSAGE)))

(DEFN RECEIVE (CHANNEL STATE)
  (CDR (CHANNEL CHANNEL STATE)))

;;; PROGRAM SPECIFIC

(DEFN STATUS (NODE STATE)
  (VALUE (CONS 'STATUS NODE) STATE))

(DEFN FOUND-VALUE (NODE STATE)
  (VALUE (CONS 'FOUND-VALUE NODE) STATE))

(DEFN OUTSTANDING (NODE STATE)
  (VALUE (CONS 'OUTSTANDING NODE) STATE))

(DEFN NODE-VALUE (NODE STATE)
  (VALUE (CONS 'NODE-VALUE NODE) STATE))

(DEFN SEND-FIND (TO-CHILDREN OLD NEW)
  (IF (LISTP TO-CHILDREN)
      (AND (EQUAL (CHANNEL (CAR TO-CHILDREN) NEW)
                  (SEND (CAR TO-CHILDREN) 'FIND OLD))
           (SEND-FIND (CDR TO-CHILDREN) OLD NEW))
    T))

;;; THE FOUR PROGRAM STATEMENTS

(DEFN RECEIVE-FIND (OLD NEW NODE FROM-PARENT TO-PARENT TO-CHILDREN)
  (IF (EQUAL (HEAD FROM-PARENT OLD) 'FIND)
      (AND (EQUAL (CHANNEL FROM-PARENT NEW)
                  (RECEIVE FROM-PARENT OLD))
           (EQUAL (STATUS NODE NEW) 'STARTED)
           (EQUAL (FOUND-VALUE NODE NEW) (NODE-VALUE NODE OLD))
           (EQUAL (OUTSTANDING NODE NEW) (LENGTH TO-CHILDREN))
           (SEND-FIND TO-CHILDREN OLD NEW)
           (EQUAL (CHANNEL TO-PARENT NEW)
                  (IF (ZEROP (LENGTH TO-CHILDREN))
                      (SEND TO-PARENT (NODE-VALUE NODE OLD) OLD)
                    (CHANNEL TO-PARENT OLD)))
           (CHANGED OLD NEW
                    (APPEND (LIST FROM-PARENT TO-PARENT
                                  (CONS 'STATUS NODE)
                                  (CONS 'FOUND-VALUE NODE)
                                  (CONS 'OUTSTANDING NODE))
                            TO-CHILDREN)))
    (CHANGED OLD NEW NIL)))

(DEFN MIN (X Y)
  (IF (LESSP X Y)
      (FIX X)
    (FIX Y)))

(DEFN RECEIVE-REPORT (OLD NEW NODE FROM-CHILD TO-PARENT)
  (IF (EMPTY FROM-CHILD OLD)
      (CHANGED OLD NEW NIL)
    (AND (EQUAL (CHANNEL FROM-CHILD NEW)
                (RECEIVE FROM-CHILD OLD))
         (EQUAL (FOUND-VALUE NODE NEW)
                (MIN (FOUND-VALUE NODE OLD)
                     (HEAD FROM-CHILD OLD)))
         (EQUAL (OUTSTANDING NODE NEW)
                (SUB1 (OUTSTANDING NODE OLD)))
         (EQUAL (CHANNEL TO-PARENT NEW)
                (IF (ZEROP (OUTSTANDING NODE NEW))
                    (SEND TO-PARENT (FOUND-VALUE NODE NEW)
                          OLD)
```

```
                    (CHANNEL TO-PARENT OLD)))
           (CHANGED OLD NEW (LIST FROM-CHILD TO-PARENT
                                  (CONS 'OUTSTANDING NODE)
                                  (CONS 'FOUND-VALUE NODE))))))

(DEFN START (OLD NEW ROOT TO-CHILDREN)
  (IF (EQUAL (STATUS ROOT OLD) 'NOT-STARTED)
      (AND (EQUAL (STATUS ROOT NEW) 'STARTED)
           (EQUAL (FOUND-VALUE ROOT NEW) (NODE-VALUE ROOT OLD))
           (EQUAL (OUTSTANDING ROOT NEW) (LENGTH TO-CHILDREN))
           (SEND-FIND TO-CHILDREN OLD NEW)
           (CHANGED OLD NEW
                    (APPEND (LIST (CONS 'STATUS ROOT)
                                  (CONS 'FOUND-VALUE ROOT)
                                  (CONS 'OUTSTANDING ROOT))
                            TO-CHILDREN)))
    (CHANGED OLD NEW NIL)))

(DEFN ROOT-RECEIVE-REPORT (OLD NEW ROOT FROM-CHILD)
  (IF (EMPTY FROM-CHILD OLD)
      (CHANGED OLD NEW NIL)
    (AND (EQUAL (CHANNEL FROM-CHILD NEW)
                (RECEIVE FROM-CHILD OLD))
         (EQUAL (FOUND-VALUE ROOT NEW)
                (MIN (FOUND-VALUE ROOT OLD)
                     (HEAD FROM-CHILD OLD)))
         (EQUAL (OUTSTANDING ROOT NEW)
                (SUB1 (OUTSTANDING ROOT OLD)))
         (CHANGED OLD NEW (LIST FROM-CHILD
                                (CONS 'OUTSTANDING ROOT)
                                (CONS 'FOUND-VALUE ROOT))))))

;;; THE PROGRAM

(DEFN RFP (NODE CHILDREN)
  (IF (LISTP CHILDREN)
      (CONS (CONS NODE (CAR CHILDREN))
            (RFP NODE (CDR CHILDREN)))
    NIL))

(DEFN RECEIVE-FIND-PRG (NODES TREE)
  (IF (LISTP NODES)
      (CONS (LIST 'RECEIVE-FIND
                  (CAR NODES)
                  (CONS (PARENT (CAR NODES) TREE) (CAR NODES))
                  (CONS (CAR NODES) (PARENT (CAR NODES) TREE))
                  (RFP (CAR NODES) (CHILDREN (CAR NODES) TREE)))
            (RECEIVE-FIND-PRG (CDR NODES) TREE))
    NIL))

(PROVE-LEMMA MEMBER-RECEIVE-FIND-PRG (REWRITE)
             (EQUAL (MEMBER STATEMENT (RECEIVE-FIND-PRG NODES TREE))
                    (AND (EQUAL (CAR STATEMENT) 'RECEIVE-FIND)
                         (MEMBER (CADR STATEMENT) NODES)
                         (LISTP (CADDR STATEMENT))
                         (EQUAL (CAADDR STATEMENT)
                                (PARENT (CADR STATEMENT) TREE))
                         (EQUAL (CDADDR STATEMENT) (CADR STATEMENT))
                         (LISTP (CADDDR STATEMENT))
                         (EQUAL (CAADDDR STATEMENT) (CADR STATEMENT))
                         (EQUAL (CDADDDR STATEMENT)
                                (PARENT (CADR STATEMENT) TREE))
                         (EQUAL (CADDDDR STATEMENT)
                                (RFP (CADR STATEMENT)
                                     (CHILDREN (CADR STATEMENT) TREE)))
                         (EQUAL (CDDDDDR STATEMENT) NIL)))
             ((DISABLE CHILDREN PARENT)))

(DEFN RRP (NODE CHILDREN PARENT)
```

```
  (IF (LISTP CHILDREN)
      (CONS (LIST 'RECEIVE-REPORT
                  NODE
                  (CONS (CAR CHILDREN) NODE)
                  (CONS NODE PARENT))
            (RRP NODE (CDR CHILDREN) PARENT))
    NIL))

(PROVE-LEMMA MEMBER-RRP (REWRITE)
             (EQUAL (MEMBER STATEMENT (RRP NODE CHILDREN PARENT))
                    (AND (EQUAL (CAR STATEMENT) 'RECEIVE-REPORT)
                         (EQUAL (CADR STATEMENT) NODE)
                         (LISTP (CADDR STATEMENT))
                         (MEMBER (CAADDR STATEMENT) CHILDREN)
                         (EQUAL (CDADDR STATEMENT) NODE)
                         (LISTP (CADDDR STATEMENT))
                         (EQUAL (CAADDDR STATEMENT) NODE)
                         (EQUAL (CDADDDR STATEMENT) PARENT)
                         (EQUAL (CDDDDR STATEMENT) NIL))))

(DEFN RECEIVE-REPORT-PRG (NODES TREE)
  (IF (LISTP NODES)
      (APPEND (RRP (CAR NODES) (CHILDREN (CAR NODES) TREE)
                   (PARENT (CAR NODES) TREE))
              (RECEIVE-REPORT-PRG (CDR NODES) TREE))
    NIL))

(PROVE-LEMMA MEMBER-RECEIVE-REPORT-PRG (REWRITE)
             (EQUAL (MEMBER STATEMENT (RECEIVE-REPORT-PRG NODES TREE))
                    (AND (EQUAL (CAR STATEMENT) 'RECEIVE-REPORT)
                         (MEMBER (CADR STATEMENT) NODES)
                         (LISTP (CADDR STATEMENT))
                         (MEMBER (CAADDR STATEMENT)
                                 (CHILDREN (CADR STATEMENT) TREE))
                         (EQUAL (CDADDR STATEMENT) (CADR STATEMENT))
                         (LISTP (CADDDR STATEMENT))
                         (EQUAL (CAADDDR STATEMENT) (CADR STATEMENT))
                         (EQUAL (CDADDDR STATEMENT)
                                (PARENT (CADR STATEMENT) TREE))
                         (EQUAL (CDDDDR STATEMENT) NIL)))
             ((DISABLE PARENT CHILDREN)))

(DEFN START-PRG (ROOT TREE)
  (LIST (LIST 'START ROOT (RFP ROOT (CHILDREN ROOT TREE)))))

(PROVE-LEMMA MEMBER-START-PRG (REWRITE)
             (EQUAL (MEMBER STATEMENT (START-PRG ROOT TREE))
                    (AND (EQUAL (CAR STATEMENT) 'START)
                         (EQUAL (CADR STATEMENT) ROOT)
                         (EQUAL (CADDR STATEMENT)
                                (RFP ROOT (CHILDREN ROOT TREE)))
                         (EQUAL (CDDDR STATEMENT) NIL)))
             ((DISABLE CHILDREN)))

(DEFN RRRP (ROOT CHILDREN)
  (IF (LISTP CHILDREN)
      (CONS (LIST 'ROOT-RECEIVE-REPORT
                  ROOT
                  (CONS (CAR CHILDREN) ROOT))
            (RRRP ROOT (CDR CHILDREN)))
    NIL))

(PROVE-LEMMA MEMBER-RRRP (REWRITE)
             (EQUAL (MEMBER STATEMENT (RRRP ROOT CHILDREN))
                    (AND (EQUAL (CAR STATEMENT) 'ROOT-RECEIVE-REPORT)
                         (EQUAL (CADR STATEMENT) ROOT)
                         (LISTP (CADDR STATEMENT))
                         (MEMBER (CAADDR STATEMENT) CHILDREN)
                         (EQUAL (CDADDR STATEMENT) ROOT)
```

```
                         (EQUAL (CDDDR STATEMENT) NIL))))

(DEFN ROOT-RECEIVE-REPORT-PRG (ROOT TREE)
  (RRRP ROOT (CHILDREN ROOT TREE)))

(PROVE-LEMMA MEMBER-ROOT-RECEIVE-REPORT-PRG (REWRITE)
             (EQUAL (MEMBER STATEMENT (ROOT-RECEIVE-REPORT-PRG ROOT TREE))
                    (AND (EQUAL (CAR STATEMENT) 'ROOT-RECEIVE-REPORT)
                         (EQUAL (CADR STATEMENT) ROOT)
                         (LISTP (CADDR STATEMENT))
                         (MEMBER (CAADDR STATEMENT) (CHILDREN ROOT TREE))
                         (EQUAL (CDADDR STATEMENT) ROOT)
                         (EQUAL (CDDDR STATEMENT) NIL)))
             ((DISABLE CHILDREN)))

(DEFN TREE-PRG (TREE)
  (APPEND (START-PRG (CAR TREE) TREE)
          (APPEND (ROOT-RECEIVE-REPORT-PRG (CAR TREE) TREE)
                  (APPEND (RECEIVE-FIND-PRG (CDR (NODES TREE)) TREE)
                          (RECEIVE-REPORT-PRG (CDR (NODES TREE)) TREE)))))

(PROVE-LEMMA EQUAL-IF (REWRITE)
             (EQUAL (EQUAL (IF TEST P1 P2)
                           (IF TEST R1 R2))
                    (IF TEST
                        (EQUAL P1 R1)
                      (EQUAL P2 R2))))

(PROVE-LEMMA MEMBER-TREE-PRG (REWRITE)
   (EQUAL (MEMBER STATEMENT (TREE-PRG TREE))
;(START ROOT TO-CHILDREN)
          (OR (AND (EQUAL (CAR STATEMENT) 'START)
                   (EQUAL (CADR STATEMENT) (CAR TREE))
                   (EQUAL (CADDR STATEMENT)
                          (RFP (CAR TREE)
                               (CHILDREN (CAR TREE) TREE)))
                   (EQUAL (CDDDR STATEMENT) NIL))
;(ROOT-RECEIVE-REPORT ROOT FROM-CHILD)
         (AND (EQUAL (CAR STATEMENT) 'ROOT-RECEIVE-REPORT)
              (EQUAL (CADR STATEMENT) (CAR TREE))
              (LISTP (CADDR STATEMENT))
              (MEMBER (CAADDR STATEMENT)
                      (CHILDREN (CAR TREE) TREE))
              (EQUAL (CDADDR STATEMENT) (CAR TREE))
              (EQUAL (CDDDR STATEMENT) NIL))
;(RECEIVE-FIND NODE FROM-PARENT TO-PARENT TO-CHILDREN)
         (AND (EQUAL (CAR STATEMENT) 'RECEIVE-FIND)
              (MEMBER (CADR STATEMENT) (CDR (NODES TREE)))
              (LISTP (CADDR STATEMENT))
              (EQUAL (CAADDR STATEMENT)
                     (PARENT (CADR STATEMENT) TREE))
              (EQUAL (CDADDR STATEMENT) (CADR STATEMENT))
              (LISTP (CADDDR STATEMENT))
              (EQUAL (CAADDDR STATEMENT) (CADR STATEMENT))
              (EQUAL (CDADDDR STATEMENT)
                     (PARENT (CADR STATEMENT) TREE))
              (EQUAL (CADDDDR STATEMENT)
                     (RFP (CADR STATEMENT)
                          (CHILDREN (CADR STATEMENT) TREE)))
              (EQUAL (CDDDDDR STATEMENT) NIL))
;(RECEIVE-REPORT NODE FROM-CHILD TO-PARENT)
         (AND (EQUAL (CAR STATEMENT) 'RECEIVE-REPORT)
              (MEMBER (CADR STATEMENT) (CDR (NODES TREE)))
              (LISTP (CADDR STATEMENT))
              (MEMBER (CAADDR STATEMENT)
                      (CHILDREN (CADR STATEMENT) TREE))
              (EQUAL (CDADDR STATEMENT) (CADR STATEMENT))
              (LISTP (CADDDR STATEMENT))
              (EQUAL (CAADDDR STATEMENT) (CADR STATEMENT))
```

```
              (EQUAL (CDADDDR STATEMENT)
                     (PARENT (CADR STATEMENT) TREE))
              (EQUAL (CDDDDR STATEMENT) NIL))))
   ((DISABLE PARENT CHILDREN NODES
             ROOT-RECEIVE-REPORT-PRG
             START-PRG)))

;;; CORRECTNESS

(DEFN TREEP (TREE)
  (AND (SETP (NODES TREE))
       (ALL-NUMBERPS (NODES TREE))
       (PROPER-TREE 'TREE TREE)))

(DEFN TOTAL-OUTSTANDING (NODES TREE STATE)
  (IF (LISTP NODES)
      (PLUS (TOTAL-OUTSTANDING (CDR NODES) TREE STATE)
            (IF (EQUAL (STATUS (CAR NODES) STATE) 'STARTED)
                (OUTSTANDING (CAR NODES) STATE)
              (ADD1 (LENGTH (CHILDREN (CAR NODES) TREE)))))
    0))

(DEFN DL (DOWN-LINKS STATE)
  (IF (LISTP DOWN-LINKS)
      (AND (OR (AND (EMPTY (CAR DOWN-LINKS) STATE)
                    (EQUAL (STATUS (CAAR DOWN-LINKS) STATE)
                           (STATUS (CDAR DOWN-LINKS) STATE)))
               (AND (EQUAL (CHANNEL (CAR DOWN-LINKS) STATE)
                           (LIST 'FIND))
                    (EQUAL (STATUS (CAAR DOWN-LINKS) STATE)
                           'STARTED)
                    (EQUAL (STATUS (CDAR DOWN-LINKS) STATE)
                           'NOT-STARTED)))
           (DL (CDR DOWN-LINKS) STATE))
    T))

(DEFN DONE (NODE STATE)
  (AND (EQUAL (STATUS NODE STATE) 'STARTED)
       (ZEROP (OUTSTANDING NODE STATE))))

(DEFN UL (UP-LINKS STATE)
  (IF (LISTP UP-LINKS)
      (AND (OR (EMPTY (CAR UP-LINKS) STATE)
               (AND (EQUAL (CHANNEL (CAR UP-LINKS) STATE)
                           (LIST (FOUND-VALUE (CAAR UP-LINKS) STATE)))
                    (DONE (CAAR UP-LINKS) STATE)))
           (UL (CDR UP-LINKS) STATE))
    T))

(DEFN REPORTED (NODE PARENT STATE)
  (AND (DONE NODE STATE)
       (EMPTY (CONS NODE PARENT) STATE)))

(DEFN NUMBER-NOT-REPORTED (CHILDREN PARENT STATE)
  (IF (LISTP CHILDREN)
      (IF (REPORTED (CAR CHILDREN) PARENT STATE)
          (NUMBER-NOT-REPORTED (CDR CHILDREN) PARENT STATE)
        (ADD1 (NUMBER-NOT-REPORTED (CDR CHILDREN) PARENT STATE)))
    0))

(DEFN MIN-OF-REPORTED (CHILDREN PARENT STATE MIN)
  (IF (LISTP CHILDREN)
      (IF (REPORTED (CAR CHILDREN) PARENT STATE)
          (MIN (FOUND-VALUE (CAR CHILDREN) STATE)
               (MIN-OF-REPORTED (CDR CHILDREN) PARENT STATE MIN))
        (MIN-OF-REPORTED (CDR CHILDREN) PARENT STATE MIN))
    MIN))

(DEFN NO (NODES TREE STATE)
```

```
  (IF (LISTP NODES)
      (AND (IF (EQUAL (STATUS (CAR NODES) STATE) 'STARTED)
               (AND (EQUAL
                     (OUTSTANDING (CAR NODES) STATE)
                     (NUMBER-NOT-REPORTED (CHILDREN (CAR NODES) TREE)
                                          (CAR NODES) STATE))
                    (EQUAL
                     (FOUND-VALUE (CAR NODES) STATE)
                     (MIN-OF-REPORTED (CHILDREN (CAR NODES) TREE)
                                      (CAR NODES) STATE
                                      (NODE-VALUE (CAR NODES) STATE))))
             T)
           (NO (CDR NODES) TREE STATE))
    T))

(DEFN DOWN-LINKS-1 (PARENT CHILDREN)
  (IF (LISTP CHILDREN)
      (CONS (CONS PARENT (CAR CHILDREN))
            (DOWN-LINKS-1 PARENT (CDR CHILDREN)))
    NIL))

(DEFN DOWN-LINKS (NODES TREE)
  (IF (LISTP NODES)
      (APPEND (DOWN-LINKS-1 (CAR NODES) (CHILDREN (CAR NODES) TREE))

              (DOWN-LINKS (CDR NODES) TREE))
    NIL))

(DEFN UP-LINKS (NODES TREE)
  (IF (LISTP NODES)
      (CONS (CONS (CAR NODES) (PARENT (CAR NODES) TREE))
            (UP-LINKS (CDR NODES) TREE))
    NIL))

(DEFN INV (TREE STATE)
  (AND (DL (DOWN-LINKS (NODES TREE) TREE) STATE)
       (UL (UP-LINKS (CDR (NODES TREE)) TREE) STATE)
       (NO (NODES TREE) TREE STATE)))

(DEFN NOT-STARTED (NODES STATE)
  (IF (LISTP NODES)
      (AND (EQUAL (STATUS (CAR NODES) STATE) 'NOT-STARTED)
           (NOT-STARTED (CDR NODES) STATE))
    T))

(DEFN ALL-CHANNELS (TREE)
  (APPEND (UP-LINKS (CDR (NODES TREE)) TREE)
          (DOWN-LINKS (NODES TREE) TREE)))

(DEFN ALL-EMPTY (CHANNELS STATE)
  (IF (LISTP CHANNELS)
      (AND (EMPTY (CAR CHANNELS) STATE)
           (ALL-EMPTY (CDR CHANNELS) STATE))
    T))

(DEFN MIN-NODE-VALUE (NODES STATE MIN)
  (IF (LISTP NODES)
      (MIN (NODE-VALUE (CAR NODES) STATE)
           (MIN-NODE-VALUE (CDR NODES) STATE MIN))
    MIN))

(DEFN CORRECT (TREE STATE)
  (EQUAL (FOUND-VALUE (CAR TREE) STATE)
         (MIN-NODE-VALUE (CDR (NODES TREE)) STATE
                         (NODE-VALUE (CAR TREE) STATE))))

;;; PROOF OF CORRECTNESS

(PROVE-LEMMA ALL-EMPTY-IMPLIES-EMPTY (REWRITE)
```

```
             (IMPLIES (AND (ALL-EMPTY CHANNELS STATE)
                           (MEMBER CHANNEL CHANNELS))
                      (NOT (LISTP (CHANNEL CHANNEL STATE)))))

(PROVE-LEMMA NOT-STARTED-IMPLIES-NOT-STARTED (REWRITE)
             (IMPLIES (AND (NOT-STARTED NODES STATE)
                           (MEMBER NODE NODES))
                      (EQUAL (CDR (ASSOC (CONS 'STATUS NODE) STATE))
                             'NOT-STARTED)))

(PROVE-LEMMA ALL-EMPTY-APPEND (REWRITE)
             (EQUAL (ALL-EMPTY (APPEND A B) STATE)
                    (AND (ALL-EMPTY A STATE)
                         (ALL-EMPTY B STATE))))

(PROVE-LEMMA ALL-EMPTY-IMPLIES-UL (REWRITE)
             (IMPLIES (ALL-EMPTY UP-LINKS STATE)
                      (UL UP-LINKS STATE)))

(DEFN NODES-IN-CHANNELS (CHANNELS)
  (IF (LISTP CHANNELS)
      (CONS (CAAR CHANNELS)
            (CONS (CDAR CHANNELS)
                  (NODES-IN-CHANNELS (CDR CHANNELS))))
    NIL))

(PROVE-LEMMA ALL-EMPTY-NOT-STARTED-IMPLIES-DL (REWRITE)
  (IMPLIES (AND (ALL-EMPTY DOWN-LINKS STATE)
                (NOT-STARTED (NODES-IN-CHANNELS DOWN-LINKS) STATE))
                      (DL DOWN-LINKS STATE)))

(PROVE-LEMMA NOT-STARTED-IMPLIES-NO (REWRITE)
             (IMPLIES (NOT-STARTED NODES STATE)
                      (NO NODES TREE STATE)))

(PROVE-LEMMA NODES-IN-DOWN-LINKS-1-IN-NODES (REWRITE)
             (EQUAL (MEMBER NODE (NODES-IN-CHANNELS
                                  (DOWN-LINKS-1 PARENT CHILDREN)))
                    (IF (LISTP CHILDREN)
                        (MEMBER NODE (CONS PARENT CHILDREN))
                      F)))

(PROVE-LEMMA NODES-IN-CHANNELS-APPEND (REWRITE)
             (EQUAL (NODES-IN-CHANNELS (APPEND A B))
                    (APPEND (NODES-IN-CHANNELS A)
                            (NODES-IN-CHANNELS B))))

(PROVE-LEMMA NODES-IN-DOWN-LINKS-IN-NODES (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                           (MEMBER NODE (NODES-IN-CHANNELS
                                         (DOWN-LINKS NODES TREE))))
                      (MEMBER NODE (NODES TREE))))

(PROVE-LEMMA SUBLISTP-NOT-STARTED (REWRITE)
             (IMPLIES (AND (SUBLISTP SUB LIST)
                           (NOT-STARTED LIST STATE))
                      (NOT-STARTED SUB STATE)))

(PROVE-LEMMA SUBLISTP-DOWN-LINKS-1 (REWRITE)
             (IMPLIES (AND (SUBLISTP CHILDREN NODES)
                           (MEMBER PARENT NODES))
                      (SUBLISTP (NODES-IN-CHANNELS
                                 (DOWN-LINKS-1 PARENT CHILDREN))
                                NODES)))

(PROVE-LEMMA CHILDREN-OF-NON-NODE (REWRITE)
             (IMPLIES (NOT (MEMBER PARENT (NODES-REC FLAG TREE)))
                      (EQUAL (CHILDREN-REC FLAG PARENT TREE)
                             NIL)))
```

```
(PROVE-LEMMA DOWN-LINKS-IS-SUBLISTP (REWRITE)
   (IMPLIES (PROPER-TREE 'TREE TREE)
            (SUBLISTP (NODES-IN-CHANNELS (DOWN-LINKS NODES TREE))
                      (NODES-REC 'TREE TREE)))
    ((INSTRUCTIONS (INDUCT (LENGTH NODES))
         (CLAIM (MEMBER (CAR NODES) (NODES-REC 'TREE TREE)) 0) BASH
         BASH BASH)))

(PROVE-LEMMA INITIAL-CONDITIONS-IMPLY-INVARIANT (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                           (ALL-EMPTY (ALL-CHANNELS TREE) STATE)
                           (NOT-STARTED (NODES TREE) STATE))
                      (INV TREE STATE))
    ((INSTRUCTIONS BASH PROMOTE
         (REWRITE ALL-EMPTY-NOT-STARTED-IMPLIES-DL)
         (REWRITE SUBLISTP-NOT-STARTED (($LIST (NODES-REC 'TREE TREE))))
         (REWRITE DOWN-LINKS-IS-SUBLISTP))))

(DEFN FOUND-VALUE-NODE-VALUE (SUBTREES STATE)
  (IF (LISTP SUBTREES)
      (AND (EQUAL (FOUND-VALUE (CAAR SUBTREES) STATE)
                  (MIN-NODE-VALUE (CDR (NODES-REC 'TREE (CAR SUBTREES)))
                                  STATE
                                  (NODE-VALUE (CAAR SUBTREES) STATE)))
           (FOUND-VALUE-NODE-VALUE (CDR SUBTREES) STATE))
    T))

(DEFN NATI (SUBTREES)
  (IF (LISTP SUBTREES)
      (NATI (NEXT-LEVEL SUBTREES))
    T))

(PROVE-LEMMA FOUND-VALUE-NODE-VALUE-APPEND (REWRITE)
             (EQUAL (FOUND-VALUE-NODE-VALUE (APPEND A B) STATE)
                    (AND (FOUND-VALUE-NODE-VALUE A STATE)
                         (FOUND-VALUE-NODE-VALUE B STATE))))

;FIND-VALUE-OF-NODE-VALUE FOR A SUBTREE IS TRUE IF
;FIND-VALUE-OF-NODE-VALUE FOR THE NEXT-LEVEL OF THAT SUBTREE IS TRUE.

(PROVE-LEMMA NO-IMPLIES (REWRITE)
  (IMPLIES (AND (NO NODES TREE STATE)
                (MEMBER NODE NODES)
                (EQUAL (STATUS NODE STATE) 'STARTED))
           (AND (EQUAL (NUMBER-NOT-REPORTED (CHILDREN NODE TREE)
                                            NODE STATE)
                       (CDR (ASSOC (CONS 'OUTSTANDING NODE) STATE)))
                (NUMBERP (CDR (ASSOC (CONS 'OUTSTANDING NODE)
                                     STATE)))
                (EQUAL (CDR (ASSOC (CONS 'FOUND-VALUE NODE) STATE))
                       (MIN-OF-REPORTED (CHILDREN NODE TREE)
                                        NODE STATE
                                        (NODE-VALUE NODE STATE))))))

(PROVE-LEMMA TOTAL-OUTSTANDING-0-IMPLIES (REWRITE)
  (AND (IMPLIES (AND (EQUAL (TOTAL-OUTSTANDING NODES TREE STATE) 0)
                     (MEMBER NODE NODES)
                     (NUMBERP (CDR (ASSOC (CONS 'OUTSTANDING NODE)
                                          STATE))))
                (EQUAL (CDR (ASSOC (CONS 'OUTSTANDING NODE) STATE))
                       0))
       (IMPLIES (AND (EQUAL (TOTAL-OUTSTANDING NODES TREE STATE) 0)
                     (MEMBER NODE NODES))
                (EQUAL (CDR (ASSOC (CONS 'STATUS NODE) STATE))
                       'STARTED))))

(PROVE-LEMMA NUMBER-NOT-REPORTED-0-IMPLIES (REWRITE)
  (IMPLIES (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT STATE)
                       0)
```

```
                (MEMBER NODE CHILDREN))
           (REPORTED NODE PARENT STATE))
   ((DISABLE REPORTED)))

(PROVE-LEMMA PROPER-TREE-TREE-IMPLIES-NODES-EXISTS (REWRITE)
             (IMPLIES (PROPER-TREE 'TREE TREE)
                      (LISTP (NODES-REC 'TREE TREE))))

(PROVE-LEMMA MIN-OF-TWO-NODES-VALUES (REWRITE)
  (EQUAL (MIN (MIN-NODE-VALUE FOREST-1
                              STATE
                              (CDR (ASSOC (CONS 'NODE-VALUE ROOT)
                                          STATE)))
              (MIN-NODE-VALUE REST-OF-FOREST
                              STATE MIN))
         (MIN-NODE-VALUE (CONS ROOT (APPEND FOREST-1
                                            REST-OF-FOREST))
                         STATE MIN)))

(PROVE-LEMMA FOUND-VALUE-MIN-VALUE-GENERALIZED (REWRITE)
             (IMPLIES (AND (FOUND-VALUE-NODE-VALUE FOREST STATE)
                           (EQUAL (NUMBER-NOT-REPORTED (ROOTS FOREST)
                                                       ROOT STATE)
                                  0)
                           (PROPER-TREE 'FOREST FOREST))
                      (EQUAL (MIN-OF-REPORTED (ROOTS FOREST)
                                              ROOT STATE
                                              MIN)
                             (MIN-NODE-VALUE (NODES-REC 'FOREST FOREST)
                                             STATE
                                             MIN)))
    ((INSTRUCTIONS (INDUCT (LENGTH FOREST))
         (BASH T (DISABLE MIN REPORTED)) PROMOTE (DIVE 1) (DIVE 1) = UP
         (REWRITE MIN-OF-TWO-NODES-VALUES) TOP DROP
         (BASH (DISABLE MIN)) PROVE)))

(PROVE-LEMMA NO-AT-TERMINATION (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                           (PROPER-TREE 'FOREST SUBTREES)
                           (SETP (NODES-REC 'TREE TREE))
                           (NO (NODES-REC 'TREE TREE) TREE STATE)
                           (EQUAL (TOTAL-OUTSTANDING
                                   (NODES-REC 'TREE TREE)
                                   TREE STATE)
                                  0)
                           (SUBLISTP SUBTREES (SUBTREES 'TREE TREE)))
                      (FOUND-VALUE-NODE-VALUE SUBTREES STATE))
    ((INSTRUCTIONS (INDUCT (NATI SUBTREES)) CHANGE-GOAL PROVE (BASH T)
         (INDUCT (FOUND-VALUE-NODE-VALUE SUBTREES STATE)) CHANGE-GOAL
         PROVE (BASH T) PROMOTE (DIVE 1)
         (REWRITE FOUND-VALUE-MIN-VALUE-GENERALIZED) TOP S (DIVE 1)
         (DIVE 1) (= (CHILDREN W TREE)) UP
         (REWRITE NO-IMPLIES (($NODES (NODES-REC 'TREE TREE))))
         (REWRITE TOTAL-OUTSTANDING-0-IMPLIES
                  (($NODES (NODES-REC 'TREE TREE)) ($TREE TREE)))
         TOP S (REWRITE MEMBER-SUBTREE-MEMBER-TREE)
         (REWRITE NO-IMPLIES
                  (($NODES (NODES-REC 'TREE TREE)) ($TREE TREE)))
         (REWRITE MEMBER-SUBTREE-MEMBER-TREE) (DIVE 1) S
         (REWRITE TOTAL-OUTSTANDING-0-IMPLIES
                  (($NODES (NODES-REC 'TREE TREE)) ($TREE TREE)))
         TOP S (REWRITE MEMBER-SUBTREE-MEMBER-TREE)
         (REWRITE MEMBER-SUBTREE-MEMBER-TREE) (DIVE 1) S
         (REWRITE TOTAL-OUTSTANDING-0-IMPLIES
                  (($NODES (NODES-REC 'TREE TREE)) ($TREE TREE)))
         TOP S (REWRITE MEMBER-SUBTREE-MEMBER-TREE))))

(PROVE-LEMMA INV-IMPLIES-AUGMENTED-CORRECTNESS-CONDITION (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'TREE TREE)
```

```
                              (SETP (NODES-REC 'TREE TREE))
                              (INV TREE STATE)
                              (EQUAL (TOTAL-OUTSTANDING (NODES TREE)
                                                         TREE STATE)
                                     0))
                         (CORRECT TREE STATE))
                ((USE (NO-AT-TERMINATION (SUBTREES (LIST TREE))
                                         (TREE TREE) (STATE STATE)))
                 (DISABLE NO-AT-TERMINATION)))


(DEFN SEND-FIND-FUNC (TO-CHILDREN OLD)
  (IF (LISTP TO-CHILDREN)
      (UPDATE-ASSOC (CAR TO-CHILDREN)
                    (SEND (CAR TO-CHILDREN) 'FIND OLD)
                    (SEND-FIND-FUNC (CDR TO-CHILDREN) OLD))
    OLD))

(DEFN RECEIVE-FIND-FUNC (OLD NODE FROM-PARENT TO-PARENT TO-CHILDREN)
  (IF (EQUAL (HEAD FROM-PARENT OLD) 'FIND)
      (UPDATE-ASSOC
       FROM-PARENT (RECEIVE FROM-PARENT OLD)
       (UPDATE-ASSOC
        (CONS 'STATUS NODE) 'STARTED
        (UPDATE-ASSOC
         (CONS 'FOUND-VALUE NODE) (NODE-VALUE NODE OLD)
         (UPDATE-ASSOC
          (CONS 'OUTSTANDING NODE) (LENGTH TO-CHILDREN)
          (IF (ZEROP (LENGTH TO-CHILDREN))
              (UPDATE-ASSOC
               TO-PARENT (SEND TO-PARENT (NODE-VALUE NODE OLD) OLD)
               (SEND-FIND-FUNC TO-CHILDREN OLD))
            (SEND-FIND-FUNC TO-CHILDREN OLD))))))
    OLD))

(PROVE-LEMMA SEND-FIND-FUNC-IMPLEMENTS-SEND-FIND (REWRITE)
             (SEND-FIND TO-CHILDREN
                        OLD
                        (SEND-FIND-FUNC TO-CHILDREN OLD)))

(PROVE-LEMMA NODES-ARE-NOT-LITATOMS (REWRITE)
             (IMPLIES (AND (ALL-NUMBERPS (NODES-REC FLAG TREE))
                           (MEMBER NODE (NODES-REC FLAG TREE)))
                      (EQUAL (EQUAL (PACK X) NODE) F))
             ((USE (ALL-NUMBERPS-IMPLIES (LIST (NODES-REC FLAG TREE))
                                         (E NODE)))
              (DISABLE ALL-NUMBERPS-IMPLIES)))

(PROVE-LEMMA PARENT-IS-NOT-A-LITATOM (REWRITE)
             (IMPLIES (AND (ALL-NUMBERPS (NODES-REC 'TREE TREE))
                           (SETP (NODES-REC 'TREE TREE))
                           (PROPER-TREE 'TREE TREE)
                           (MEMBER CHILD (CDR (NODES-REC 'TREE TREE))))
                      (EQUAL (EQUAL (PACK X)
                                    (CAR (PARENT-REC 'TREE CHILD TREE)))
                             F))
    ((INSTRUCTIONS PROMOTE (DIVE 1)
         (REWRITE NODES-ARE-NOT-LITATOMS (($FLAG 'TREE) ($TREE TREE)))
         TOP S (REWRITE NODE-HAS-PARENT) (DROP 1 2) (DIVE 2) X TOP
         PROVE (DROP 1) (DEMOTE 1) (DIVE 1) (DIVE 1) X TOP PROVE)))

(PROVE-LEMMA CHILDREN-ARE-NOT-LITATOMS (REWRITE)
             (IMPLIES (AND (ALL-NUMBERPS (NODES-REC FLAG TREE))
                           (PROPER-TREE FLAG TREE)
                           (MEMBER CHILD (CHILDREN-REC FLAG PARENT TREE)))
                      (EQUAL (EQUAL (PACK X) CHILD)
                             F))
    ((INSTRUCTIONS PROMOTE (DIVE 1)
         (REWRITE NODES-ARE-NOT-LITATOMS (($FLAG FLAG) ($TREE TREE)))
```

```
         TOP S (REWRITE MEMBER-CHILD-TREE (($NODE PARENT))))))

(PROVE-LEMMA CHILDREN-ARE-NOT-LITATOMS-MEMBER (REWRITE)
  (IMPLIES (AND (ALL-NUMBERPS (NODES-REC FLAG TREE))
                (PROPER-TREE FLAG TREE))
           (EQUAL (MEMBER (PACK X) (CHILDREN-REC FLAG PARENT TREE))
                  F))
 ((USE (CHILDREN-ARE-NOT-LITATOMS (CHILD (PACK X))))))

(PROVE-LEMMA SEND-FIND-OF-UPDATE-ASSOC (REWRITE)
             (IMPLIES (NOT (MEMBER KEY TO-CHILDREN))
                      (EQUAL (SEND-FIND TO-CHILDREN OLD
                                        (UPDATE-ASSOC KEY VALUE STATE))
                             (SEND-FIND TO-CHILDREN OLD STATE))))

(PROVE-LEMMA ASSOC-OF-SEND-FIND-FUNC (REWRITE)
             (IMPLIES (NOT (MEMBER KEY TO-CHILDREN))
                      (EQUAL (ASSOC KEY (SEND-FIND-FUNC TO-CHILDREN OLD))
                             (ASSOC KEY OLD))))

(PROVE-LEMMA ABOUT-RFP (REWRITE)
             (IMPLIES (NOT (MEMBER P C))
                      (NOT (MEMBER (CONS V P)
                                   (RFP V C)))))

(PROVE-LEMMA ABOUT-RFP-NUMBERP (REWRITE)
             (IMPLIES (NUMBERP A)
                      (NOT (MEMBER (CONS (PACK X)  Y)
                                   (RFP A B)))))

(PROVE-LEMMA PARENT-NOT-IN-RFP (REWRITE)
  (IMPLIES (AND (SETP (NODES-REC 'TREE TREE))
                (PROPER-TREE 'TREE TREE)
                (MEMBER V (CDR (NODES-REC 'TREE TREE))))
           (NOT (MEMBER (CONS V (CAR (PARENT-REC 'TREE V TREE)))
                        (RFP V (CHILDREN-REC 'TREE V TREE)))))
    ((INSTRUCTIONS PROMOTE (DIVE 1) (REWRITE ABOUT-RFP) TOP S (DIVE 1)
         (REWRITE PARENT-IS-NOT-CHILD) TOP S (BASH T))))

(PROVE-LEMMA TO-NODE-NOT-IN-RFP (REWRITE)
             (IMPLIES (NOT (MEMBER NODE CHILDREN))
                      (NOT (MEMBER (CONS X NODE)
                                   (RFP NODE CHILDREN)))))


(PROVE-LEMMA UC-OF-SEND-FIND-FUNC (REWRITE)
             (IMPLIES (SUBLISTP TO-CHILDREN EXCPT)
                      (EQUAL (UC OLD (SEND-FIND-FUNC TO-CHILDREN STATE)
                                 KEYS EXCPT)
                             (UC OLD STATE KEYS EXCPT))))

(PROVE-LEMMA RECEIVE-FIND-FUNC-IMPLEMENTS-RECEIVE-FIND (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER STATEMENT (RECEIVE-FIND-PRG
                                              (CDR (NODES TREE)) TREE)))
                      (N OLD
                         (RECEIVE-FIND-FUNC OLD
                                            (CADR STATEMENT)
                                            (CADDR STATEMENT)
                                            (CADDDR STATEMENT)
                                            (CADDDDR STATEMENT))
                         STATEMENT)))

(DEFN RECEIVE-REPORT-FUNC (OLD NODE FROM-CHILD TO-PARENT)
  (IF (EMPTY FROM-CHILD OLD)
      OLD
    (UPDATE-ASSOC
     FROM-CHILD (RECEIVE FROM-CHILD OLD)
     (UPDATE-ASSOC
```

```
      (CONS 'FOUND-VALUE NODE) (MIN (FOUND-VALUE NODE OLD)
                                    (HEAD FROM-CHILD OLD))
      (UPDATE-ASSOC
       (CONS 'OUTSTANDING NODE) (SUB1 (OUTSTANDING NODE OLD))
       (IF (ZEROP (SUB1 (OUTSTANDING NODE OLD)))
           (UPDATE-ASSOC TO-PARENT
                         (SEND TO-PARENT (MIN (FOUND-VALUE NODE OLD)
                                              (HEAD FROM-CHILD OLD))
                               OLD)
                         OLD)
         OLD))))))

(PROVE-LEMMA RECEIVE-REPORT-FUNC-IMPLEMENTS-RECEIVE-REPORT (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER STATEMENT (RECEIVE-REPORT-PRG
                                              (CDR (NODES TREE)) TREE)))
                      (N OLD
                         (RECEIVE-REPORT-FUNC OLD
                                              (CADR STATEMENT)
                                              (CADDR STATEMENT)
                                              (CADDDR STATEMENT))
                         STATEMENT))
             ((DISABLE MIN)))

(DEFN START-FUNC (OLD ROOT TO-CHILDREN)
  (IF (EQUAL (STATUS ROOT OLD) 'NOT-STARTED)
      (UPDATE-ASSOC
       (CONS 'STATUS ROOT) 'STARTED
       (UPDATE-ASSOC
        (CONS 'FOUND-VALUE ROOT) (NODE-VALUE ROOT OLD)
        (UPDATE-ASSOC
         (CONS 'OUTSTANDING ROOT) (LENGTH TO-CHILDREN)
         (SEND-FIND-FUNC TO-CHILDREN OLD))))
    OLD))

(PROVE-LEMMA START-FUNC-IMPLEMENTS-START (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER STATEMENT (START-PRG (CAR TREE) TREE)))
                      (N OLD
                         (START-FUNC OLD
                                     (CADR STATEMENT)
                                     (CADDR STATEMENT))
                         STATEMENT)))

(DEFN ROOT-RECEIVE-REPORT-FUNC (OLD ROOT FROM-CHILD)
  (IF (EMPTY FROM-CHILD OLD)
      OLD
    (UPDATE-ASSOC
     FROM-CHILD (RECEIVE FROM-CHILD OLD)
     (UPDATE-ASSOC
      (CONS 'FOUND-VALUE ROOT) (MIN (FOUND-VALUE ROOT OLD)
                                    (HEAD FROM-CHILD OLD))
      (UPDATE-ASSOC (CONS 'OUTSTANDING ROOT)
                    (SUB1 (OUTSTANDING ROOT OLD))
                    OLD)))))

(PROVE-LEMMA ROOT-RECEIVE-REPORT-FUNC-IMPLEMENTS-ROOT-RECEIVE-REPORT
             (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER STATEMENT (ROOT-RECEIVE-REPORT-PRG
                                              (CAR TREE) TREE)))
                      (N OLD
                         (ROOT-RECEIVE-REPORT-FUNC OLD
                                                   (CADR STATEMENT)
                                                   (CADDR STATEMENT))
                         STATEMENT))
             ((DISABLE MIN)))

(PROVE-LEMMA RECEIVE-FIND-PRG-IS-TOTAL (REWRITE)
```

```
             (IMPLIES (TREEP TREE)
                      (TOTAL-SUFFICIENT STATEMENT
                                        (RECEIVE-FIND-PRG
                                         (CDR (NODES TREE))
                                         TREE)
                                        OLD
                                        (RECEIVE-FIND-FUNC
                                         OLD
                                         (CADR STATEMENT)
                                         (CADDR STATEMENT)
                                         (CADDDR STATEMENT)
                                         (CADDDDR STATEMENT))))
             ((DISABLE N
                       RECEIVE-FIND
                       RECEIVE-FIND-FUNC
                       MEMBER-RECEIVE-FIND-PRG
                       NODES SETP PROPER-TREE ALL-NUMBERPS)
              (USE (RECEIVE-FIND-FUNC-IMPLEMENTS-RECEIVE-FIND))))

(PROVE-LEMMA RECEIVE-REPORT-PRG-IS-TOTAL (REWRITE)
             (IMPLIES (TREEP TREE)
                      (TOTAL-SUFFICIENT STATEMENT
                                        (RECEIVE-REPORT-PRG
                                         (CDR (NODES TREE))
                                         TREE)
                                        OLD
                                        (RECEIVE-REPORT-FUNC
                                         OLD
                                         (CADR STATEMENT)
                                         (CADDR STATEMENT)
                                         (CADDDR STATEMENT))))
             ((DISABLE N
                       RECEIVE-REPORT
                       RECEIVE-REPORT-FUNC
                       MEMBER-RECEIVE-REPORT-PRG
                       NODES SETP PROPER-TREE ALL-NUMBERPS)
              (USE (RECEIVE-REPORT-FUNC-IMPLEMENTS-RECEIVE-REPORT))))

(PROVE-LEMMA START-PRG-IS-TOTAL (REWRITE)
             (IMPLIES (TREEP TREE)
                      (TOTAL-SUFFICIENT STATEMENT
                                        (START-PRG (CAR TREE) TREE)
                                        OLD
                                        (START-FUNC
                                         OLD
                                         (CADR STATEMENT)
                                         (CADDR STATEMENT))))
             ((DISABLE N
                       START
                       START-FUNC
                       START-PRG
                       NODES SETP PROPER-TREE ALL-NUMBERPS)
              (USE (START-FUNC-IMPLEMENTS-START))))

(PROVE-LEMMA ROOT-RECEIVE-REPORT-PRG-IS-TOTAL (REWRITE)
             (IMPLIES (TREEP TREE)
                      (TOTAL-SUFFICIENT STATEMENT
                                        (ROOT-RECEIVE-REPORT-PRG
                                         (CAR TREE) TREE)
                                        OLD
                                        (ROOT-RECEIVE-REPORT-FUNC
                                         OLD
                                         (CADR STATEMENT)
                                         (CADDR STATEMENT))))
  ((DISABLE N
            ROOT-RECEIVE-REPORT
            ROOT-RECEIVE-REPORT-FUNC
            ROOT-RECEIVE-REPORT-PRG
            MEMBER-ROOT-RECEIVE-REPORT-PRG
```

```
             NODES SETP PROPER-TREE ALL-NUMBERPS)
    (USE (ROOT-RECEIVE-REPORT-FUNC-IMPLEMENTS-ROOT-RECEIVE-REPORT))))

(PROVE-LEMMA TOTAL-TREE-PRG (REWRITE)
             (IMPLIES (TREEP TREE)
                      (TOTAL (TREE-PRG TREE)))
    ((INSTRUCTIONS
         (BASH (DISABLE N ROOT-RECEIVE-REPORT-PRG-IS-TOTAL
                        START-PRG-IS-TOTAL RECEIVE-REPORT-PRG-IS-TOTAL
                        RECEIVE-FIND-PRG-IS-TOTAL
                        MEMBER-ROOT-RECEIVE-REPORT-PRG MEMBER-START-PRG
                        MEMBER-RECEIVE-REPORT-PRG
                        MEMBER-RECEIVE-FIND-PRG
                        ROOT-RECEIVE-REPORT-FUNC START-FUNC
                        RECEIVE-REPORT-FUNC RECEIVE-FIND-FUNC
                        ROOT-RECEIVE-REPORT-PRG START-PRG
                        RECEIVE-REPORT-PRG RECEIVE-FIND-PRG PROPER-TREE
                        ALL-NUMBERPS SETP NODES TREEP MEMBER-TREE-PRG
                        TOTAL-SUFFICIENT))
         PROMOTE (REWRITE HELP-PROVE-TOTAL)
         (PUT (NEW (RECEIVE-REPORT-FUNC
                       (OLDT (RECEIVE-REPORT-PRG (CDR (NODES TREE))
                                 TREE))
                       (CADR (ET (RECEIVE-REPORT-PRG (CDR (NODES TREE))
                                     TREE)))
                       (CADDR (ET (RECEIVE-REPORT-PRG
                                      (CDR (NODES TREE)) TREE)))
                       (CADDDR (ET (RECEIVE-REPORT-PRG
                                       (CDR (NODES TREE)) TREE))))))
         (REWRITE RECEIVE-REPORT-PRG-IS-TOTAL) PROMOTE
         (REWRITE HELP-PROVE-TOTAL)
         (PUT (NEW (RECEIVE-FIND-FUNC
                       (OLDT (RECEIVE-FIND-PRG (CDR (NODES TREE)) TREE))
                       (CADR (ET (RECEIVE-FIND-PRG (CDR (NODES TREE))
                                     TREE)))
                       (CADDR (ET (RECEIVE-FIND-PRG (CDR (NODES TREE))
                                      TREE)))
                       (CADDDR (ET (RECEIVE-FIND-PRG (CDR (NODES TREE))
                                     TREE)))
                       (CADDDDR (ET (RECEIVE-FIND-PRG
                                     (CDR (NODES TREE)) TREE))))))
         (REWRITE RECEIVE-FIND-PRG-IS-TOTAL) PROMOTE
         (REWRITE HELP-PROVE-TOTAL)
         (PUT (NEW (ROOT-RECEIVE-REPORT-FUNC
                       (OLDT (ROOT-RECEIVE-REPORT-PRG (CAR TREE) TREE))
                       (CADR (ET (ROOT-RECEIVE-REPORT-PRG (CAR TREE)
                                     TREE)))
                       (CADDR (ET (ROOT-RECEIVE-REPORT-PRG (CAR TREE)
                                      TREE))))))
         (REWRITE ROOT-RECEIVE-REPORT-PRG-IS-TOTAL) PROMOTE
         (REWRITE HELP-PROVE-TOTAL)
         (PUT (NEW (START-FUNC (OLDT (START-PRG (CAR TREE) TREE))
                       (CADR (ET (START-PRG (CAR TREE) TREE)))
                       (CADDR (ET (START-PRG (CAR TREE) TREE))))))
         (REWRITE START-PRG-IS-TOTAL))))

(PROVE-LEMMA LISTP-TREE-PRG (REWRITE)
             (LISTP (TREE-PRG TREE)))

(PROVE-LEMMA NODE-VALUES-CONSTANT-UNLESS-SUFFICIENT (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER NODE (NODES TREE)))
                      (UNLESS-SUFFICIENT STATEMENT
                                         (TREE-PRG TREE)
                                         OLD NEW
                                         '(EQUAL (NODE-VALUE (QUOTE ,NODE)
                                                             STATE)
                                                 (QUOTE ,K))
                                         '(FALSE)))
```

```
              ((DISABLE TREE-PRG)))


(PROVE-LEMMA NODE-VALUES-CONSTANT-INVARIANT (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION
                 ‘(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS TREE))
                                  STATE)
                       (AND (NOT-STARTED (QUOTE ,(NODES TREE))
                                         STATE)
                            (EQUAL (NODE-VALUE (QUOTE ,NODE) STATE)
                                   (QUOTE ,K))))
                 (TREE-PRG TREE))
                (TREEP TREE)
                (MEMBER NODE (NODES TREE)))
           (INVARIANT ‘(EQUAL (NODE-VALUE (QUOTE ,NODE) STATE)
                              (QUOTE ,K))
                      (TREE-PRG TREE)))
  ((INSTRUCTIONS
    PROMOTE
    (REWRITE UNLESS-PROVES-INVARIANT
             (($IC (LIST ’AND
                         (LIST ’ALL-EMPTY
                               (LIST ’QUOTE (ALL-CHANNELS TREE))
                               ’STATE)
                         (LIST ’AND
                               (LIST ’NOT-STARTED
                                     (LIST ’QUOTE (NODES TREE)) ’STATE)
                               (LIST ’EQUAL
                                     (LIST ’NODE-VALUE
                                           (LIST ’QUOTE NODE) ’STATE)
                                     (LIST ’QUOTE K)))))))
    (REWRITE HELP-PROVE-UNLESS)
    (REWRITE NODE-VALUES-CONSTANT-UNLESS-SUFFICIENT)
    (BASH (DISABLE EVAL TREE-PRG)))))


(PROVE-LEMMA DL-IMPLIES-INSTANCE-OF-DL (REWRITE)
             (IMPLIES (AND (DL DOWN-LINKS STATE)
                           (MEMBER DOWN-LINK DOWN-LINKS))
                      (OR (AND (EMPTY DOWN-LINK STATE)
                               (EQUAL (STATUS (CAR DOWN-LINK) STATE)
                                      (STATUS (CDR DOWN-LINK) STATE)))
                          (AND (EQUAL (CHANNEL DOWN-LINK STATE)
                                      (LIST ’FIND))
                               (EQUAL (STATUS (CAR DOWN-LINK) STATE)
                                      ’STARTED)
                               (EQUAL (STATUS (CDR DOWN-LINK) STATE)
                                      ’NOT-STARTED)))))

(DISABLE DL-IMPLIES-INSTANCE-OF-DL)

(PROVE-LEMMA UL-IMPLIES-INSTANCE-OF-UL (REWRITE)
             (IMPLIES (AND (UL UPLINKS STATE)
                           (MEMBER UPLINK UPLINKS))
                      (OR (EMPTY UPLINK STATE)
                          (AND (EQUAL (CHANNEL UPLINK STATE)
                                      (LIST (FOUND-VALUE (CAR UPLINK)
                                                         STATE)))
                               (DONE (CAR UPLINK) STATE)))))

(DISABLE UL-IMPLIES-INSTANCE-OF-UL)

(PROVE-LEMMA UL-IMPLIES-INSTANCE-OF-UL-NOT-EMPTY-UPLINK (REWRITE)
   (IMPLIES (AND (UL UPLINKS STATE)
                 (MEMBER UPLINK UPLINKS)
                 (NOT (EMPTY UPLINK STATE)))
            (AND (EQUAL (CDR (ASSOC UPLINK STATE))
                        (LIST (FOUND-VALUE (CAR UPLINK) STATE)))
                 (EQUAL (CDR (ASSOC (CONS ’STATUS (CAR UPLINK))
```

```
                                    STATE))
                        'STARTED)
                 (ZEROP (CDR (ASSOC (CONS 'OUTSTANDING (CAR UPLINK))
                                    STATE)))))
   ((USE (UL-IMPLIES-INSTANCE-OF-UL))))


(PROVE-LEMMA NO-IMPLIES-INSTANCE-OF-NO (REWRITE)
   (IMPLIES (AND (NO NODES TREE STATE)
                 (MEMBER NODE NODES)
                 (EQUAL (STATUS NODE STATE) 'STARTED))
            (AND (EQUAL (CDR (ASSOC (CONS 'OUTSTANDING NODE) STATE))
                        (NUMBER-NOT-REPORTED (CHILDREN-REC 'TREE
                                                           NODE TREE)
                                             NODE STATE))
                 (EQUAL (CDR (ASSOC (CONS 'FOUND-VALUE NODE) STATE))
                        (MIN-OF-REPORTED (CHILDREN-REC 'TREE
                                                       NODE TREE)
                                         NODE STATE
                                         (NODE-VALUE NODE STATE))))))

(PROVE-LEMMA MEMBER-DOWN-LINKS-1 (REWRITE)
             (EQUAL (MEMBER DOWN-LINK (DOWN-LINKS-1 PARENT CHILDREN))
                    (AND (EQUAL (CAR DOWN-LINK) PARENT)
                         (MEMBER (CDR DOWN-LINK) CHILDREN)
                         (LISTP DOWN-LINK))))

(PROVE-LEMMA MEMBER-DOWN-LINKS (REWRITE)
             (EQUAL (MEMBER DOWN-LINK (DOWN-LINKS NODES TREE))
                    (AND (MEMBER (CAR DOWN-LINK) NODES)
                         (MEMBER (CDR DOWN-LINK) (CHILDREN (CAR DOWN-LINK)
                                                           TREE))
                         (LISTP DOWN-LINK)))
             ((DISABLE CHILDREN)))

(PROVE-LEMMA PARENT-NOT-CHILD (REWRITE)
  (IMPLIES (AND (PROPER-TREE FLAG TREE)
                (SETP (NODES-REC FLAG TREE)))
           (NOT (MEMBER PARENT (CHILDREN-REC FLAG PARENT TREE)))))

(PROVE-LEMMA PARENT-NOT-GRANDCHILD (REWRITE)
  (IMPLIES (AND (PROPER-TREE FLAG TREE)
                (SETP (NODES-REC FLAG TREE))
                (MEMBER CHILD (CHILDREN-REC FLAG PARENT TREE)))
           (NOT (MEMBER PARENT (CHILDREN-REC FLAG CHILD TREE)))))

(PROVE-LEMMA PARENT-OF-PARENT-NOT-NODE (REWRITE)
   (IMPLIES (AND (PROPER-TREE FLAG TREE)
                 (SETP (NODES-REC FLAG TREE))
                 (LISTP (PARENT-REC FLAG NODE TREE))
                 (LISTP (PARENT-REC FLAG
                                    (CAR (PARENT-REC FLAG NODE TREE))
                                    TREE)))
            (NOT (EQUAL (CAR (PARENT-REC FLAG
                                         (CAR (PARENT-REC FLAG
                                                          NODE TREE))
                                         TREE))
                        NODE)))
    ((INSTRUCTIONS
         (USE-LEMMA PARENT-NOT-GRANDCHILD
             ((CHILD (CAR (PARENT-REC FLAG NODE TREE)))
              (PARENT (CAR (PARENT-REC FLAG
                                  (CAR (PARENT-REC FLAG NODE TREE)) TREE)))))
         PROMOTE (DEMOTE 1) (DIVE 1) (DIVE 1) S
         (REWRITE PARENT-REC-CHILDREN-REC) (DIVE 2)
         (REWRITE SETP-TREE-UNIQUE-PARENT) UP UP (DIVE 2) (DIVE 1)
         (REWRITE PARENT-REC-CHILDREN-REC) (DIVE 2)
         (REWRITE SETP-TREE-UNIQUE-PARENT) TOP PROMOTE PROVE)))
```

```
(PROVE-LEMMA MEMBER-RFP (REWRITE)
             (EQUAL (MEMBER CHANNEL (RFP PARENT CHILDREN))
                    (AND (EQUAL (CAR CHANNEL) PARENT)
                         (MEMBER (CDR CHANNEL) CHILDREN)
                         (LISTP CHANNEL))))

(PROVE-LEMMA SEND-FIND-IMPLIES (REWRITE)
             (IMPLIES (AND (SEND-FIND CHANNELS OLD NEW)
                           (MEMBER KEY CHANNELS))
                      (EQUAL (CDR (ASSOC KEY NEW))
                             (SEND KEY 'FIND OLD))))

(PROVE-LEMMA ASSOC-OF-CHANNEL-PRESERVED-ROOT-RECEIVE-REPORT (REWRITE)
             (IMPLIES (AND (NOT (MEMBER W (NODES-REC 'FOREST D)))
                           (SETP (NODES-REC 'FOREST D))
                           (MEMBER Z (NODES-REC 'FOREST D))
                           (UC NEW OLD
                               (APPEND (STRIP-CARS NEW)
                                       (STRIP-CARS OLD))
                               (LIST (CONS V W)
                                     (CONS 'OUTSTANDING W)
                                     (CONS 'FOUND-VALUE W))))
                      (EQUAL (ASSOC (CONS X Z) NEW)
                             (ASSOC (CONS X Z) OLD)))
             ((USE (ABOUT-UC (A NEW) (B OLD)
                             (EXCPT (LIST (CONS V W)
                                          (CONS 'OUTSTANDING W)
                                          (CONS 'FOUND-VALUE W)))
                             (KEY (CONS X Z))))
              (DISABLE ABOUT-UC)))

(PROVE-LEMMA ASSOC-EQUAL-CONS (REWRITE)
             (EQUAL (EQUAL (ASSOC KEY ALIST) (CONS KEY VALUE))
                    (AND (LISTP (ASSOC KEY ALIST))
                         (EQUAL (CDR (ASSOC KEY ALIST)) VALUE))))

(PROVE-LEMMA SEND-FIND-GENERAL (REWRITE)
             (IMPLIES (AND (SEND-FIND CHANNELS OLD NEW)
                           (MEMBER KEY CHANNELS))
                      (EQUAL (ASSOC KEY NEW)
                             (CONS KEY (SEND KEY 'FIND OLD)))))

(PROVE-LEMMA ALL-NUMBERPS-DO-NOT-CONTAIN-LITATOM (REWRITE)
             (IMPLIES (ALL-NUMBERPS LIST)
                      (NOT (MEMBER (PACK X) LIST))))

(PROVE-LEMMA ALL-NUMBERPS-APPEND (REWRITE)
             (EQUAL (ALL-NUMBERPS (APPEND X Y))
                    (AND (ALL-NUMBERPS X)
                         (ALL-NUMBERPS Y))))

(PROVE-LEMMA ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-PARENT (REWRITE)
             (IMPLIES (ALL-NUMBERPS (NODES-REC FLAG TREE))
                      (ALL-NUMBERPS (PARENT-REC FLAG CHILD TREE))))

(PROVE-LEMMA ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-CAR-PARENT (REWRITE)
             (IMPLIES (ALL-NUMBERPS (NODES-REC FLAG TREE))
                      (NUMBERP (CAR (PARENT-REC FLAG CHILD TREE))))
             ((USE (ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-PARENT))
              (DISABLE ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-PARENT)))

(PROVE-LEMMA PARENT-NOT-LITATOM (REWRITE)
             (IMPLIES (ALL-NUMBERPS (NODES-REC FLAG TREE))
                      (EQUAL (EQUAL (PACK X)
                                    (CAR (PARENT-REC FLAG CHILD TREE)))
                             F))
  ((USE (ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-CAR-PARENT))
   (DISABLE ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-CAR-PARENT)))
```

```
(PROVE-LEMMA ALL-NUMBERPS-FOREST-IMPLIES-ALL-NUMBERPS-ROOTS (REWRITE)
             (IMPLIES (ALL-NUMBERPS (NODES-REC 'FOREST FOREST))
                      (ALL-NUMBERPS (ROOTS FOREST))))

(PROVE-LEMMA ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-CHILDREN (REWRITE)
             (IMPLIES (ALL-NUMBERPS (NODES-REC FLAG TREE))
                      (ALL-NUMBERPS (CHILDREN-REC FLAG PARENT TREE))))


(PROVE-LEMMA DL-PRESERVES-INSTANCE-OF-DL (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (MEMBER DOWN-LINK (DOWN-LINKS (NODES TREE) TREE))
                 (N OLD NEW STATEMENT)
                 (MEMBER STATEMENT (TREE-PRG TREE))
                 (DL (DOWN-LINKS (NODES TREE) TREE) OLD))
            (OR (AND (EMPTY DOWN-LINK NEW)
                     (EQUAL (STATUS (CAR DOWN-LINK) NEW)
                            (STATUS (CDR DOWN-LINK) NEW)))
                (AND (EQUAL (CHANNEL DOWN-LINK NEW)
                            (LIST 'FIND))
                     (EQUAL (STATUS (CAR DOWN-LINK) NEW)
                            'STARTED)
                     (EQUAL (STATUS (CDR DOWN-LINK) NEW)
                            'NOT-STARTED))))
  ((INSTRUCTIONS
    PROMOTE
    (USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
               ((DOWN-LINK DOWN-LINK) (STATE OLD)
                (DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))))
    (DEMOTE 6) (DIVE 1) (DIVE 1) S-PROP UP (S-PROP IMPLIES) TOP
    PROMOTE (CLAIM (EQUAL (CAR STATEMENT) 'START) 0) (DROP 5)
    (BASH T (DISABLE TREE-PRG)) PROMOTE (DIVE 1) (DIVE 1)
    (REWRITE
     ABOUT-UC
     (($B OLD)
      ($EXCPT (CONS (CONS 'STATUS C)
                    (CONS (CONS 'FOUND-VALUE C)
                          (CONS (CONS 'OUTSTANDING C)
                                (RFP C (APPEND (ROOTS W) NIL))))))))
    TOP S PROVE PROMOTE (CLAIM (NOT (EQUAL Z C))) PROVE PROMOTE
    (DIVE 1) (DIVE 1)
    (REWRITE
     ABOUT-UC
     (($B OLD)
      ($EXCPT (CONS (CONS 'STATUS D)
                    (CONS (CONS 'FOUND-VALUE D)
                          (CONS (CONS 'OUTSTANDING D)
                                (RFP D (APPEND (ROOTS W) NIL))))))))
    TOP S PROVE
    (CLAIM (EQUAL (CAR STATEMENT) 'ROOT-RECEIVE-REPORT) 0)
    (DROP 5) (PROVE (DISABLE TREE-PRG))
    (CLAIM (EQUAL (CAR STATEMENT) 'RECEIVE-REPORT) 0) (DROP 5)
    (CLAIM (AND (EQUAL (STATUS (CAR DOWN-LINK) NEW)
                       (STATUS (CAR DOWN-LINK) OLD))
                (EQUAL (STATUS (CDR DOWN-LINK) NEW)
                       (STATUS (CDR DOWN-LINK) OLD))
                (EQUAL (CHANNEL DOWN-LINK NEW)
                       (CHANNEL DOWN-LINK OLD)))
           0)
    (DROP 1 2 3 4 6 7 8) PROVE (CONTRADICT 9) (DROP 5 6 7 9) SPLIT
    (PROVE (DISABLE TREE-PRG)) (PROVE (DISABLE TREE-PRG))
    (CLAIM (AND (NOT (EQUAL DOWN-LINK (CADDR STATEMENT)))
                (NOT (EQUAL DOWN-LINK (CADDDR STATEMENT))))
           0)
    (PROVE (DISABLE TREE-PRG)) (CONTRADICT 6) (DROP 3 6) SPLIT
    (PROVE (DISABLE TREE-PRG)) (BASH T (DISABLE TREE-PRG)) PROMOTE
    (CONTRADICT 4) (DIVE 1) (REWRITE PARENT-NOT-GRANDCHILD) TOP S
    (REWRITE PARENT-REC-CHILDREN-REC) (DIVE 2)
    (REWRITE SETP-TREE-UNIQUE-PARENT) TOP PROVE
```

```
(CLAIM (EQUAL DOWN-LINK (CADDDR STATEMENT)) 0) (CONTRADICT 10)
(DROP 3 5 6 10) (BASH T (DISABLE TREE-PRG)) PROMOTE
(CONTRADICT 4) (DIVE 1) (REWRITE PARENT-NOT-GRANDCHILD) TOP S
(REWRITE PARENT-REC-CHILDREN-REC) (DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT) TOP PROVE
(CLAIM (EQUAL DOWN-LINK (CADDR STATEMENT)) 0)
(CLAIM (MEMBER DOWN-LINK (CADDDDR STATEMENT)) 0)
(CONTRADICT 12) (DROP 3 5 6 10 12) (BASH T (DISABLE TREE-PRG))
(DROP 5) (BASH T (DISABLE TREE-PRG))
(CLAIM (MEMBER DOWN-LINK (CADDDDR STATEMENT)) 0)
(USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
           ((DOWN-LINK
             (CONS (PARENT (CAR DOWN-LINK) TREE) (CAR DOWN-LINK)))
            (STATE OLD) (DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))))
(DEMOTE 13) (DIVE 1) (DIVE 1) (= * T IFF) TOP (DROP 5)
(BASH T (DISABLE TREE-PRG)) PROMOTE (CLAIM (NOT (EQUAL Z C)))
PROVE S-PROP
(CLAIM (MEMBER (CAR DOWN-LINK) (CDR (NODES TREE))) 0)
(DROP 3 4 5 6 7 8 9 10 11 12) (REWRITE MEMBER-DOWN-LINKS) S
(DIVE 1) (REWRITE NODE-HAS-PARENT) TOP S
(REWRITE PARENT-REC-CHILDREN-REC) (DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT) TOP PROVE PROVE PROVE PROVE
PROVE PROVE (CONTRADICT 13) (DROP 3 5 6 10 11 13) PROVE
(CLAIM (AND (EQUAL (STATUS (CAR DOWN-LINK) NEW)
                   (STATUS (CAR DOWN-LINK) OLD))
            (EQUAL (STATUS (CDR DOWN-LINK) NEW)
                   (STATUS (CDR DOWN-LINK) OLD))
            (EQUAL (CHANNEL DOWN-LINK NEW)
                   (CHANNEL DOWN-LINK OLD)))
       0)
(DROP 1 2 3 4 5 7 8 9 10 11 12) PROVE (CONTRADICT 13)
(DROP 5 6 13) (DEMOTE 3) (DIVE 1)
(= *
   (RECEIVE-FIND OLD NEW (CADR STATEMENT) (CADDR STATEMENT)
                 (CADDDR STATEMENT) (CADDDDR STATEMENT))
   ((DISABLE TREE-PRG MEMBER-DOWN-LINKS)))
X-DUMB TOP PROMOTE
(CLAIM (EQUAL (HEAD (CADDR STATEMENT) OLD) 'FIND) 0) (DIVE 1)
(DIVE 1) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CADDR STATEMENT)
                                (CADDDR STATEMENT)
                                (CONS 'STATUS (CADR STATEMENT))
                                (CONS 'FOUND-VALUE
                                      (CADR STATEMENT))
                                (CONS 'OUTSTANDING
                                      (CADR STATEMENT)))
                          (CADDDDR STATEMENT)))))
TOP (DIVE 2) (DIVE 1) (DIVE 1) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CADDR STATEMENT)
                                (CADDDR STATEMENT)
                                (CONS 'STATUS (CADR STATEMENT))
                                (CONS 'FOUND-VALUE
                                      (CADR STATEMENT))
                                (CONS 'OUTSTANDING
                                      (CADR STATEMENT)))
                          (CADDDDR STATEMENT)))))
TOP (DIVE 2) (DIVE 2) (DIVE 1) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CADDR STATEMENT)
                                (CADDDR STATEMENT)
                                (CONS 'STATUS (CADR STATEMENT))
                                (CONS 'FOUND-VALUE
                                      (CADR STATEMENT))
                                (CONS 'OUTSTANDING
```

```
                                       (CADR STATEMENT)))
                             (CADDDDR STATEMENT)))))
    TOP S (BASH T (DISABLE TREE-PRG)) (DROP 3 4 5 6 10 11) PROVE
    (BASH T (DISABLE TREE-PRG))
    (CLAIM (NOT (EQUAL (CDR DOWN-LINK) (CADR STATEMENT))))
    (DROP 10 11) PROVE PROVE (DROP 10 11)
    (CLAIM (NOT (EQUAL (CAR DOWN-LINK) (CADR STATEMENT))))
    (PROVE (DISABLE TREE-PRG)) (DROP 1 2 3 4 5 6 7 8 9) PROVE)))

(PROVE-LEMMA DL-PRESERVES-SUBLIST (REWRITE)
  (IMPLIES (AND (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                (TREEP TREE)
                (N OLD NEW STATEMENT)
                (MEMBER STATEMENT (TREE-PRG TREE))
                (SUBLISTP SUBLIST (DOWN-LINKS (NODES TREE) TREE)))
           (DL SUBLIST NEW))
    ((INSTRUCTIONS (INDUCT (DL SUBLIST NEW)) PROMOTE PROMOTE (DEMOTE 2)
         (DIVE 1) (DIVE 1)
         (= * T ((DISABLE N TREE-PRG MEMBER-TREE-PRG))) UP S TOP
         PROMOTE X (DROP 7)
         (USE-LEMMA DL-PRESERVES-INSTANCE-OF-DL
             ((TREE TREE) (DOWN-LINK (CAR SUBLIST))
              (STATEMENT STATEMENT) (OLD OLD) (NEW NEW)))
         (DEMOTE 7) (DIVE 1) (DIVE 1)
         (= * T ((DISABLE TREEP N TREE-PRG MEMBER-TREE-PRG NODES))) TOP
         S PROVE)))

(PROVE-LEMMA DL-PRESERVES-DL (REWRITE)
             (IMPLIES (AND (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                           (TREEP TREE)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (TREE-PRG TREE)))
                      (DL (DOWN-LINKS (NODES TREE) TREE) NEW))
             ((USE (DL-PRESERVES-SUBLIST
                    (SUBLIST (DOWN-LINKS (NODES TREE) TREE))))
              (DISABLE TREE-PRG MEMBER-TREE-PRG)))

(PROVE-LEMMA MEMBER-UP-LINKS (REWRITE)
             (EQUAL (MEMBER UP-LINK (UP-LINKS NODES TREE))
                    (AND (MEMBER (CAR UP-LINK) NODES)
                         (EQUAL (CDR UP-LINK) (PARENT (CAR UP-LINK) TREE))
                         (LISTP UP-LINK))))

(PROVE-LEMMA ZERO-NOT-REPORTED-IMPLIES-CHILDREN-REPORTED (REWRITE)
  (IMPLIES (AND (ZEROP (NUMBER-NOT-REPORTED CHILDREN PARENT STATE))
                (MEMBER CHILD CHILDREN))
           (AND (EQUAL (CDR (ASSOC (CONS 'STATUS CHILD) STATE))
                       'STARTED)
                (ZEROP (OUTSTANDING CHILD STATE))
                (NOT (LISTP (CDR (ASSOC (CONS CHILD PARENT)
                                        STATE))))))
  ((INDUCT (MEMBER CHILD CHILDREN))))


(PROVE-LEMMA DL-UL-NO-PRESERVES-INSTANCE-OF-UL (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (MEMBER UP-LINK (UP-LINKS (CDR (NODES TREE)) TREE))
                 (N OLD NEW STATEMENT)
                 (MEMBER STATEMENT (TREE-PRG TREE))
                 (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                 (UL (UP-LINKS (CDR (NODES TREE)) TREE) OLD)
                 (NO (NODES TREE) TREE OLD))
            (OR (EMPTY UP-LINK NEW)
                (AND (EQUAL (CHANNEL UP-LINK NEW)
                            (LIST (FOUND-VALUE (CAR UP-LINK) NEW)))
                     (DONE (CAR UP-LINK) NEW))))
((INSTRUCTIONS PROMOTE
  (USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL
             ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))
```

```
            (UPLINK UP-LINK)
            (STATE OLD)))
(DEMOTE 8)
(DIVE 1)
(DIVE 1)
S-PROP UP
(S-PROP IMPLIES)
TOP PROMOTE
(CLAIM (EQUAL (CAR STATEMENT) 'START)
       0)
(DROP 5 6 7)
(CLAIM (AND (EQUAL (FOUND-VALUE (CAR UP-LINK) NEW)
                   (FOUND-VALUE (CAR UP-LINK) OLD))
            (EQUAL (CHANNEL UP-LINK NEW)
                   (CHANNEL UP-LINK OLD))
            (EQUAL (STATUS (CAR UP-LINK) NEW)
                   (STATUS (CAR UP-LINK) OLD))
            (EQUAL (OUTSTANDING (CAR UP-LINK) NEW)
                   (OUTSTANDING (CAR UP-LINK) OLD)))
       0)
(DROP 1 2 3 4 6)
PROVE
(CONTRADICT 7)
(DROP 5 7)
(BASH T (DISABLE TREE-PRG))
PROMOTE
(CLAIM (NOT (EQUAL X D)))
PROVE PROMOTE
(CLAIM (NOT (EQUAL X D)))
PROVE PROMOTE
(CLAIM (NOT (MEMBER (CONS X D)
                    (RFP D (APPEND (ROOTS V) NIL)))))
PROVE PROMOTE
(CLAIM (NOT (MEMBER (CONS X
                          (CAR (PARENT-REC 'FOREST X V)))
                    (RFP D (APPEND (ROOTS V) NIL)))))
PROVE PROMOTE
(CLAIM (NOT (EQUAL X D)))
PROVE
(CLAIM (EQUAL (CAR STATEMENT)
              'ROOT-RECEIVE-REPORT)
       0)
(CLAIM (NOT (EQUAL (CAR UP-LINK) (CAR TREE)))
       0)
CHANGE-GOAL
(CONTRADICT 11)
(DROP 3 4 5 6 7 8 9 10 11)
PROVE
(CLAIM (EQUAL UP-LINK (CADDR STATEMENT))
       0)
(DROP 5 6 7 9)
(BASH T (DISABLE TREE-PRG))
(CLAIM (AND (EQUAL (FOUND-VALUE (CAR UP-LINK) NEW)
                   (FOUND-VALUE (CAR UP-LINK) OLD))
            (EQUAL (CHANNEL UP-LINK NEW)
                   (CHANNEL UP-LINK OLD))
            (EQUAL (STATUS (CAR UP-LINK) NEW)
                   (STATUS (CAR UP-LINK) OLD))
            (EQUAL (OUTSTANDING (CAR UP-LINK) NEW)
                   (OUTSTANDING (CAR UP-LINK) OLD)))
       0)
(DROP 1 2 3 4 5 6 7 9 10 11 12)
PROVE
(CONTRADICT 13)
(DROP 5 6 7 8 9 13)
(BASH T (DISABLE TREE-PRG))
(CLAIM (EQUAL (CAR STATEMENT) 'RECEIVE-FIND)
       0)
(CLAIM (EQUAL (CADR STATEMENT) (CAR UP-LINK))
```

```
       0)
(USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
           ((STATE OLD)
            (DOWN-LINK (CONS (PARENT (CAR UP-LINK) TREE)
                             (CAR UP-LINK)))
            (DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))))
(DEMOTE 13)
(DIVE 1)
(DIVE 1)
S-PROP
(REWRITE MEMBER-DOWN-LINKS)
S
(DIVE 1)
(= * T IFF)
UP S
(REWRITE PARENT-REC-CHILDREN-REC)
(DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT)
UP S
(= * T IFF)
CHANGE-GOAL
(DROP 3 4 5 6 7 8 9 10 11 12)
S
(DIVE 2)
(DIVE 1)
(= T)
UP S
(DIVE 1)
(= F)
UP S TOP X CHANGE-GOAL
(DEMOTE 1)
S CHANGE-GOAL
(DEMOTE 1)
S CHANGE-GOAL
(DROP 3 4 5 6 7 8 9 10 11 12)
S
(REWRITE NODE-HAS-PARENT)
PROVE DEMOTE S S PROVE UP
(S-PROP IMPLIES)
TOP PROMOTE
(DROP 5 6 7 9 10)
(BASH T (DISABLE TREE-PRG))
(CLAIM (AND (EQUAL (FOUND-VALUE (CAR UP-LINK) NEW)
                   (FOUND-VALUE (CAR UP-LINK) OLD))
            (EQUAL (CHANNEL UP-LINK NEW)
                   (CHANNEL UP-LINK OLD))
            (EQUAL (STATUS (CAR UP-LINK) NEW)
                   (STATUS (CAR UP-LINK) OLD))
            (EQUAL (OUTSTANDING (CAR UP-LINK) NEW)
                   (OUTSTANDING (CAR UP-LINK) OLD)))
       0)
(DROP 1 2 3 4 5 6 7 9 10 11 12)
PROVE
(CONTRADICT 13)
(DROP 5 6 7 8 9 10 13)
(BASH T (DISABLE TREE-PRG))
PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
    (CONS (CAR (PARENT-REC 'TREE D TREE))
          D)
    (CONS
       (CONS D
             (CAR (PARENT-REC 'TREE D TREE)))
       (CONS (CONS 'STATUS D)
```

```
             (CONS (CONS 'FOUND-VALUE D)
                   (CONS (CONS 'OUTSTANDING D)
                         (RFP D
                              (CHILDREN-REC 'TREE D TREE))))))))))
TOP S
(DROP 7 8 9 10 11 12 14 15)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
    (CONS (CAR (PARENT-REC 'TREE D TREE))
          D)
    (CONS
       (CONS D
             (CAR (PARENT-REC 'TREE D TREE)))
       (CONS (CONS 'STATUS D)
             (CONS (CONS 'FOUND-VALUE D)
                   (CONS (CONS 'OUTSTANDING D)
                         (RFP D
                              (CHILDREN-REC 'TREE D TREE))))))))))
TOP S
(DROP 7 8 9 10 11 12 14 15)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
    (CONS (CAR (PARENT-REC 'TREE D TREE))
          D)
    (CONS
       (CONS D
             (CAR (PARENT-REC 'TREE D TREE)))
       (CONS (CONS 'STATUS D)
             (CONS (CONS 'FOUND-VALUE D)
                   (CONS (CONS 'OUTSTANDING D)
                         (RFP D
                              (CHILDREN-REC 'TREE D TREE))))))))))
TOP S
(DROP 7 8 9 10 11 12 14 15)
(BASH T)
PROMOTE
(DIVE 1)
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S PROVE
(REWRITE LISTP-PARENT-REC-EQUALS)
PROVE PROVE
(REWRITE LISTP-PARENT-REC-EQUALS)
PROVE PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
    (CONS (CAR (PARENT-REC 'TREE D TREE))
          D)
    (CONS
       (CONS D
             (CAR (PARENT-REC 'TREE D TREE)))
       (CONS (CONS 'STATUS D)
             (CONS (CONS 'FOUND-VALUE D)
                   (CONS (CONS 'OUTSTANDING D)
                         (RFP D
                              (CHILDREN-REC 'TREE D TREE))))))))))
```

```
TOP S
(DROP 7 8 9 10 11 12 14 15)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
    (CONS (CAR (PARENT-REC 'TREE D TREE))
          D)
    (CONS
       (CONS D
             (CAR (PARENT-REC 'TREE D TREE)))
       (CONS (CONS 'STATUS D)
             (CONS (CONS 'FOUND-VALUE D)
                   (CONS (CONS 'OUTSTANDING D)
                         (RFP D
                              (CHILDREN-REC 'TREE D TREE)))))))))
TOP S
(DROP 7 8 9 10 11 13 14)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
    (CONS (CAR (PARENT-REC 'TREE D TREE))
          D)
    (CONS
       (CONS D
             (CAR (PARENT-REC 'TREE D TREE)))
       (CONS (CONS 'STATUS D)
             (CONS (CONS 'FOUND-VALUE D)
                   (CONS (CONS 'OUTSTANDING D)
                         (RFP D
                              (CHILDREN-REC 'TREE D TREE)))))))))
TOP S
(DROP 7 8 9 10 11 13 14)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
    (CONS (CAR (PARENT-REC 'TREE D TREE))
          D)
    (CONS
       (CONS D
             (CAR (PARENT-REC 'TREE D TREE)))
       (CONS (CONS 'STATUS D)
             (CONS (CONS 'FOUND-VALUE D)
                   (CONS (CONS 'OUTSTANDING D)
                         (RFP D
                              (CHILDREN-REC 'TREE D TREE)))))))))
TOP S
(DROP 7 8 9 10 11 13 14)
(BASH T)
PROMOTE
(CLAIM (SETP (NODES-REC 'TREE TREE)))
(DIVE 1)
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S PROVE PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
```

```
  ($EXCPT
   (CONS
    (CONS (CAR (PARENT-REC ’TREE D TREE))
          D)
    (CONS
       (CONS D
             (CAR (PARENT-REC ’TREE D TREE)))
       (CONS (CONS ’STATUS D)
             (CONS (CONS ’FOUND-VALUE D)
                   (CONS (CONS ’OUTSTANDING D)
                         (RFP D
                              (CHILDREN-REC ’TREE D TREE)))))))))
TOP S
(DROP 7 8 9 10 11 13 14)
PROVE
(CLAIM (EQUAL UP-LINK (CADDR STATEMENT))
       0)
(CLAIM (NOT (EQUAL (CADDR STATEMENT)
                   (CADDDR STATEMENT)))
       0)
(DROP 5 6 7)
(BASH T (DISABLE TREE-PRG))
(CONTRADICT 13)
(DROP 3 5 6 7 8 13)
(PROVE (DISABLE TREE-PRG))
(CLAIM (EQUAL UP-LINK (CADDDR STATEMENT))
       0)
(CLAIM (EMPTY (CADDR STATEMENT) OLD)
       0)
(CLAIM (CHANGED OLD NEW NIL) 0)
(DROP 1 2 3 4 5 6 7 9 10 11 12 13 14)
PROVE
(CONTRADICT 15)
(DROP 1 2 5 6 7 8 12 13 15)
(PROVE (DISABLE TREE-PRG))
(CLAIM (EMPTY UP-LINK OLD) 0)
(CLAIM (EQUAL (STATUS (CADR STATEMENT) OLD)
              ’STARTED)
       0)
(DROP 5 6 7)
(BASH T (DISABLE TREE-PRG MIN))
PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS Z1 W)
                        (CONS W
                              (CAR (PARENT-REC ’TREE W TREE)))
                        (CONS ’OUTSTANDING W)
                        (CONS ’FOUND-VALUE W)))))
TOP S
(DROP 6 7 8 9 10 11 14 15 16)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS Z1 W)
                        (CONS W
                              (CAR (PARENT-REC ’TREE W TREE)))
                        (CONS ’OUTSTANDING W)
                        (CONS ’FOUND-VALUE W)))))
TOP S
(DROP 6 7 8 9 10 11 14 15 16)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
```

```
          (($B OLD)
           ($EXCPT (LIST (CONS Z1 W)
                         (CONS W
                               (CAR (PARENT-REC 'TREE W TREE)))
                         (CONS 'OUTSTANDING W)
                         (CONS 'FOUND-VALUE W)))))
TOP S
(DROP 6 7 8 9 10 11 14 15 16)
PROVE
(USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL
           ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))
            (STATE OLD)
            (UPLINK (CADDR STATEMENT))))
(DEMOTE 17)
(DIVE 1)
(DIVE 1)
S-PROP
(= * T IFF)
UP
(S-PROP IMPLIES)
(S-PROP OR)
UP PROMOTE
(USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
           ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))
            (STATE OLD)
            (DOWN-LINK (CONS (CADR STATEMENT)
                             (CAADDR STATEMENT)))))
(DEMOTE 19)
(DIVE 1)
(DIVE 1)
S-PROP
(= * T IFF)
UP
(S-PROP IMPLIES)
TOP PROMOTE
(CONTRADICT 16)
(DROP 1 2 3 4 5 6 7 8 9 10 11 16)
PROVE
(DROP 3 5 6 7 8 14 15 16 17 18)
(BASH T (DISABLE TREE-PRG))
(DROP 3 5 6 7 8 14 15 16)
(BASH T (DISABLE TREE-PRG))
PROMOTE
(CONTRADICT 7)
(DIVE 1)
(REWRITE PARENT-REC-CHILDREN-REC)
(DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT)
TOP PROVE PROVE
(USE-LEMMA NO-IMPLIES-INSTANCE-OF-NO
           ((NODES (NODES TREE))
            (TREE TREE)
            (STATE OLD)
            (NODE (CADR STATEMENT))))
(DEMOTE 16)
(DIVE 1)
(DIVE 1)
(= * T IFF)
UP
(S-PROP IMPLIES)
TOP PROMOTE
(USE-LEMMA ZERO-NOT-REPORTED-IMPLIES-CHILDREN-REPORTED
           ((CHILDREN (CHILDREN (CADR STATEMENT) TREE))
            (PARENT (CADR STATEMENT))
            (STATE OLD)
            (CHILD (CAADDR STATEMENT))))
(CONTRADICT 14)
(DROP 3 5 6 7 14)
(BASH T (DISABLE TREE-PRG))
```

```
  S-PROP
  (CLAIM (EQUAL (CADR STATEMENT) (CAR UP-LINK))
         ((DISABLE TREE-PRG)))
  (DROP 3 4 5 6 7 9 10 11 12 13 14)
  PROVE
  (CLAIM (AND (EQUAL (CHANNEL UP-LINK NEW)
                     (CHANNEL UP-LINK OLD))
              (EQUAL (STATUS (CAR UP-LINK) NEW)
                     (STATUS (CAR UP-LINK) OLD))
              (EQUAL (FOUND-VALUE (CAR UP-LINK) NEW)
                     (FOUND-VALUE (CAR UP-LINK) OLD))
              (EQUAL (OUTSTANDING (CAR UP-LINK) NEW)
                     (OUTSTANDING (CAR UP-LINK) OLD)))
         0)
  (DROP 1 2 3 4 5 6 7 9 10 11 12 13)
  PROVE
  (CONTRADICT 14)
  (CLAIM (NOT (EQUAL (CAR UP-LINK)
                     (CADR STATEMENT)))
         ((DISABLE TREE-PRG)))
  (DROP 5 6 7 8 14)
  (CLAIM (NUMBERP (CAR UP-LINK)))
  (DROP 2)
  (BASH T (DISABLE TREE-PRG MIN)))))


(PROVE-LEMMA DL-UL-NO-PRESERVES-UL-SUBLIST (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (TREE-PRG TREE))
                           (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                           (UL (UP-LINKS (CDR (NODES TREE)) TREE) OLD)
                           (NO (NODES TREE) TREE OLD)
                           (SUBLISTP SUBLIST (UP-LINKS (CDR (NODES TREE))
                                                       TREE)))
                      (UL SUBLIST NEW))
    ((INSTRUCTIONS (INDUCT (UL SUBLIST NEW)) PROMOTE PROMOTE (DEMOTE 2)
         (DIVE 1) (DIVE 1) S-PROP
         (= * T ((DISABLE N TREE-PRG MEMBER-TREE-PRG TREEP NODES))) UP
         S TOP PROMOTE X (DROP 9)
         (USE-LEMMA DL-UL-NO-PRESERVES-INSTANCE-OF-UL
             ((TREE TREE) (UP-LINK (CAR SUBLIST)) (STATEMENT STATEMENT)
              (OLD OLD) (NEW NEW)))
         (DEMOTE 9) (DIVE 1) (DIVE 1) S-PROP
         (= * T ((DISABLE N TREE-PRG MEMBER-TREE-PRG TREEP NODES))) TOP
         S PROMOTE PROMOTE X)))


(PROVE-LEMMA DL-UL-NO-PRESERVES-UL (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (TREE-PRG TREE))
                           (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                           (UL (UP-LINKS (CDR (NODES TREE)) TREE) OLD)
                           (NO (NODES TREE) TREE OLD))
                      (UL (UP-LINKS (CDR (NODES TREE)) TREE) NEW))
             ((USE (DL-UL-NO-PRESERVES-UL-SUBLIST
                    (SUBLIST (UP-LINKS (CDR (NODES TREE)) TREE))))
              (DISABLE TREEP TREE-PRG MEMBER-TREE-PRG MEMBER-UP-LINKS
                       NODES N)))

(PROVE-LEMMA PARENT-NOT-STARTED-IMPLIES-ALL-EMPTY-AND-NOT-STARTED
 (REWRITE)
 (IMPLIES (AND (EQUAL (STATUS PARENT STATE) 'NOT-STARTED)
               (DL (RFP PARENT CHILDREN) STATE))
          (AND (ALL-EMPTY (RFP PARENT CHILDREN) STATE)
               (NOT-STARTED CHILDREN STATE))))
```

```
(PROVE-LEMMA START-PRESERVES-NO-FOR-PARENT (REWRITE)
  (IMPLIES (AND (NUMBERP PARENT)
                (NOT (MEMBER PARENT CHILDREN))
                (NOT-STARTED CHILDREN OLD)
                (SUBLISTP (RFP PARENT CHILDREN)
                          (RFP PARENT EXCPT))
                (CHANGED OLD NEW
                         (APPEND (LIST (CONS 'STATUS PARENT)
                                       (CONS 'FOUND-VALUE PARENT)
                                       (CONS 'OUTSTANDING PARENT))
                                 (RFP PARENT EXCPT))))
           (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT NEW)
                       (LENGTH CHILDREN))
                (EQUAL (MIN-OF-REPORTED CHILDREN PARENT
                                        NEW VALUE)
                       VALUE)))
    ((INSTRUCTIONS (INDUCT (LENGTH CHILDREN))
         (CLAIM (EQUAL (CDR (ASSOC (CONS 'STATUS (CAR CHILDREN)) NEW))
                       'STARTED)
                0)
         PROMOTE PROMOTE (CONTRADICT 1) (DIVE 1) (DIVE 1) (DIVE 1)
         (REWRITE ABOUT-UC
                  (($B OLD)
                   ($EXCPT (APPEND (LIST (CONS 'STATUS PARENT)
                                         (CONS 'FOUND-VALUE PARENT)
                                         (CONS 'OUTSTANDING PARENT))
                                   (RFP PARENT EXCPT)))))
         TOP PROVE PROVE PROVE PROVE PROVE)))

(PROVE-LEMMA UNCHANGED-PRESERVES-NO (REWRITE)
   (IMPLIES (CHANGED OLD NEW NIL)
            (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT NEW)
                        (NUMBER-NOT-REPORTED CHILDREN PARENT OLD))
                 (EQUAL (MIN-OF-REPORTED CHILDREN PARENT NEW VALUE)
                        (MIN-OF-REPORTED CHILDREN PARENT
                                         OLD VALUE)))))

(PROVE-LEMMA START-PRESERVES-NO-FOR-REST-OF-TREE (REWRITE)
 (IMPLIES (AND (NUMBERP ROOT)
               (NUMBERP PARENT)
               (NOT (MEMBER PARENT CHILDREN))
               (NOT (MEMBER ROOT CHILDREN))
               (NOT (EQUAL ROOT PARENT))
               (CHANGED OLD NEW
                        (APPEND (LIST (CONS 'STATUS ROOT)
                                      (CONS 'FOUND-VALUE ROOT)
                                      (CONS 'OUTSTANDING ROOT))
                                (RFP ROOT EXCPT))))
          (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT NEW)
                      (NUMBER-NOT-REPORTED CHILDREN PARENT OLD))
               (EQUAL (MIN-OF-REPORTED CHILDREN PARENT NEW VALUE)
                      (MIN-OF-REPORTED CHILDREN PARENT
                                       OLD VALUE)))))

(PROVE-LEMMA LENGTH-RFP (REWRITE)
             (EQUAL (LENGTH (RFP PARENT CHILDREN))
                    (LENGTH CHILDREN)))

(PROVE-LEMMA START-PRESERVES-INSTANCE-OF-NO (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (START OLD NEW (CAR TREE)
                        (RFP (CAR TREE)
                             (CHILDREN (CAR TREE) TREE)))
                 (MEMBER NODE (NODES TREE))
                 (DL (RFP (CAR TREE) (CHILDREN (CAR TREE) TREE))
                     OLD)
                 (EQUAL (STATUS NODE NEW) 'STARTED)
                 (IMPLIES (EQUAL (STATUS NODE OLD) 'STARTED)
                          (AND (EQUAL (OUTSTANDING NODE OLD)
```

```
                                  (NUMBER-NOT-REPORTED
                                   (CHILDREN NODE TREE)
                                   NODE OLD))
                           (EQUAL (FOUND-VALUE NODE OLD)
                                  (MIN-OF-REPORTED
                                   (CHILDREN NODE TREE)
                                   NODE OLD
                                   (NODE-VALUE NODE OLD))))))
        (AND (EQUAL (OUTSTANDING NODE NEW)
                    (NUMBER-NOT-REPORTED (CHILDREN NODE TREE)
                                         NODE NEW))
             (EQUAL (FOUND-VALUE NODE NEW)
                    (MIN-OF-REPORTED (CHILDREN NODE TREE)
                                     NODE NEW
                                     (NODE-VALUE NODE NEW)))))
((INSTRUCTIONS PROMOTE (DEMOTE 2) (DIVE 1) X-DUMB
     (CLAIM (EQUAL (STATUS (CAR TREE) OLD) 'NOT-STARTED) 0)
     (DIVE 1) (DIVE 1) = UP UP S-PROP TOP PROMOTE
     (CLAIM (EQUAL NODE (CAR TREE)) 0) (CLAIM (NUMBERP NODE) 0)
     (CLAIM (NOT (MEMBER NODE (CHILDREN NODE TREE))) 0)
     (CLAIM (NOT-STARTED (CHILDREN NODE TREE) OLD) 0)
     (CLAIM (SUBLISTP (RFP NODE (CHILDREN NODE TREE))
                (RFP NODE (CHILDREN (CAR TREE) TREE)))
            0)
     (DIVE 1) (DIVE 2)
     (CLAIM (CHANGED OLD NEW
                     (APPEND (LIST (CONS 'STATUS NODE)
                                   (CONS 'FOUND-VALUE NODE)
                                   (CONS 'OUTSTANDING NODE))
                             (RFP NODE (CHILDREN (CAR TREE) TREE))))
            0)
     (REWRITE START-PRESERVES-NO-FOR-PARENT
             (($OLD OLD) ($EXCPT (CHILDREN (CAR TREE) TREE))))
     TOP (DIVE 2) (DIVE 2)
     (REWRITE START-PRESERVES-NO-FOR-PARENT
             (($OLD OLD) ($EXCPT (CHILDREN (CAR TREE) TREE))))
     TOP (DROP 3 9 10 11 12 13) PROVE TOP (CONTRADICT 13)
     (DROP 1 2 3 4 5 6 9 10 11 12 13) PROVE (CONTRADICT 12)
     (DIVE 2) (DIVE 2) (DIVE 1) = TOP (REWRITE SUBLISTP-REFLEXIVE)
     (CONTRADICT 11)
     (REWRITE PARENT-NOT-STARTED-IMPLIES-ALL-EMPTY-AND-NOT-STARTED
             (($PARENT (CAR TREE))))
     (DIVE 1) (DIVE 2) (DIVE 1) = TOP S-PROP (CONTRADICT 10)
     (DROP 3 4 5 6 7 8 9 10) PROVE (CONTRADICT 9)
     (DROP 2 3 4 5 6 7 9) PROVE (CLAIM (NUMBERP (CAR TREE)) 0)
     (CLAIM (NUMBERP NODE) 0)
     (CLAIM (NOT (MEMBER NODE (CHILDREN NODE TREE))) 0)
     (CLAIM (NOT (MEMBER (CAR TREE) (CHILDREN NODE TREE))) 0)
     (DIVE 1) (DIVE 2)
     (CLAIM (CHANGED OLD NEW
                     (APPEND (LIST (CONS 'STATUS (CAR TREE))
                                   (CONS 'FOUND-VALUE (CAR TREE))
                                   (CONS 'OUTSTANDING (CAR TREE)))
                             (RFP (CAR TREE)
                                  (CHILDREN (CAR TREE) TREE))))
            0)
     (REWRITE START-PRESERVES-NO-FOR-REST-OF-TREE
             (($OLD OLD) ($ROOT (CAR TREE))
              ($EXCPT (CHILDREN (CAR TREE) TREE))))
     TOP (DIVE 2) (DIVE 2)
     (REWRITE START-PRESERVES-NO-FOR-REST-OF-TREE
             (($OLD OLD) ($ROOT (CAR TREE))
              ($EXCPT (CHILDREN (CAR TREE) TREE))))
     TOP
     (CLAIM (AND (EQUAL (STATUS NODE NEW) (STATUS NODE OLD))
                 (EQUAL (OUTSTANDING NODE NEW)
                        (OUTSTANDING NODE OLD))
                 (EQUAL (FOUND-VALUE NODE NEW)
                        (FOUND-VALUE NODE OLD))
```

```
                      (EQUAL (NODE-VALUE NODE NEW)
                             (NODE-VALUE NODE OLD)))
                 0)
          (DROP 1 2 3 6 7 9 10 11 12 13)
          (PROVE (DISABLE STATUS OUTSTANDING FOUND-VALUE NODE-VALUE
                          CHILDREN))
          (CONTRADICT 14) (DROP 3 4 5 6 7 11 12 14) PROVE TOP
          (CONTRADICT 13) (DROP 1 2 3 4 5 6 8 9 10 11 12 13) PROVE
          (CONTRADICT 12) (DROP 3 4 5 6 7 9 10 11 12) PROVE
          (CONTRADICT 11) (DROP 3 4 5 6 7 8 9 10 11) PROVE
          (CONTRADICT 10) (DROP 3 4 5 6 7 8 9 10) PROVE (CONTRADICT 9)
          (DROP 2 3 4 5 6 7 8 9) PROVE S-PROP TOP PROMOTE (DIVE 1)
          (DIVE 2) (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP
          (DIVE 2) (DIVE 2)
          (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP PROVE)))


(PROVE-LEMMA MIN-COMMUTATIVE (REWRITE)
             (EQUAL (MIN A B) (MIN B A)))

(PROVE-LEMMA MIN-ASSOCIATIVE (REWRITE)
             (EQUAL (MIN (MIN A B) C) (MIN A (MIN B C))))

(PROVE-LEMMA MIN-COMMUTATIVE-1 (REWRITE)
             (EQUAL (MIN A (MIN B C)) (MIN B (MIN A C))))

(PROVE-LEMMA MIN-OF-REPORTED-OF-MIN (REWRITE)
             (EQUAL (MIN-OF-REPORTED CHILDREN PARENT STATE
                                     (MIN VALUE X))
                    (MIN (MIN-OF-REPORTED CHILDREN PARENT STATE VALUE)
                         X))
             ((DISABLE REPORTED FOUND-VALUE MIN)))

(PROVE-LEMMA UPDATE-MIN-OF-REPORTED (REWRITE)
  (IMPLIES (AND (NUMBERP PARENT)
                (NUMBERP CHILD)
                (NOT (EQUAL PARENT CHILD))
                (ALL-NUMBERPS CHILDREN)
                (SETP CHILDREN)
                (NOT (MEMBER PARENT CHILDREN))
                (EQUAL (CHANNEL (CONS CHILD PARENT) OLD)
                       (LIST (FOUND-VALUE CHILD OLD)))
                (DONE CHILD OLD)
                (EQUAL (CHANNEL (CONS CHILD PARENT) NEW)
                       (RECEIVE (CONS CHILD PARENT) OLD))
                (CHANGED OLD NEW (LIST (CONS CHILD PARENT)
                                       (CONS 'OUTSTANDING PARENT)
                                       (CONS 'FOUND-VALUE PARENT))))
           (EQUAL (MIN-OF-REPORTED CHILDREN PARENT NEW
                                   VALUE)
                  (IF (MEMBER CHILD CHILDREN)
                      (MIN (FOUND-VALUE CHILD OLD)
                           (MIN-OF-REPORTED CHILDREN PARENT OLD
                                            VALUE))
                    (MIN-OF-REPORTED CHILDREN PARENT OLD VALUE))))
    ((INSTRUCTIONS (INDUCT (LENGTH CHILDREN)) PROMOTE PROMOTE
         (DISABLE REPORTED FOUND-VALUE HEAD CHANNEL RECEIVE MIN DONE)
         (DEMOTE 2) (DIVE 1) (DIVE 1) S-PROP (= T) UP TOP SPLIT
         (DIVE 1) X (DIVE 2) (DIVE 2) = UP UP (DIVE 3) = TOP (DROP 12)
         (CLAIM (EQUAL CHILD (CAR CHILDREN)) 0)
         (CLAIM (AND (NOT (MEMBER CHILD (CDR CHILDREN)))
                     (MEMBER CHILD CHILDREN))
                0)
         S S-PROP SPLIT (DIVE 2) (DIVE 2) X (DIVE 1) (= F) UP S TOP
         (DIVE 1) (DIVE 1) (DIVE 1) = UP (= (FOUND-VALUE CHILD OLD))
         TOP S (CONTRADICT 15) (DROP 1 5 6 7 13 14 15) PROVE
         (CONTRADICT 13) (DROP 2 3 4 5 7 8 9 10 11 13) PROVE
         (CLAIM (MEMBER CHILD (CDR CHILDREN)) 0)
         (CLAIM (MEMBER CHILD CHILDREN) 0) S S-PROP SPLIT (DIVE 2)
```

```
        (DIVE 2) X (DIVE 1) (= T) TOP S (DIVE 1) (DIVE 1)
        (= (FOUND-VALUE (CAR CHILDREN) OLD)) TOP DROP
        (PROVE (DISABLE DONE MIN RECEIVE CHANNEL HEAD FOUND-VALUE
                        REPORTED))
        (DIVE 2) (DIVE 2) X (DIVE 1) (= F) TOP S (CONTRADICT 14)
        (DROP 1 2 3 4 5 6 7 8 9 10 11 12 14) PROVE
        (CLAIM (NOT (MEMBER CHILD CHILDREN)) 0) S (DIVE 2) X (DIVE 1)
        (= (REPORTED (CAR CHILDREN) PARENT NEW)) UP (DIVE 2) (DIVE 1)
        (= (FOUND-VALUE (CAR CHILDREN) NEW)) TOP S (CONTRADICT 14)
        (DROP 1 2 3 4 5 6 7 8 9 10 11 14) PROVE
        (PROVE (DISABLE DONE MIN RECEIVE CHANNEL HEAD FOUND-VALUE
                        REPORTED)))))

(PROVE-LEMMA MIN-OF-REPORTED-OF-NON-ROOT (REWRITE)
   (IMPLIES (AND (NUMBERP ROOT)
                 (NUMBERP CHILD)
                 (NUMBERP PARENT)
                 (ALL-NUMBERPS CHILDREN)
                 (SETP CHILDREN)
                 (NOT (MEMBER PARENT CHILDREN))
                 (NOT (EQUAL ROOT PARENT))
                 (NOT (MEMBER ROOT CHILDREN))
                 (CHANGED OLD NEW (LIST (CONS CHILD ROOT)
                                        (CONS 'OUTSTANDING ROOT)
                                        (CONS 'FOUND-VALUE ROOT)))))
            (EQUAL (MIN-OF-REPORTED CHILDREN PARENT NEW VALUE)
                   (MIN-OF-REPORTED CHILDREN PARENT OLD VALUE)))
    ((INSTRUCTIONS
        (DISABLE MIN REPORTED DONE FOUND-VALUE OUTSTANDING STATUS)
        (INDUCT (LENGTH CHILDREN)) PROMOTE PROMOTE (DEMOTE 2) (DIVE 1)
        (DIVE 1) (= T) UP S TOP PROMOTE (DIVE 1) X (DIVE 2) (DIVE 2) =
        UP UP (DIVE 3) = TOP (DROP 11) (DIVE 1) (DIVE 1)
        (= (REPORTED (CAR CHILDREN) PARENT OLD)) UP (DIVE 2) (DIVE 1)
        (= (FOUND-VALUE (CAR CHILDREN) OLD)) TOP (DIVE 2) X TOP S
        (PROVE (DISABLE STATUS OUTSTANDING FOUND-VALUE DONE REPORTED
                        MIN)))))

(PROVE-LEMMA NUMBER-NOT-REPORTED-OF-NON-ROOT (REWRITE)
  (IMPLIES (AND (NUMBERP ROOT)
                (NUMBERP CHILD)
                (NUMBERP PARENT)
                (ALL-NUMBERPS CHILDREN)
                (SETP CHILDREN)
                (NOT (MEMBER PARENT CHILDREN))
                (NOT (EQUAL ROOT PARENT))
                (NOT (MEMBER ROOT CHILDREN))
                (CHANGED OLD NEW (LIST (CONS CHILD ROOT)
                                       (CONS 'OUTSTANDING ROOT)
                                       (CONS 'FOUND-VALUE ROOT)))))
           (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT NEW)
                  (NUMBER-NOT-REPORTED CHILDREN PARENT OLD)))
    ((INSTRUCTIONS (INDUCT (LENGTH CHILDREN)) PROMOTE PROMOTE
        (DEMOTE 2) (DIVE 1) (DIVE 1) (= T) UP S TOP PROMOTE (DIVE 1)
        (DISABLE REPORTED) X (DIVE 1)
        (= (REPORTED (CAR CHILDREN) PARENT OLD)) UP (DIVE 2) = UP
        (DIVE 3) (DIVE 1) = TOP (DROP 11) (DIVE 2) X TOP S
        (PROVE (DISABLE REPORTED)))))

(PROVE-LEMMA NUMBER-NOT-REPORTED-OF-ROOT (REWRITE)
  (IMPLIES (AND (NUMBERP PARENT)
                (NUMBERP CHILD)
                (NOT (EQUAL PARENT CHILD))
                (ALL-NUMBERPS CHILDREN)
                (SETP CHILDREN)
                (NOT (MEMBER PARENT CHILDREN))
                (EQUAL (CHANNEL (CONS CHILD PARENT) OLD)
                       (LIST (FOUND-VALUE CHILD OLD)))
                (DONE CHILD OLD)
                (EQUAL (CHANNEL (CONS CHILD PARENT) NEW)
```

```
                        (RECEIVE (CONS CHILD PARENT) OLD))
                 (CHANGED OLD NEW (LIST (CONS CHILD PARENT)
                                        (CONS 'OUTSTANDING PARENT)
                                        (CONS 'FOUND-VALUE PARENT))))
            (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT NEW)
                   (IF (MEMBER CHILD CHILDREN)
                       (SUB1 (NUMBER-NOT-REPORTED CHILDREN
                                                  PARENT OLD))
                     (NUMBER-NOT-REPORTED CHILDREN PARENT OLD))))
    ((INSTRUCTIONS (INDUCT (LENGTH CHILDREN)) PROMOTE PROMOTE
         (CLAIM (EQUAL CHILD (CAR CHILDREN)) 0)
         (CLAIM (AND (MEMBER CHILD CHILDREN)
                     (NOT (MEMBER CHILD (CDR CHILDREN))))
                0)
         S (DEMOTE 2) (DIVE 1) (DIVE 1) (= T) UP S TOP PROMOTE (DIVE 1)
         (DISABLE REPORTED) X (DIVE 1) (= T) UP S = TOP (DIVE 2)
         (DIVE 1) X (DIVE 1) (= F) TOP S (CONTRADICT 14)
         (DROP 2 3 4 5 6 8 9 10 11 12 14) PROVE (DEMOTE 2) (DIVE 1)
         (DIVE 1) (= T) UP S TOP (DIVE 2) (DIVE 2) (DIVE 1) X TOP
         S-PROP SPLIT (DIVE 1) X (DIVE 1)
         (= (REPORTED (CAR CHILDREN) PARENT OLD)) UP (DIVE 2) = UP
         (DIVE 3) (DIVE 1) = TOP (DIVE 2) (DIVE 1) X TOP CHANGE-GOAL
         (DIVE 1) X (DIVE 1) (= (REPORTED (CAR CHILDREN) PARENT OLD))
         TOP (DIVE 2) X TOP (DIVE 1) (DIVE 2) = UP (DIVE 3) (DIVE 1) =
         TOP S
         (CLAIM (NOT (ZEROP (NUMBER-NOT-REPORTED (CDR CHILDREN) PARENT
                                 OLD)))
                0)
         (DROP 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
         (PROVE (DISABLE REPORTED)) (CONTRADICT 15)
         (DROP 1 2 3 4 5 6 7 9 10 11 12 14 15)
         (GENERALIZE (((CDR CHILDREN) L))) (INDUCT (MEMBER CHILD L))
         PROVE PROVE PROVE (PROVE (DISABLE REPORTED)))))


(PROVE-LEMMA SETP-NODES-IMPLIES-SETP-ROOTS (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'FOREST FOREST)
                           (SETP (NODES-REC 'FOREST FOREST)))
                      (SETP (ROOTS FOREST)))
             ((INDUCT (ROOTS FOREST))))

(PROVE-LEMMA SETP-NODES-SETP-CHILDREN (REWRITE)
             (IMPLIES (AND (PROPER-TREE FLAG TREE)
                           (SETP (NODES-REC FLAG TREE)))
                      (SETP (CHILDREN-REC FLAG PARENT TREE)))
    ((INSTRUCTIONS (INDUCT (CHILDREN-REC FLAG PARENT TREE)) PROMOTE
         PROMOTE (DIVE 1) X (DIVE 2) (= NIL) TOP PROVE PROMOTE PROMOTE
         PROVE PROMOTE PROMOTE (DIVE 1) X
         (CLAIM (EQUAL (CHILDREN-REC 'TREE PARENT (CAR TREE)) NIL) 0)
         TOP PROVE (DIVE 2)
         (CLAIM (MEMBER PARENT (NODES-REC 'TREE (CAR TREE)))) (= NIL)
         TOP PROVE PROVE)))


(PROVE-LEMMA ROOT-RECEIVE-REPORT-PRESERVES-INSTANCE-OF-NO (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (MEMBER CHILD (CHILDREN (CAR TREE) TREE))
                (ROOT-RECEIVE-REPORT OLD NEW (CAR TREE)
                                     (CONS CHILD (CAR TREE)))
                (UL (UP-LINKS (CDR (NODES TREE)) TREE) OLD)
                (MEMBER NODE (NODES TREE))
                (EQUAL (STATUS NODE NEW) 'STARTED)
                (IMPLIES (EQUAL (STATUS NODE OLD) 'STARTED)
                         (AND (EQUAL (OUTSTANDING NODE OLD)
                                     (NUMBER-NOT-REPORTED
                                      (CHILDREN NODE TREE)
                                      NODE OLD))
                              (EQUAL (FOUND-VALUE NODE OLD)
                                     (MIN-OF-REPORTED
```

```
                                   (CHILDREN NODE TREE)
                                   NODE OLD
                                   (NODE-VALUE NODE OLD))))))
       (AND (EQUAL (OUTSTANDING NODE NEW)
                   (NUMBER-NOT-REPORTED (CHILDREN NODE TREE)
                                        NODE NEW))
            (EQUAL (FOUND-VALUE NODE NEW)
                   (MIN-OF-REPORTED (CHILDREN NODE TREE)
                                    NODE NEW
                                    (NODE-VALUE NODE NEW))))))
((INSTRUCTIONS PROMOTE (CLAIM (EQUAL (STATUS NODE OLD) 'STARTED))
     (DEMOTE 7) (DIVE 1) (DIVE 1) S-PROP UP (S-PROP IMPLIES) TOP
     PROMOTE (CLAIM (EQUAL NODE (CAR TREE)) 0) (DEMOTE 3) (DIVE 1)
     X-DUMB TOP SPLIT (DIVE 2)
     (REWRITE NUMBER-NOT-REPORTED-OF-ROOT
              (($CHILD CHILD) ($OLD OLD)))
     TOP (DIVE 1) (DIVE 1) = UP = (DIVE 1) (DIVE 1) = UP = UP UP
     (DROP 1 3 4 5 6 7 8 10 11 12 13 14) PROVE
     (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
     (DROP 3 4 5 6 7 8 9 10 11 12 13 14) PROVE
     (DROP 3 4 5 6 7 8 10 11 12 13 14) PROVE
     (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
     (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
     (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE (DIVE 1) S
     (REWRITE UL-IMPLIES-INSTANCE-OF-UL-NOT-EMPTY-UPLINK
              (($UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))))
     TOP S (DROP 3 5 6 7 8 10 11 12 13 14) PROVE (DIVE 1) (DIVE 1)
     (DIVE 2) = TOP S-PROP
     (USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL-NOT-EMPTY-UPLINK
         ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))
          (UPLINK (CONS CHILD (CAR TREE))) (STATE OLD)))
     (DEMOTE 15) (DIVE 1) (DIVE 1) S-PROP
     (DROP 3 4 5 6 7 8 9 10 11 12 13 14) (= T) TOP S
     (DROP 1 2 3 4 5 6 7 8 10 12 13 14) PROVE
     (DROP 1 2 3 4 5 6 7 8 10 11 12 13) PROVE
     (DEMOTE 1 2 3 4 5 6 7 8 10 11 12 13 14)
     (BOOKMARK (BEGIN (EQSUB 1))) (= NODE (CAR TREE) 0)
     (CLAIM (EQUAL NODE (CAR TREE)) TAUT) (DROP 1)
     (BOOKMARK (END (EQSUB 1))) DROP PROMOTE (DIVE 2)
     (REWRITE UPDATE-MIN-OF-REPORTED (($CHILD CHILD) ($OLD OLD)))
     TOP S-PROP (DIVE 1) = (DIVE 2) X-DUMB (DIVE 1) S
     (REWRITE UL-IMPLIES-INSTANCE-OF-UL-NOT-EMPTY-UPLINK
              (($UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))))
     TOP (DIVE 2) (DIVE 2) (DIVE 4)
     (= * (NODE-VALUE (CAR TREE) OLD) 0) UP = TOP DROP
     (PROVE (DISABLE MIN)) (DROP 3 4 5 6 7 8 9 10 11 12) PROVE
     (DROP 3 4 5 6 7 8 9 10 11 12 13) PROVE
     (DROP 2 3 4 5 6 7 8 9 10 11 12 13) PROVE
     (DROP 3 4 5 6 7 8 9 10 11 12 13) PROVE
     (DROP 3 4 5 6 7 8 9 10 11 12 13) PROVE
     (DROP 3 4 5 6 7 8 9 10 11 12 13) PROVE
     (DROP 2 3 4 5 6 7 8 9 10 11 12 13) PROVE
     (DROP 2 3 4 5 6 7 8 9 10 11 12 13) PROVE (DIVE 1) S
     (REWRITE UL-IMPLIES-INSTANCE-OF-UL-NOT-EMPTY-UPLINK
              (($UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))))
     TOP S (DROP 4 5 6 7 8 9 10 11 12 13) PROVE
     (USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL-NOT-EMPTY-UPLINK
         ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))
          (UPLINK (CONS CHILD (CAR TREE))) (STATE OLD)))
     (DEMOTE 14) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP S
     (DROP 3 4 5 6 7 8 9 10 11 12 13) PROVE (DIVE 2)
     (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP
     (DROP 1 2 3 4 5 6 8 9 11 12 13 14) PROVE (DIVE 2)
     (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP
     (DROP 1 2 3 4 5 6 7 9 11 12 13 14) PROVE (DIVE 2)
     (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP
     (DROP 1 2 3 4 5 6 8 9 11) PROVE (DIVE 2)
     (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP
     (DROP 1 2 3 4 5 6 7 9 11) PROVE (DEMOTE 3) (DIVE 1) X-DUMB TOP
```

```
         SPLIT (DIVE 2)
         (REWRITE NUMBER-NOT-REPORTED-OF-NON-ROOT
                  (($ROOT (CAR TREE)) ($CHILD CHILD) ($OLD OLD)))
         TOP (DIVE 1) (= * (OUTSTANDING NODE OLD) 0) = TOP S
         (DROP 2 3 5 6 7 8 10 11 12 13) PROVE
         (DROP 2 3 4 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 3 4 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE (DIVE 1)
         (= * (FOUND-VALUE NODE OLD) 0) UP (DIVE 2)
         (REWRITE MIN-OF-REPORTED-OF-NON-ROOT
                  (($OLD OLD) ($CHILD CHILD) ($ROOT (CAR TREE))))
         TOP (DIVE 1) = TOP (DIVE 2) (DIVE 4)
         (= * (NODE-VALUE NODE OLD) 0) TOP S
         (DROP 1 2 3 4 5 6 7 8 10 11 12 13) PROVE
         (DROP 2 3 4 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 3 4 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 2 3 5 6 7 8 9 10 11 12 13 14) PROVE
         (DROP 1 2 3 4 5 6 7 8 10 11 12 13) PROVE (DIVE 2)
         (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP (DIVE 1)
         (= * (OUTSTANDING NODE OLD) 0) = TOP S
         (DROP 1 2 3 4 5 6 7 8 9 11 12 13 14) PROVE (DIVE 2)
         (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) (DIVE 4)
         (= * (NODE-VALUE NODE OLD) 0) TOP (DIVE 1)
         (= * (FOUND-VALUE NODE OLD) 0) = TOP S
         (DROP 1 2 3 4 5 6 7 8 9 11 12 13 14) PROVE
         (DROP 1 2 3 4 5 6 7 8 9 11 12 13 14) PROVE (DIVE 1)
         (= * (OUTSTANDING NODE OLD) 0) = TOP (DIVE 2)
         (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP S
         (DROP 1 2 3 4 5 6 7 8 9 11) PROVE (DIVE 1)
         (= * (FOUND-VALUE NODE OLD) 0) = TOP (DIVE 2)
         (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) (DIVE 4)
         (= * (NODE-VALUE NODE OLD) 0) TOP S
         (DROP 1 2 3 4 5 6 7 8 9 11) PROVE (DROP 1 2 3 4 5 6 7 8 9 11)
         PROVE)))

(PROVE-LEMMA RECEIVE-FIND-PRESERVES-NO-FOR-REST-OF-TREE (REWRITE)
  (IMPLIES (AND (NUMBERP NODE)
                (NUMBERP PARENT-OF-NODE)
                (NUMBERP PARENT)
                (NOT (EQUAL PARENT NODE))
                (NOT (MEMBER NODE CHILDREN))
                (CHANGED OLD NEW
                         (APPEND (LIST (CONS PARENT-OF-NODE NODE)
                                       (CONS NODE PARENT-OF-NODE)
                                       (CONS 'STATUS NODE)
                                       (CONS 'FOUND-VALUE NODE)
                                       (CONS 'OUTSTANDING NODE))
                                 (RFP NODE EXCPT))))
           (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT NEW)
                       (NUMBER-NOT-REPORTED CHILDREN PARENT OLD))
                (EQUAL (MIN-OF-REPORTED CHILDREN PARENT NEW VALUE)
                       (MIN-OF-REPORTED CHILDREN PARENT
                                        OLD VALUE))))
              ((DISABLE MIN)
               (INDUCT (LENGTH CHILDREN))))

(PROVE-LEMMA RECEIVE-FIND-PRESERVES-NO-FOR-NODE (REWRITE)
  (IMPLIES (AND (NUMBERP NODE)
                (NUMBERP PARENT-OF-NODE)
                (NOT (MEMBER NODE CHILDREN))
                (NOT-STARTED CHILDREN OLD)
```

```
                (SUBLISTP (RFP NODE CHILDREN)
                          (RFP NODE EXCPT))
                (CHANGED OLD NEW
                         (APPEND (LIST (CONS PARENT-OF-NODE NODE)
                                       (CONS NODE PARENT-OF-NODE)
                                       (CONS 'STATUS NODE)
                                       (CONS 'FOUND-VALUE NODE)
                                       (CONS 'OUTSTANDING NODE))
                                 (RFP NODE EXCPT))))
           (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN NODE NEW)
                       (LENGTH CHILDREN))
                (EQUAL (MIN-OF-REPORTED CHILDREN NODE NEW VALUE)
                       (MIN-OF-REPORTED CHILDREN NODE
                                        OLD VALUE))))
    ((INSTRUCTIONS (DISABLE MIN) (INDUCT (LENGTH CHILDREN))
         (CLAIM (EQUAL (CDR (ASSOC (CONS 'STATUS (CAR CHILDREN)) NEW))
                       'STARTED)
                0)
         PROMOTE PROMOTE (CONTRADICT 1) (DIVE 1) (DIVE 1) (DIVE 1)
         (REWRITE ABOUT-UC
                  (($B OLD)
                   ($EXCPT (APPEND (LIST (CONS PARENT-OF-NODE NODE)
                                         (CONS NODE PARENT-OF-NODE)
                                         (CONS 'STATUS NODE)
                                         (CONS 'FOUND-VALUE NODE)
                                         (CONS 'OUTSTANDING NODE))
                                   (RFP NODE EXCPT)))))
         TOP PROVE (DROP 1 2 3 4 5 6 7 8) PROVE (DROP 1 3 9) PROVE
         PROVE PROVE)))

(PROVE-LEMMA RECEIVE-FIND-PRESERVES-NO-FOR-PARENT-OF-NODE (REWRITE)
   (IMPLIES (AND (NUMBERP NODE)
                 (NUMBERP PARENT-OF-NODE)
                 (NOT (EQUAL NODE PARENT-OF-NODE))
                 (NOT (EQUAL (STATUS NODE OLD) 'STARTED))
                 (IMPLIES (ZEROP (OUTSTANDING NODE NEW))
                          (NOT (EMPTY (CONS NODE PARENT-OF-NODE)
                                      NEW)))
                 (CHANGED OLD NEW
                          (APPEND (LIST (CONS PARENT-OF-NODE NODE)
                                        (CONS NODE PARENT-OF-NODE)
                                        (CONS 'STATUS NODE)
                                        (CONS 'FOUND-VALUE NODE)
                                        (CONS 'OUTSTANDING NODE))
                                  (RFP NODE EXCPT))))
            (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN
                                             PARENT-OF-NODE NEW)
                        (NUMBER-NOT-REPORTED CHILDREN
                                             PARENT-OF-NODE OLD))
                 (EQUAL (MIN-OF-REPORTED CHILDREN
                                         PARENT-OF-NODE NEW VALUE)
                        (MIN-OF-REPORTED CHILDREN
                                         PARENT-OF-NODE OLD VALUE))))
    ((INSTRUCTIONS (DISABLE MIN) (INDUCT (LENGTH CHILDREN)) PROMOTE
         PROMOTE (DEMOTE 2) (DIVE 1) (DIVE 1) S-PROP
         (= * T ((DISABLE ZEROP OUTSTANDING EMPTY))) UP
         (S-PROP IMPLIES) TOP PROMOTE (DIVE 1) (DIVE 1) X-DUMB S-PROP
         (DIVE 2) = UP (DIVE 3) (DIVE 1) = TOP (DIVE 2) (DIVE 1) X-DUMB
         S-PROP (DIVE 2) (DIVE 2) = UP UP (DIVE 3) = TOP (DROP 8 9)
         (CLAIM (EQUAL (CAR CHILDREN) NODE) 0) (PROVE (DISABLE MIN))
         (CLAIM (EQUAL (REPORTED (CAR CHILDREN) PARENT-OF-NODE NEW)
                       (REPORTED (CAR CHILDREN) PARENT-OF-NODE OLD))
                0)
         (PROVE (DISABLE MIN REPORTED)) (CONTRADICT 9) (DROP 5 6 9)
         (PROVE (DISABLE MIN)) PROVE)))

(PROVE-LEMMA DL-OF-APPEND (REWRITE)
             (EQUAL (DL (APPEND A B) STATE)
                    (AND (DL A STATE)
```

```
                         (DL B STATE))))

(PROVE-LEMMA DOWN-LINKS-1-RFP (REWRITE)
             (EQUAL (DOWN-LINKS-1 PARENT CHILDREN)
                    (RFP PARENT CHILDREN)))

(PROVE-LEMMA DL-DOWN-LINKS-IMPLIES-DL-RFP (REWRITE)
             (IMPLIES (AND (DL (DOWN-LINKS NODES TREE) STATE)
                           (MEMBER NODE NODES))
                      (DL (RFP NODE (CHILDREN NODE TREE)) STATE)))

(DISABLE DL-DOWN-LINKS-IMPLIES-DL-RFP)
(DISABLE DOWN-LINKS-1-RFP)
(DISABLE DL-OF-APPEND)

(PROVE-LEMMA RECEIVE-FIND-PRESERVES-INSTANCE-OF-NO (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER NODE (CDR (NODES TREE)))
                           (RECEIVE-FIND OLD NEW
                                         NODE
                                         (CONS (PARENT NODE TREE) NODE)
                                         (CONS NODE (PARENT NODE TREE))
                                         (RFP NODE (CHILDREN NODE TREE)))
                           (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                           (MEMBER N (NODES TREE))
                           (EQUAL (STATUS N NEW) 'STARTED)
                           (IMPLIES (EQUAL (STATUS N OLD) 'STARTED)
                                    (AND (EQUAL (OUTSTANDING N OLD)
                                                (NUMBER-NOT-REPORTED
                                                 (CHILDREN N TREE)
                                                 N OLD))
                                         (EQUAL (FOUND-VALUE N OLD)
                                                (MIN-OF-REPORTED
                                                 (CHILDREN N TREE)
                                                 N OLD
                                                 (NODE-VALUE N OLD))))))
                      (AND (EQUAL (OUTSTANDING N NEW)
                                  (NUMBER-NOT-REPORTED (CHILDREN N TREE)
                                                       N NEW))
                           (EQUAL (FOUND-VALUE N NEW)
                                  (MIN-OF-REPORTED (CHILDREN N TREE)
                                                   N NEW
                                                   (NODE-VALUE N NEW)))))
    ((INSTRUCTIONS PROMOTE
         (CLAIM (EQUAL (HEAD (CONS (PARENT NODE TREE) NODE) OLD) 'FIND)
                0)
         (CLAIM (EQUAL N NODE) 0)
         (CLAIM (NOT-STARTED (CHILDREN NODE TREE) OLD) 0) (DEMOTE 3)
         (DIVE 1) X-DUMB S-PROP TOP (DROP 6) PROMOTE (DIVE 1) (DIVE 2)
         (REWRITE RECEIVE-FIND-PRESERVES-NO-FOR-NODE
                  (($OLD OLD) ($PARENT-OF-NODE (PARENT NODE TREE))
                   ($EXCPT (CHILDREN NODE TREE))))
         TOP (DIVE 2) (DIVE 2)
         (REWRITE RECEIVE-FIND-PRESERVES-NO-FOR-NODE
                  (($OLD OLD) ($PARENT-OF-NODE (PARENT NODE TREE))
                   ($EXCPT (CHILDREN NODE TREE))))
         TOP (BASH T) PROMOTE
         (GENERALIZE (((CHILDREN-REC 'TREE NODE TREE) CHILDREN)))
         (DROP 1 2 3 4 5 6 7 8 9 11 12 13 14 16 17)
         (INDUCT (LENGTH CHILDREN)) PROVE PROVE (DROP 2 3 5 6 7 8 9)
         PROVE (DROP 3 4 5 6 7 8 9) PROVE (DROP 2 3 5 6 7 8 9) PROVE
         (DIVE 1) (DIVE 1) = TOP S-PROP (DIVE 1) (DIVE 2) (DIVE 1) =
         TOP (REWRITE SUBLISTP-REFLEXIVE) PROVE (DROP 2 3 5 6 7 8 9)
         PROVE (DROP 3 4 5 6 7 8 9) PROVE (DROP 2 3 5 6 7 8 9) PROVE
         (DIVE 1) (DIVE 1) = TOP S-PROP (DIVE 1) (DIVE 2) (DIVE 1) =
         TOP (REWRITE SUBLISTP-REFLEXIVE) PROVE (CONTRADICT 10)
         (REWRITE PARENT-NOT-STARTED-IMPLIES-ALL-EMPTY-AND-NOT-STARTED
                  (($PARENT NODE)))
         (USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
```

```
    ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))
     (DOWN-LINK (CONS (PARENT NODE TREE) NODE)) (STATE OLD)))
(DEMOTE 11) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP
(DROP 1 2 3 4 5 6 7 9 10) PROVE (DROP 3 4 5 6 7 8 9 10) S
(REWRITE MEMBER-DOWN-LINKS) S (DIVE 2)
(REWRITE PARENT-REC-CHILDREN-REC) TOP (DIVE 1)
(REWRITE NODE-HAS-PARENT) TOP S (DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT) TOP PROVE PROVE PROVE PROVE
PROVE PROVE
(REWRITE DL-DOWN-LINKS-IMPLIES-DL-RFP (($NODES (NODES TREE))))
PROVE (CLAIM (EQUAL N (PARENT NODE TREE)) 0) (DEMOTE 3)
(DIVE 1) X-DUMB S-PROP TOP PROMOTE (DIVE 1) (DIVE 2)
(REWRITE RECEIVE-FIND-PRESERVES-NO-FOR-PARENT-OF-NODE
         (($OLD OLD) ($NODE NODE)
          ($EXCPT (CHILDREN NODE TREE))))
UP TOP (DIVE 2) (DIVE 2)
(REWRITE RECEIVE-FIND-PRESERVES-NO-FOR-PARENT-OF-NODE
         (($OLD OLD) ($NODE NODE)
          ($EXCPT (CHILDREN NODE TREE))))
TOP (DIVE 1) (DIVE 1) (= * (OUTSTANDING N OLD) 0) TOP (DIVE 2)
(DIVE 1) (= * (FOUND-VALUE N OLD) 0) TOP (DIVE 2) (DIVE 2)
(DIVE 4) (= * (NODE-VALUE N OLD) 0) TOP (DEMOTE 6) (DIVE 1)
(DIVE 1) (= * T 0) TOP S (DEMOTE 5) (DIVE 1) (DIVE 1) S
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CONS (PARENT NODE TREE) NODE)
                                (CONS NODE (PARENT NODE TREE))
                                (CONS 'STATUS NODE)
                                (CONS 'FOUND-VALUE NODE)
                                (CONS 'OUTSTANDING NODE))
                               (RFP NODE (CHILDREN NODE TREE))))))
TOP S (DROP 1 2 3 4 5 6 7) PROVE (CLAIM (NUMBERP NODE) 0)
(DROP 1 2 3 4 5 7 8) PROVE (CONTRADICT 9) (DROP 3 4 5 6 7 8 9)
PROVE (DIVE 1) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CONS (PARENT NODE TREE) NODE)
                                (CONS NODE (PARENT NODE TREE))
                                (CONS 'STATUS NODE)
                                (CONS 'FOUND-VALUE NODE)
                                (CONS 'OUTSTANDING NODE))
                               (RFP NODE (CHILDREN NODE TREE))))))
TOP S (DROP 1 2 3 4 5 6 7 8 9) PROVE (CLAIM (NUMBERP NODE) 0)
(DROP 1 2 3 4 5 6 7 9 10) PROVE (CONTRADICT 11)
(DROP 3 4 5 6 7 8 9 10 11) PROVE (DIVE 1) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CONS (PARENT NODE TREE) NODE)
                                (CONS NODE (PARENT NODE TREE))
                                (CONS 'STATUS NODE)
                                (CONS 'FOUND-VALUE NODE)
                                (CONS 'OUTSTANDING NODE))
                               (RFP NODE (CHILDREN NODE TREE))))))
TOP S (DROP 1 2 3 4 5 6 7 8 9) PROVE (CLAIM (NUMBERP NODE) 0)
(DROP 1 2 3 4 5 6 7 9 10) PROVE (CONTRADICT 11)
(DROP 3 4 5 6 7 8 9 10 11) PROVE (DIVE 1) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CONS (PARENT NODE TREE) NODE)
                                (CONS NODE (PARENT NODE TREE))
                                (CONS 'STATUS NODE)
                                (CONS 'FOUND-VALUE NODE)
                                (CONS 'OUTSTANDING NODE))
                               (RFP NODE (CHILDREN NODE TREE))))))
TOP S (DROP 1 2 3 4 5 6 7 8 9) PROVE (CLAIM (NUMBERP NODE) 0)
(DROP 1 2 3 4 5 6 7 9 10) PROVE (CONTRADICT 11)
(DROP 3 4 5 6 7 8 9 10 11) PROVE (DROP 3 4 5 6 7 8 9 10) PROVE
(DROP 2 3 5 6 7 8 9 10) PROVE
```

```
(USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
    ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))
     (DOWN-LINK (CONS (PARENT NODE TREE) NODE)) (STATE OLD)))
(DEMOTE 11) (DIVE 1) (DIVE 1) S-PROP
(REWRITE MEMBER-DOWN-LINKS) (DIVE 1) S
(REWRITE NODE-HAS-PARENT) UP S
(REWRITE PARENT-REC-CHILDREN-REC) (DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT) S UP (= * T 0) TOP
(DROP 1 2 3 4 5 6 8 9 10) PROVE (DROP 3 4 5 6 7 8 9 10) PROVE
(DROP 2 3 4 5 6 7 8 9 10) PROVE (DROP 2 3 4 5 6 7 8 9 10)
PROVE (DROP 3 4 5 6 7 8 9 10) PROVE (DROP 2 3 4 5 6 7 8 9 10)
PROVE S (DROP 3 4 5 6 7 8 9 10) PROVE (DROP 1 2 3 4 5 6 7 8)
(PROVE (DISABLE OUTSTANDING LENGTH-RFP FOUND-VALUE STATUS
                NODE-VALUE CHANGED))
(DROP 1 2 3 4 5 6 7 8)
(PROVE (DISABLE CHANGED OUTSTANDING FOUND-VALUE STATUS RECEIVE
                SEND LENGTH-RFP))
(DROP 3 4 5 6 7 8 9 10) PROVE (DROP 2 3 5 6 7 8 9 10) PROVE
(USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
    ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))
     (DOWN-LINK (CONS (PARENT NODE TREE) NODE)) (STATE OLD)))
(DEMOTE 11) (DIVE 1) (DIVE 1) S-PROP
(REWRITE MEMBER-DOWN-LINKS) (DIVE 1) S
(REWRITE NODE-HAS-PARENT) UP S
(REWRITE PARENT-REC-CHILDREN-REC) (DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT) UP S (= * T 0) TOP
(DROP 1 2 3 4 5 6 8 10) PROVE (DROP 3 4 5 6 7 8 9 10) PROVE
(DROP 2 3 4 5 6 7 8 9 10) PROVE (DROP 2 3 4 5 6 7 8 9 10)
PROVE (DROP 3 4 5 6 7 8 9 10) PROVE (DROP 2 3 4 5 6 7 8 9 10)
PROVE S (DROP 3 4 5 6 7 8 9 10) PROVE (DROP 1 2 3 4 5 6 7 8)
(PROVE (DISABLE SEND LENGTH-RFP OUTSTANDING NODE-VALUE
                FOUND-VALUE STATUS RECEIVE CHANGED))
(DROP 1 2 3 4 5 6 7 8)
(PROVE (DISABLE LENGTH-RFP CHANGED FOUND-VALUE NODE-VALUE SEND
                RECEIVE OUTSTANDING))
(DEMOTE 3) (DIVE 1) X-DUMB S-PROP TOP PROMOTE (DIVE 1)
(DIVE 2)
(REWRITE RECEIVE-FIND-PRESERVES-NO-FOR-REST-OF-TREE
         (($OLD OLD) ($NODE NODE)
          ($PARENT-OF-NODE (PARENT NODE TREE))
          ($EXCPT (CHILDREN NODE TREE))))
UP (DIVE 1) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CONS (PARENT NODE TREE) NODE)
                                (CONS NODE (PARENT NODE TREE))
                                (CONS 'STATUS NODE)
                                (CONS 'FOUND-VALUE NODE)
                                (CONS 'OUTSTANDING NODE))
                               (RFP NODE (CHILDREN NODE TREE))))))
TOP (DIVE 2) (DIVE 1) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CONS (PARENT NODE TREE) NODE)
                                (CONS NODE (PARENT NODE TREE))
                                (CONS 'STATUS NODE)
                                (CONS 'FOUND-VALUE NODE)
                                (CONS 'OUTSTANDING NODE))
                               (RFP NODE (CHILDREN NODE TREE))))))
UP UP (DIVE 2)
(REWRITE RECEIVE-FIND-PRESERVES-NO-FOR-REST-OF-TREE
         (($OLD OLD) ($NODE NODE)
          ($PARENT-OF-NODE (PARENT NODE TREE))
          ($EXCPT (CHILDREN NODE TREE))))
(DIVE 4) S (DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (APPEND (LIST (CONS (PARENT NODE TREE) NODE)
                                (CONS NODE (PARENT NODE TREE))
```

```
                              (CONS 'STATUS NODE)
                              (CONS 'FOUND-VALUE NODE)
                              (CONS 'OUTSTANDING NODE))
                             (RFP NODE (CHILDREN NODE TREE))))))
         TOP (DEMOTE 6) (DEMOTE 5) (DIVE 1) (DIVE 1) S (DIVE 1)
         (REWRITE ABOUT-UC
                  (($B OLD)
                   ($EXCPT (APPEND (LIST (CONS (PARENT NODE TREE) NODE)
                              (CONS NODE (PARENT NODE TREE))
                              (CONS 'STATUS NODE)
                              (CONS 'FOUND-VALUE NODE)
                              (CONS 'OUTSTANDING NODE))
                             (RFP NODE (CHILDREN NODE TREE))))))
         TOP S (DROP 1 2 3 4 5 6 7)
         (PROVE (DISABLE FOUND-VALUE STATUS OUTSTANDING NODE-VALUE SEND
                         RECEIVE CHANNEL EMPTY HEAD LENGTH-RFP))
         (CLAIM (NUMBERP NODE) 0) (DROP 1 2 3 4 5 7 8) PROVE
         (CONTRADICT 9) (DROP 3 4 5 6 7 8 9) PROVE
         (DROP 1 2 3 4 5 6 7 8 9)
         (PROVE (DISABLE LENGTH-RFP STATUS OUTSTANDING NODE-VALUE
                         FOUND-VALUE SEND RECEIVE CHANNEL EMPTY HEAD
                         CHILDREN PARENT))
         (CLAIM (NUMBERP NODE) 0) (DROP 1 2 3 4 5 6 7 9 10) PROVE
         (CONTRADICT 11) (DROP 3 4 5 6 7 8 9 10 11) PROVE
         (DROP 3 4 5 6 7 8 9 10) PROVE (DROP 3 4 5 6 7 8 9 10) PROVE
         (DROP 2 3 5 6 7 8 9 10) PROVE (DROP 3 5 6 7 10) PROVE
         (DROP 1 2 3 4 5 6 7 8 9)
         (PROVE (DISABLE LENGTH-RFP CHANGED STATUS FOUND-VALUE
                         OUTSTANDING NODE-VALUE SEND RECEIVE CHANNEL
                         EMPTY CHILDREN PARENT))
         (DROP 1 2 3 4 5 6 7 8 9)
         (PROVE (DISABLE LENGTH-RFP STATUS OUTSTANDING NODE-VALUE
                         FOUND-VALUE CHANNEL EMPTY SEND RECEIVE
                         CHILDREN PARENT))
         (CLAIM (NUMBERP NODE) 0) (DROP 1 2 3 4 5 6 7 9 10) PROVE
         (CONTRADICT 11) (DROP 3 4 5 6 7 8 9 10 11) PROVE
         (DROP 1 2 3 4 5 6 7 8 9)
         (PROVE (DISABLE LENGTH-RFP CHILDREN PARENT STATUS OUTSTANDING
                         NODE-VALUE FOUND-VALUE SEND RECEIVE CHANNEL
                         EMPTY))
         (CLAIM (NUMBERP NODE) 0) (DROP 1 2 3 4 5 6 7 9 10) PROVE
         (CONTRADICT 11) (DROP 3 4 5 6 7 8 9 10 11) PROVE
         (DROP 3 4 5 6 7 8 9 10) PROVE (DROP 3 4 5 6 7 8 9 10) PROVE
         (DROP 2 3 5 6 7 8 9 10) PROVE (DROP 3 5 6 7 10) PROVE
         (DROP 1 2 3 4 5 6 7 8 9)
         (PROVE (DISABLE PARENT CHILDREN OUTSTANDING NODE-VALUE
                         FOUND-VALUE STATUS LENGTH-RFP RECEIVE SEND
                         CHANNEL EMPTY HEAD))
         (DEMOTE 3) (DIVE 1) X-DUMB S-PROP TOP PROMOTE (DROP 7)
         (DIVE 1) (DIVE 2)
         (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP (DIVE 2)
         (DIVE 2) (REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP
         (DROP 1 2 3 4) (PROVE (DISABLE PARENT CHILDREN)))))

(PROVE-LEMMA RECEIVE-REPORT-PRESERVES-NO-FOR-REST-OF-TREE (REWRITE)
   (IMPLIES (AND (NUMBERP NODE)
                 (NUMBERP PARENT-OF-NODE)
                 (NUMBERP CHILD-OF-NODE)
                 (NUMBERP PARENT)
                 (NOT (EQUAL PARENT NODE))
                 (NOT (MEMBER NODE CHILDREN))
                 (CHANGED OLD NEW
                          (LIST (CONS CHILD-OF-NODE NODE)
                                (CONS NODE PARENT-OF-NODE)
                                (CONS 'OUTSTANDING NODE)
                                (CONS 'FOUND-VALUE NODE))))
            (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT NEW)
                        (NUMBER-NOT-REPORTED CHILDREN PARENT OLD))
                 (EQUAL (MIN-OF-REPORTED CHILDREN PARENT NEW VALUE)
```

```
                        (MIN-OF-REPORTED CHILDREN PARENT
                                         OLD VALUE))))
  ((DISABLE MIN)
   (INDUCT (LENGTH CHILDREN))))

(PROVE-LEMMA RECEIVE-REPORT-PRESERVES-NO-FOR-NODE (REWRITE)
   (IMPLIES (AND (NUMBERP NODE)
                 (NUMBERP PARENT)
                 (NUMBERP CHILD)
                 (NOT (MEMBER NODE CHILDREN))
                 (ALL-NUMBERPS CHILDREN)
                 (SETP CHILDREN)
                 (EQUAL (CHANNEL (CONS CHILD NODE) OLD)
                        (LIST (FOUND-VALUE CHILD OLD)))
                 (DONE CHILD OLD)
                 (EQUAL (CHANNEL (CONS CHILD NODE) NEW)
                        (RECEIVE (CONS CHILD NODE) OLD))
                 (CHANGED OLD NEW
                          (LIST (CONS CHILD NODE)
                                (CONS NODE PARENT)
                                (CONS 'OUTSTANDING NODE)
                                (CONS 'FOUND-VALUE NODE))))
            (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN NODE NEW)
                        (IF (MEMBER CHILD CHILDREN)
                            (SUB1 (NUMBER-NOT-REPORTED
                                   CHILDREN NODE OLD))
                          (NUMBER-NOT-REPORTED CHILDREN NODE OLD)))
                 (EQUAL (MIN-OF-REPORTED CHILDREN NODE NEW VALUE)
                        (IF (MEMBER CHILD CHILDREN)
                            (MIN (FOUND-VALUE CHILD OLD)
                                 (MIN-OF-REPORTED CHILDREN
                                                  NODE OLD VALUE))
                          (MIN-OF-REPORTED CHILDREN NODE
                                           OLD VALUE)))))
    ((INSTRUCTIONS (INDUCT (LENGTH CHILDREN)) PROMOTE PROMOTE
         (DISABLE REPORTED FOUND-VALUE HEAD CHANNEL RECEIVE MIN DONE)
         (DEMOTE 2) (DIVE 1) (DIVE 1) S-PROP (= T) TOP
         (CLAIM (EQUAL CHILD (CAR CHILDREN)) 0)
         (CLAIM (AND (MEMBER CHILD CHILDREN)
                     (NOT (MEMBER CHILD (CDR CHILDREN))))
                0)
         S SPLIT (DIVE 1) X (DIVE 1) (= * T 0) UP S (DIVE 1)
         (= * (FOUND-VALUE CHILD OLD) 0) UP (DIVE 2) = TOP (DIVE 2)
         (DIVE 2) X (DIVE 1) (= * F ((DISABLE DONE CHANNEL))) TOP S
         (DROP 8 9 10 15 16 17) PROVE (DROP 15 16 17) PROVE (DIVE 1) X
         (DIVE 1) (= * T 0) UP S = TOP (DIVE 2) (DIVE 1) X (DIVE 1)
         (= * F ((DISABLE DONE CHANNEL))) TOP S (DROP 15 16) PROVE
         (CONTRADICT 13) (DROP 2 3 4 5 6 8 9 10 11 13) PROVE
         (CLAIM (AND (MEMBER CHILD (CDR CHILDREN))
                     (MEMBER CHILD CHILDREN))
                0)
         S SPLIT (DIVE 1) X (DIVE 1)
         (= * (REPORTED (CAR CHILDREN) NODE OLD) 0) UP (DIVE 2)
         (DIVE 1) (= * (FOUND-VALUE (CAR CHILDREN) OLD) 0) UP (DIVE 2)
         = UP UP (DIVE 3) = TOP (DIVE 2) (DIVE 2) X TOP DROP
         (PROVE (DISABLE DONE MIN RECEIVE CHANNEL HEAD FOUND-VALUE
                         REPORTED))
         (DROP 8 9 10 15 16 17) PROVE (DROP 8 9 10 15 16 17) PROVE
         (DIVE 1) X (DIVE 1) (= * (REPORTED (CAR CHILDREN) NODE OLD) 0)
         UP (DIVE 2) = UP (DIVE 3) (DIVE 1) = TOP (DIVE 2) (DIVE 1) X
         TOP S-PROP S S-PROP SPLIT (CONTRADICT 18)
         (DROP 1 2 3 4 5 6 7 10 11 12 14 15 16 17 18)
         (GENERALIZE (((CDR CHILDREN) L)))
         (PROVE (DISABLE CHANNEL DONE)) (DROP 8 9 10 13 14 15 16) PROVE
         (DEMOTE 13) (DIVE 1) (DIVE 1) (DIVE 2) X UP S TOP PROMOTE
         (CLAIM (NOT (MEMBER CHILD CHILDREN)) 0) S SPLIT (DIVE 1) X
         (DIVE 1) (= * (REPORTED (CAR CHILDREN) NODE OLD) 0) UP
         (DIVE 2) (DIVE 2) = UP (DIVE 1)
         (= * (FOUND-VALUE (CAR CHILDREN) OLD) 0) UP UP (DIVE 3) = TOP
```

```
          (DIVE 2) X TOP S (DROP 8 9 10 15 16 17 18) PROVE
          (DROP 8 9 10 15 16 17) PROVE (DIVE 1) X (DIVE 1)
          (= * (REPORTED (CAR CHILDREN) NODE OLD) 0) UP (DIVE 2) = UP
          (DIVE 3) (DIVE 1) = TOP (DIVE 2) X TOP S (DROP 8 9 10 15 16)
          PROVE (CONTRADICT 14) (DROP 2 3 4 5 6 8 9 10 11 14) PROVE
          PROVE)))

(PROVE-LEMMA RECEIVE-REPORT-PRESERVES-NO-FOR-PARENT (REWRITE)
  (IMPLIES (AND (NUMBERP NODE)
                (NUMBERP PARENT)
                (NOT (EQUAL NODE PARENT))
                (IMPLIES (ZEROP (OUTSTANDING NODE NEW))
                         (NOT (EMPTY (CONS NODE PARENT) NEW)))
                (NOT (ZEROP (OUTSTANDING NODE OLD)))
                (CHANGED OLD NEW
                         (LIST (CONS CHILD NODE)
                               (CONS NODE PARENT)
                               (CONS 'OUTSTANDING NODE)
                               (CONS 'FOUND-VALUE NODE))))
           (AND (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT NEW)
                       (NUMBER-NOT-REPORTED CHILDREN PARENT OLD))
                (EQUAL (MIN-OF-REPORTED CHILDREN PARENT NEW VALUE)
                       (MIN-OF-REPORTED CHILDREN
                                        PARENT OLD VALUE))))
    ((INSTRUCTIONS (DISABLE MIN REPORTED) (INDUCT (LENGTH CHILDREN))
          PROMOTE PROMOTE (DEMOTE 2) (DIVE 1) (DIVE 1) (S-PROP AND) S UP
          (S-PROP IMPLIES) TOP PROMOTE
          (CLAIM (EQUAL (CAR CHILDREN) NODE) 0) PROVE PROVE PROVE)))


(PROVE-LEMMA CHILD-MEMBER-CDR-NODES (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                           (SETP (NODES-REC 'TREE TREE))
                           (MEMBER CHILD (CHILDREN-REC 'TREE NODE TREE)))
                      (MEMBER CHILD (CDR (NODES-REC 'TREE TREE))))
    ((INSTRUCTIONS PROMOTE (CONTRADICT 3) (DIVE 1)
          (REWRITE PARENT-REC-CHILDREN-REC) (DIVE 2)
          (REWRITE SETP-TREE-UNIQUE-PARENT) TOP (BASH T))))

(PROVE-LEMMA RECEIVE-REPORT-PRESERVES-INSTANCE-OF-NO (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER NODE (CDR (NODES TREE)))
                           (MEMBER CHILD (CHILDREN NODE TREE))
                           (MEMBER N (NODES TREE))
                           (RECEIVE-REPORT OLD NEW
                                           NODE
                                           (CONS CHILD NODE)
                                           (CONS NODE (PARENT NODE TREE)))
                           (EQUAL (STATUS N NEW) 'STARTED)
                           (UL (UP-LINKS (CDR (NODES TREE)) TREE) OLD)
                           (NO (NODES TREE) TREE OLD)
                           (DL (DOWN-LINKS (NODES TREE) TREE) OLD))
                      (AND (EQUAL (OUTSTANDING N NEW)
                                  (NUMBER-NOT-REPORTED (CHILDREN N TREE)
                                                       N NEW))
                           (EQUAL (FOUND-VALUE N NEW)
                                  (MIN-OF-REPORTED (CHILDREN N TREE)
                                                   N NEW
                                                   (NODE-VALUE N NEW)))))
    ((INSTRUCTIONS PROMOTE
          (CLAIM (AND (NUMBERP NODE) (NUMBERP (PARENT NODE TREE))
                      (NUMBERP N) (NUMBERP CHILD)
                      (NOT (EQUAL CHILD NODE))
                      (NOT (EQUAL (PARENT NODE TREE) NODE))
                      (NOT (MEMBER NODE (CHILDREN NODE TREE)))
                      (NOT (MEMBER N (CHILDREN N TREE)))
                      (ALL-NUMBERPS (CHILDREN NODE TREE))
                      (ALL-NUMBERPS (CHILDREN N TREE))
                      (SETP (CHILDREN NODE TREE))
```

```
             (SETP (CHILDREN N TREE)))
       0)
(CLAIM (AND (EQUAL (STATUS N OLD) (STATUS N NEW))
             (EQUAL (NODE-VALUE N OLD) (NODE-VALUE N NEW)))
       0)
(CLAIM (EQUAL (CHANNEL (CONS CHILD NODE) OLD)
               (LIST (FOUND-VALUE CHILD OLD)))
       0)
(CLAIM (DONE CHILD OLD) 0)
(CLAIM (EQUAL (STATUS NODE OLD) 'STARTED) 0)
(CLAIM (EQUAL N NODE) 0)
(DEMOTE 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
        22 23 24 25 26)
(= N NODE) DROP PROMOTE
(USE-LEMMA RECEIVE-REPORT-PRESERVES-NO-FOR-NODE
    ((NODE NODE) (PARENT (PARENT NODE TREE)) (CHILD CHILD)
     (CHILDREN (CHILDREN NODE TREE)) (OLD OLD) (NEW NEW)
     (VALUE (NODE-VALUE NODE OLD))))
(DEMOTE 23) (DIVE 1) (DIVE 1) S-PROP (= * T 0) UP
(S-PROP IMPLIES) TOP PROMOTE
(USE-LEMMA NO-IMPLIES-INSTANCE-OF-NO
    ((NODES (NODES TREE)) (TREE TREE) (STATE OLD) (NODE NODE)))
(DEMOTE 25) (DIVE 1) (DIVE 1) S-PROP UP (S-PROP IMPLIES) TOP
PROMOTE (DIVE 1) (DIVE 2) = TOP (DIVE 2) (DIVE 2) (DIVE 4) =
UP = TOP
(DROP 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23
      24)
(PROVE (DISABLE CHANGED MIN ZEROP CHANNEL))
(DROP 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22)
(PROVE (DISABLE CHANNEL RECEIVE MIN ZEROP CHANGED))
(CLAIM (EQUAL N (PARENT NODE TREE)) 0)
(DEMOTE 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
        22 23 24 25 26 27)
(= N (PARENT NODE TREE)) DROP PROMOTE
(USE-LEMMA RECEIVE-REPORT-PRESERVES-NO-FOR-PARENT
    ((NODE NODE) (PARENT (PARENT NODE TREE)) (OLD OLD)
     (NEW NEW) (CHILD CHILD)
     (CHILDREN (CHILDREN (PARENT NODE TREE) TREE))
     (VALUE (NODE-VALUE (PARENT NODE TREE) OLD))))
(DEMOTE 26) (DIVE 1) (DIVE 1) S-PROP (= * T 0) UP
(S-PROP IMPLIES) TOP PROMOTE
(CLAIM (AND (EQUAL (OUTSTANDING (PARENT NODE TREE) NEW)
                    (OUTSTANDING (PARENT NODE TREE) OLD))
             (EQUAL (FOUND-VALUE (PARENT NODE TREE) NEW)
                    (FOUND-VALUE (PARENT NODE TREE) OLD)))
       0)
(USE-LEMMA NO-IMPLIES-INSTANCE-OF-NO
    ((NODES (NODES TREE)) (TREE TREE) (STATE OLD)
     (NODE (PARENT NODE TREE))))
(DEMOTE 30) (DIVE 1) (DIVE 1) S-PROP (DIVE 1) = = UP S UP
(S-PROP IMPLIES) TOP PROMOTE (DIVE 1) (DIVE 1) = UP (DIVE 2) =
TOP (DIVE 2) (DIVE 1) = UP (DIVE 2) (DIVE 4) = UP = TOP
(DROP 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
      23 24 25 2 6 27 28 29)
PROVE (CONTRADICT 28)
(DROP 1 2 3 4 6 7 8 9 12 13 14 15 16 17 18 19 20 21 23 24 26
      27 28)
(PROVE (DISABLE CHANNEL MIN ZEROP)) (DIVE 1) (DIVE 1)
(= * F 0) UP S-PROP (DIVE 2) (DIVE 3) (DIVE 3) (= * T 0) TOP
(DIVE 1) (DIVE 3) (DIVE 3) (= * T 0) TOP
(CLAIM (NOT (ZEROP (OUTSTANDING NODE OLD))) 0) S-PROP
(DROP 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 23
      24 25 26)
(PROVE (DISABLE MIN CHANGED CHANNEL)) (CONTRADICT 26) (DIVE 1)
(DIVE 1) S
(REWRITE NO-IMPLIES-INSTANCE-OF-NO
         (($NODES (NODES TREE)) ($TREE TREE)))
TOP S (CLAIM (NOT (REPORTED CHILD NODE OLD)) 0)
(CONTRADICT 27)
```

```
(REWRITE NUMBER-NOT-REPORTED-0-IMPLIES
         (($CHILDREN (CHILDREN NODE TREE))))
(DEMOTE 27) DROP PROVE (CONTRADICT 27) (DIVE 1) X-DUMB S-PROP
X-DUMB (DIVE 1) (DIVE 1) = TOP S
(DROP 3 4 5 6 7 8 9 10 11 12 13 14 15 1 17 18 19 20 21 22 23
      24 25 26)
PROVE
(DROP 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 23
      24 25 26 27)
(PROVE (DISABLE CHANGED CHANNEL ZEROP MIN))
(DROP 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 23
      24 25 26 27 28)
(PROVE (DISABLE CHANNEL CHANGED MIN ZEROP))
(DROP 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
      23 24)
PROVE
(CLAIM (AND (EQUAL (FOUND-VALUE N OLD) (FOUND-VALUE N NEW))
            (EQUAL (OUTSTANDING N OLD) (OUTSTANDING N NEW)))
       0)
(USE-LEMMA RECEIVE-REPORT-PRESERVES-NO-FOR-REST-OF-TREE
    ((NODE NODE) (CHILD-OF-NODE CHILD)
     (PARENT-OF-NODE (PARENT NODE TREE)) (PARENT N)
     (CHILDREN (CHILDREN N TREE)) (VALUE (NODE-VALUE N OLD))))
(DEMOTE 31) (DIVE 1) (DIVE 1) S-PROP (= * T 0) UP
(S-PROP IMPLIES) TOP PROMOTE
(USE-LEMMA NO-IMPLIES-INSTANCE-OF-NO
    ((NODES (NODES TREE)) (TREE TREE) (STATE OLD) (NODE N)))
(DEMOTE 33) (DIVE 1) (DIVE 1) S-PROP (DIVE 1) = = UP S UP
(S-PROP IMPLIES) TOP PROMOTE (DIVE 1) (DIVE 1) = UP (DIVE 2) =
TOP (DIVE 2) (DIVE 1) = UP (DIVE 2) (DIVE 4) = UP = TOP
(DROP 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
      23 24 25 26 27 28 29 30 31 32)
PROVE S-PROP SPLIT
(DROP 3 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
      25 26 29 30)
PROVE
(DROP 1 2 3 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
      23 25 26 27 28 29 30 31)
(PROVE (DISABLE CHANGED CHANNEL MIN ZEROP)) (CONTRADICT 29)
(DROP 1 2 3 4 6 7 8 9 16 17 18 19 20 21 22 23 25 26 29)
(PROVE (DISABLE MIN ZEROP CHANNEL))
(CLAIM (EQUAL (STATUS CHILD OLD) ’STARTED) 0)
(USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
    ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))
     (DOWN-LINK (CONS NODE CHILD)) (STATE OLD)))
(CONTRADICT 26) (DEMOTE 28) (DIVE 1) (DIVE 1) S-PROP (= * T 0)
TOP (DEMOTE 27) DROP PROVE
(DROP 5 6 7 8 9 18 19 20 21 22 23 24 25 26 27) PROVE
(CONTRADICT 27)
(USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL
    ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE)) (STATE OLD)
     (UPLINK (CONS CHILD NODE))))
(DEMOTE 28) (DIVE 1) (DIVE 1) S-PROP (= * T 0) UP
(S-PROP IMPLIES) TOP (DEMOTE 24) DROP PROVE
(DROP 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
      25 26 27)
PROVE (CONTRADICT 25)
(USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL
    ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE)) (STATE OLD)
     (UPLINK (CONS CHILD NODE))))
(DEMOTE 26) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP (DEMOTE 24)
DROP PROVE
(DROP 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
      25)
PROVE (DEMOTE 5) (DIVE 1) X-DUMB (DIVE 1) (= * T 0) UP S-PROP
TOP PROMOTE (DIVE 1) (DIVE 2)
(REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP (DIVE 2)
(DIVE 2) (DIVE 4) = UP
(REWRITE UNCHANGED-PRESERVES-NO (($OLD OLD))) TOP
```

```
          (USE-LEMMA NO-IMPLIES-INSTANCE-OF-NO
              ((NODES (NODES TREE)) (TREE TREE) (STATE OLD) (NODE N)))
          (DEMOTE 25) (DIVE 1) (DIVE 1) S-PROP (DIVE 1) = = UP S UP
          (S-PROP IMPLIES) TOP (DEMOTE 24) DROP PROVE
          (USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL
              ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE)) (STATE OLD)
               (UPLINK (CONS CHILD NODE))))
          (DEMOTE 24) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP (DEMOTE 23)
          DROP PROVE
          (DROP 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
          PROVE (CONTRADICT 22)
          (DROP 1 2 3 4 6 7 8 9 16 17 18 19 20 21 22)
          (PROVE (DISABLE MIN ZEROP CHANNEL SEND RECEIVE HEAD PARENT
                          CHILDREN NODES))
          (CONTRADICT 10) (DROP 5 6 7 8 9 10) PROVE)))

(PROVE-LEMMA DL-UL-NO-PRESERVES-INSTANCE-OF-NO (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (N OLD NEW STATEMENT)
                (MEMBER STATEMENT (TREE-PRG TREE))
                (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                (UL (UP-LINKS (CDR (NODES TREE)) TREE) OLD)
                (NO (NODES TREE) TREE OLD)
                (MEMBER NODE (NODES TREE))
                (EQUAL (STATUS NODE NEW) 'STARTED))
           (AND (EQUAL (OUTSTANDING NODE NEW)
                       (NUMBER-NOT-REPORTED (CHILDREN NODE TREE)
                                            NODE NEW))
                (EQUAL (FOUND-VALUE NODE NEW)
                       (MIN-OF-REPORTED (CHILDREN NODE TREE)
                                        NODE NEW
                                        (NODE-VALUE NODE NEW))))))
    ((INSTRUCTIONS PROMOTE
          (USE-LEMMA NO-IMPLIES-INSTANCE-OF-NO
              ((NODES (NODES TREE)) (TREE TREE) (STATE OLD) (NODE NODE)))
          (DEMOTE 9) (DIVE 1) (DIVE 1) S-PROP UP (DIVE 2) (DIVE 1)
          (DIVE 1) (= (OUTSTANDING NODE OLD)) UP UP (DIVE 2) (DIVE 1)
          (= (FOUND-VALUE NODE OLD)) TOP PROMOTE
          (CLAIM (EQUAL (CAR STATEMENT) 'START) 0)
          (USE-LEMMA START-PRESERVES-INSTANCE-OF-NO
              ((TREE TREE) (OLD OLD) (NEW NEW) (NODE NODE)))
          (DEMOTE 11) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP
          (DROP 3 4 5 6 7 8 9) (PROVE (DISABLE TREE-PRG START)) (DROP 8)
          (DEMOTE 8)
          (DISABLE STATUS OUTSTANDING FOUND-VALUE NODE-VALUE START
                   ROOT-RECEIVE-REPORT RECEIVE-FIND RECEIVE-REPORT)
          S SPLIT (DROP 2 3 5 6 7 8 9 10 11 12) (DIVE 1) (DIVE 2)
          (= (CHILDREN (CAR TREE) TREE)) TOP
          (REWRITE DL-DOWN-LINKS-IMPLIES-DL-RFP (($NODES (NODES TREE))))
          PROVE (DROP 4 5 6 7 9 10 11) (PROVE (DISABLE TREE-PRG START))
          (DIVE 1) (DIVE 2) (= (CHILDREN (CAR TREE) TREE)) TOP
          (REWRITE DL-DOWN-LINKS-IMPLIES-DL-RFP (($NODES (NODES TREE))))
          (DROP 2 3 4 5 6 7 8 9 10) PROVE (DROP 4 5 6 7 9)
          (PROVE (DISABLE TREE-PRG START))
          (CLAIM (EQUAL (CAR STATEMENT) 'ROOT-RECEIVE-REPORT) 0)
          (USE-LEMMA ROOT-RECEIVE-REPORT-PRESERVES-INSTANCE-OF-NO
              ((TREE TREE) (CHILD (CAADDR STATEMENT)) (OLD OLD)
               (NEW NEW) (NODE NODE)))
          (DEMOTE 9) (DEMOTE 11) (DIVE 1) (DIVE 1) S-PROP (DIVE 1)
          (= * T 0) UP (DIVE 2) (DIVE 1) (= * T 0) TOP S
          (DROP 4 5 6 7 8 9)
          (PROVE (DISABLE TREE-PRG ROOT-RECEIVE-REPORT))
          (DROP 2 4 5 6 7 8 9) (PROVE (DISABLE TREE-PRG)) (DEMOTE 9)
          (CLAIM (EQUAL (CAR STATEMENT) 'RECEIVE-FIND) 0)
          (USE-LEMMA RECEIVE-FIND-PRESERVES-INSTANCE-OF-NO
              ((TREE TREE) (NODE (CADR STATEMENT))
               (PARENT (CAADDR STATEMENT)) (N NODE) (OLD OLD) (NEW NEW)))
          (DEMOTE 12) (DIVE 1) (DIVE 1) S-PROP S (DIVE 1) (= * T 0) UP
          (DIVE 2) (DIVE 1) (= * T 0) TOP S (DROP 4 5 6 7 8 9 10)
```

```
          (PROVE (DISABLE TREE-PRG RECEIVE-FIND))
          (DROP 2 4 5 6 7 8 9 10) (PROVE (DISABLE TREE-PRG))
          (USE-LEMMA RECEIVE-REPORT-PRESERVES-INSTANCE-OF-NO
              ((TREE TREE) (NODE (CADR STATEMENT))
               (CHILD (CAADDR STATEMENT)) (N NODE) (OLD OLD) (NEW NEW)))
          (DEMOTE 12) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP S S-PROP S
          SPLIT (DROP 4 5 6 7 8 12 13)
          (PROVE (DISABLE RECEIVE-REPORT TREE-PRG))
          (DROP 2 4 5 6 7 8 12) (PROVE (DISABLE TREE-PRG))
          (DROP 2 4 5 6 7 8) (PROVE (DISABLE TREE-PRG)))))

(PROVE-LEMMA DL-UL-NO-PRESERVES-NO-SUBLIST (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (TREE-PRG TREE))
                           (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                           (UL (UP-LINKS (CDR (NODES TREE)) TREE) OLD)
                           (NO (NODES TREE) TREE OLD)
                           (SUBLISTP SUBLIST (NODES TREE)))
                      (NO SUBLIST TREE NEW))
    ((INSTRUCTIONS (INDUCT (NO SUBLIST TREE NEW)) PROMOTE PROMOTE
          (DEMOTE 2) (DIVE 1) (DIVE 1) S-PROP (= * T 0) UP S TOP PROMOTE
          X (DROP 9)
          (USE-LEMMA DL-UL-NO-PRESERVES-INSTANCE-OF-NO
              ((TREE TREE) (OLD OLD) (NEW NEW) (STATEMENT STATEMENT)
               (NODE (CAR SUBLIST))))
          (DEMOTE 9) (DIVE 1) (DIVE 1) S-PROP (DIVE 1) (= * T 0) TOP S
          (DROP 3 4 5 6 7) PROVE (DROP 2 3 4 5 6 7) PROVE PROMOTE
          PROMOTE X)))

(PROVE-LEMMA INV-PRESERVES-INV (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (TREE-PRG TREE))
                           (INV TREE OLD))
                      (INV TREE NEW))
    ((INSTRUCTIONS (DISABLE NODES N MEMBER-TREE-PRG TREE-PRG TREEP) S
          SPLIT (REWRITE DL-UL-NO-PRESERVES-NO-SUBLIST)
          (REWRITE SUBLISTP-REFLEXIVE) (REWRITE DL-UL-NO-PRESERVES-UL)
          (REWRITE DL-PRESERVES-DL))))


(PROVE-LEMMA INV-IS-INVARIANT (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                            ‘(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS TREE))
                                             STATE)
                                  (NOT-STARTED (QUOTE ,(NODES TREE))
                                               STATE))
                            (TREE-PRG TREE))
                           (TREEP TREE))
                      (INVARIANT ‘(INV (QUOTE ,TREE) STATE)
                                 (TREE-PRG TREE)))
   ((INSTRUCTIONS PROMOTE
          (REWRITE UNLESS-PROVES-INVARIANT
                   (($IC (LIST ’AND
                               (LIST ’ALL-EMPTY
                                     (LIST ’QUOTE (ALL-CHANNELS TREE))
                                     ’STATE)
                               (LIST ’NOT-STARTED
                                     (LIST ’QUOTE (NODES TREE)) ’STATE)))))
          (BASH (DISABLE INV TREEP N MEMBER-TREE-PRG TREE-PRG))
          (BASH (DISABLE INV TREE-PRG)))))


(PROVE-LEMMA OUTSTANDING-NON-INCREASING (REWRITE)
 (IMPLIES (AND (TREEP TREE)
               (MEMBER STATEMENT (TREE-PRG TREE))
               (N OLD NEW STATEMENT)
               (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
```

```
               (MEMBER NODE (NODES TREE)))
          (NOT (LESSP (IF (EQUAL (STATUS NODE OLD) 'STARTED)
                          (OUTSTANDING NODE OLD)
                        (ADD1 (LENGTH (CHILDREN NODE TREE))))
                      (IF (EQUAL (STATUS NODE NEW) 'STARTED)
                          (OUTSTANDING NODE NEW)
                        (ADD1 (LENGTH (CHILDREN NODE TREE)))))))
    ((INSTRUCTIONS PROMOTE (CLAIM (EQUAL (CADR STATEMENT) NODE) 0)
         (CLAIM (EQUAL (CAR STATEMENT) 'RECEIVE-FIND) 0)
         (USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
                    ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))
                     (STATE OLD)
                     (DOWN-LINK (CONS (PARENT NODE TREE) NODE))))
         (DEMOTE 8) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP (DROP 4)
         (BASH T (DISABLE MIN ZEROP TREE-PRG)) (DROP 3 4)
         (CLAIM (LISTP (PARENT-REC 'TREE NODE TREE)) 0) (DROP 2 4 5)
         (PROVE (ENABLE PARENT-REC-CHILDREN-REC)) (CONTRADICT 6)
         (DIVE 1) (REWRITE SETP-TREE-UNIQUE-PARENT) TOP
         (PROVE (DISABLE TREE-PRG)) PROVE PROVE (DROP 4)
         (BASH T (DISABLE TREE-PRG MIN ZEROP)) PROMOTE (DIVE 1)
         (DIVE 2) = TOP DROP PROVE PROMOTE (DIVE 1) (DIVE 2) = TOP DROP
         PROVE (DROP 4) (BASH T (DISABLE TREE-PRG MIN ZEROP)))))

(PROVE-LEMMA TOTAL-OUTSTANDING-NON-INCREASING-SUBLIST (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER STATEMENT (TREE-PRG TREE))
                           (N OLD NEW STATEMENT)
                           (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                           (SUBLISTP SUBLIST (NODES TREE)))
                      (NOT (LESSP (TOTAL-OUTSTANDING SUBLIST TREE OLD)
                                  (TOTAL-OUTSTANDING SUBLIST TREE NEW))))
    ((INSTRUCTIONS (INDUCT (TOTAL-OUTSTANDING SUBLIST TREE OLD))
         PROMOTE PROMOTE (DEMOTE 2) (DIVE 1) (DIVE 1) S-PROP (= * T 0)
         UP (S-PROP IMPLIES) TOP PROMOTE
         (USE-LEMMA OUTSTANDING-NON-INCREASING
             ((TREE TREE) (STATEMENT STATEMENT) (OLD OLD) (NEW NEW)
              (NODE (CAR SUBLIST))))
         (DEMOTE 8) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP
         (DROP 2 3 4 5 6) (PROVE (DISABLE OUTSTANDING STATUS CHILDREN))
         (DROP 2 3 4 5 7) PROVE (DROP 2 3 4 5) PROVE
         (PROVE (DISABLE TREEP TREE-PRG MEMBER-TREE-PRG NODES N)))))

(PROVE-LEMMA TOTAL-OUTSTANDING-NON-INCREASING (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (MEMBER STATEMENT (TREE-PRG TREE))
                 (N OLD NEW STATEMENT)
                 (DL (DOWN-LINKS (NODES TREE) TREE) OLD))
            (NOT (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE OLD)
                        (TOTAL-OUTSTANDING (NODES TREE) TREE NEW))))
   ((DISABLE NODES TREEP TREE-PRG MEMBER-TREE-PRG)))

(PROVE-LEMMA POSITION-APPEND (REWRITE)
             (EQUAL (POSITION (APPEND A B) E)
                    (IF (MEMBER E A)
                        (POSITION A E)
                      (PLUS (LENGTH A)
                            (POSITION B E)))))

(PROVE-LEMMA PARENTS-POSITION-DECREASES (REWRITE)
      (IMPLIES (AND (MEMBER NODE (NODES-REC FLAG TREE))
                    (SETP (NODES-REC FLAG TREE))
                    (PROPER-TREE FLAG TREE)
                    (IF (EQUAL FLAG 'TREE)
                        (NOT (EQUAL (CAR TREE) NODE))
                      (NOT (MEMBER NODE (ROOTS TREE)))))
               (LESSP (POSITION (NODES-REC FLAG TREE)
                                (CAR (PARENT-REC FLAG NODE TREE)))
                      (POSITION (NODES-REC FLAG TREE)
                                NODE))))
```

```
(DEFN PARENT-TO-ROOT-INDUCTION (NODE TREE)
  (IF (AND (MEMBER NODE (NODES TREE))
           (SETP (NODES TREE))
           (PROPER-TREE 'TREE TREE))
      (IF (EQUAL (CAR TREE) NODE)
          T
        (PARENT-TO-ROOT-INDUCTION (PARENT NODE TREE) TREE))
    T)
  ((LESSP (POSITION (NODES TREE) NODE))))

(PROVE-LEMMA DL-AND-ALL-EMPTY-IMPLIES-ROOT-DEFINES-STATUS (REWRITE)
  (IMPLIES (AND (DL (DOWN-LINKS (NODES TREE) TREE) STATE)
                (ALL-EMPTY (DOWN-LINKS (NODES TREE) TREE) STATE)
                (SETP (NODES TREE))
                (PROPER-TREE 'TREE TREE)
                (MEMBER NODE (NODES TREE)))
           (EQUAL (CDR (ASSOC (CONS 'STATUS NODE) STATE))
                  (STATUS (CAR TREE) STATE)))
    ((INSTRUCTIONS (INDUCT (PARENT-TO-ROOT-INDUCTION NODE TREE)) PROVE
         (CLAIM (MEMBER NODE
                        (CHILDREN-REC 'TREE
                            (CAR (PARENT-REC 'TREE NODE TREE)) TREE))
                0)
         (USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
             ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE)) (STATE STATE)
              (DOWN-LINK (CONS (PARENT NODE TREE) NODE))))
         (USE-LEMMA ALL-EMPTY-IMPLIES-EMPTY
             ((CHANNELS (DOWN-LINKS (NODES TREE) TREE)) (STATE STATE)
              (CHANNEL (CONS (PARENT NODE TREE) NODE))))
         PROVE (CONTRADICT 1) (REWRITE PARENT-REC-CHILDREN-REC) PROVE
         PROVE)))


(DEFN SUFFIX (S L)
  (IF (LISTP L)
      (IF (EQUAL S L)
          T
        (SUFFIX S (CDR L)))
    (NOT (LISTP S))))


(PROVE-LEMMA SUFFIX-IMPLIES-SUFFIX-CDR (REWRITE)
             (IMPLIES (SUFFIX S L)
                      (SUFFIX (CDR S) L)))

(PROVE-LEMMA MEMBER-SUFFIX-MEMBER-LIST (REWRITE)
             (IMPLIES (AND (MEMBER E S)
                           (SUFFIX S L))
                      (MEMBER E L)))


(PROVE-LEMMA CHILDS-POSITION-INCREASES (REWRITE)
      (IMPLIES (AND (MEMBER NODE (NODES-REC FLAG TREE))
                    (SETP (NODES-REC FLAG TREE))
                    (PROPER-TREE FLAG TREE)
                    (MEMBER CHILD (CHILDREN-REC FLAG NODE TREE)))
               (LESSP (POSITION (NODES-REC FLAG TREE)
                                NODE)
                      (POSITION (NODES-REC FLAG TREE)
                                CHILD)))
      ((ENABLE PARENT-REC-CHILDREN-REC)
       (USE (PARENTS-POSITION-DECREASES (NODE CHILD)))))

(PROVE-LEMMA SETP-LIST-SETP-SUFFIX (REWRITE)
             (IMPLIES (AND (SETP L)
                           (SUFFIX S L))
                      (SETP S)))
```

```
(PROVE-LEMMA LATER-POSITIONS-ARE-IN-SUFFIX (REWRITE)
             (IMPLIES (AND (SETP L)
                           (SUFFIX S L)
                           (MEMBER X S)
                           (MEMBER Y L)
                           (LESSP (POSITION L X) (POSITION L Y)))
                      (MEMBER Y S)))



(DEFN ALL-DONE (NODES STATE)
  (IF (LISTP NODES)
      (IF (DONE (CAR NODES) STATE)
          (ALL-DONE (CDR NODES) STATE)
        F)
    T))

(PROVE-LEMMA ALL-DONE-IMPLIES-DONE (REWRITE)
             (IMPLIES (AND (ALL-DONE NODES STATE)
                           (MEMBER NODE NODES))
                      (DONE NODE STATE))
             ((DISABLE DONE)))

(PROVE-LEMMA ALL-DONE-IMPLIES-ALL-DONE-SUBLIST (REWRITE)
             (IMPLIES (AND (ALL-DONE NODES STATE)
                           (SUBLISTP SUBLIST NODES))
                      (ALL-DONE SUBLIST STATE))
             ((DISABLE DONE)))

(DEFN ULNKS (CHILDREN PARENT)
  (IF (LISTP CHILDREN)
      (CONS (CONS (CAR CHILDREN) PARENT)
            (ULNKS (CDR CHILDREN) PARENT))
    NIL))

(PROVE-LEMMA ALL-DONE-AND-ALL-EMPTY-IMPLIES-NUMBER-NOT-REPORTED-0
 (REWRITE)
  (IMPLIES (AND (ALL-DONE CHILDREN STATE)
                (ALL-EMPTY (ULNKS CHILDREN PARENT) STATE))
           (EQUAL (NUMBER-NOT-REPORTED CHILDREN PARENT STATE)
                  0))
             ((DISABLE DONE)))

(PROVE-LEMMA ALL-EMPTY-IMPLIES-ALL-EMPTY-SUBLIST (REWRITE)
             (IMPLIES (AND (ALL-EMPTY CHANNELS STATE)
                           (SUBLISTP SUBLIST CHANNELS))
                      (ALL-EMPTY SUBLIST STATE))
             ((DISABLE CHANNEL)))

(PROVE-LEMMA SUBLIST-ULNKS (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                           (SUBLISTP SUBLIST (CHILDREN PARENT TREE))
                           (SETP (NODES TREE)))
                      (SUBLISTP (ULNKS SUBLIST PARENT)
                                (UP-LINKS (CDR (NODES TREE)) TREE)))
             ((INDUCT (ULNKS SUBLIST PARENT))))


(PROVE-LEMMA CHILD-OF-NODE-IN-SUFFIX-IS-IN-SUFFIX (REWRITE)
             (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                           (SETP (NODES TREE))
                           (MEMBER CHILD (CHILDREN NODE TREE))
                           (SUFFIX NODES (NODES TREE))
                           (MEMBER NODE NODES))
                      (MEMBER CHILD (CDR NODES)))
    ((INSTRUCTIONS PROMOTE (CLAIM (MEMBER CHILD NODES)) PROVE)))

(PROVE-LEMMA CHILDREN-ARE-SUFFIX-OF-SUBLIST-GENERALIZED (REWRITE)
```

```
            (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                          (SETP (NODES TREE))
                          (SUFFIX NODES (NODES TREE))
                          (MEMBER NODE NODES)
                          (SUBLISTP SUBLIST
                                    (CHILDREN-REC 'TREE NODE TREE)))
                     (SUBLISTP SUBLIST (CDR NODES)))
    ((INSTRUCTIONS (INDUCT (LENGTH SUBLIST)) PROMOTE PROMOTE X (DIVE 1)
         (REWRITE CHILD-OF-NODE-IN-SUFFIX-IS-IN-SUFFIX
                  (($NODE NODE) ($TREE TREE)))
         TOP PROVE PROVE PROVE)))

(PROVE-LEMMA ALL-NODES-ARE-DONE (REWRITE)
  (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                (SETP (NODES TREE))
                (ALL-EMPTY (DOWN-LINKS (NODES TREE) TREE) STATE)
                (ALL-EMPTY (UP-LINKS (CDR (NODES TREE)) TREE) STATE)
                (DL (DOWN-LINKS (NODES TREE) TREE) STATE)
                (UL (UP-LINKS (CDR (NODES TREE)) TREE) STATE)
                (NO (NODES TREE) TREE STATE)
                (EQUAL (STATUS (CAR TREE) STATE) 'STARTED)
                (SUFFIX NODES (NODES TREE)))
           (ALL-DONE NODES STATE))
    ((INSTRUCTIONS (INDUCT (LENGTH NODES)) PROMOTE PROMOTE (DEMOTE 2)
         (DIVE 1) (DIVE 1) S-PROP (REWRITE SUFFIX-IMPLIES-SUFFIX-CDR)
         UP S TOP PROMOTE X (DIVE 1) (DIVE 1)
         (REWRITE DL-AND-ALL-EMPTY-IMPLIES-ROOT-DEFINES-STATUS
                  (($TREE TREE)))
         = TOP S (DIVE 1) (DIVE 1)
         (REWRITE NO-IMPLIES-INSTANCE-OF-NO
                  (($NODES (NODES TREE)) ($TREE TREE)))
         (REWRITE ALL-DONE-AND-ALL-EMPTY-IMPLIES-NUMBER-NOT-REPORTED-0)
         TOP S
         (REWRITE ALL-DONE-IMPLIES-ALL-DONE-SUBLIST
                  (($NODES (CDR NODES))))
         (REWRITE CHILDREN-ARE-SUFFIX-OF-SUBLIST-GENERALIZED
                  (($TREE TREE) ($NODE (CAR NODES))))
         X (REWRITE SUBLISTP-REFLEXIVE)
         (REWRITE ALL-EMPTY-IMPLIES-ALL-EMPTY-SUBLIST
                  (($CHANNELS (UP-LINKS (CDR (NODES TREE)) TREE))))
         (REWRITE SUBLIST-ULNKS) S (REWRITE SUBLISTP-REFLEXIVE)
         (REWRITE MEMBER-SUFFIX-MEMBER-LIST (($S NODES))) X S (DIVE 1)
         (REWRITE DL-AND-ALL-EMPTY-IMPLIES-ROOT-DEFINES-STATUS
                  (($TREE TREE)))
         = TOP S (REWRITE MEMBER-SUFFIX-MEMBER-LIST (($S NODES))) X
         (REWRITE MEMBER-SUFFIX-MEMBER-LIST (($S NODES))) X PROMOTE X)))


(PROVE-LEMMA ALL-DONE-IMPLIES-TOTAL-OUTSTANDING-0 (REWRITE)
             (IMPLIES (ALL-DONE NODES STATE)
                      (EQUAL (TOTAL-OUTSTANDING NODES TREE STATE)
                             0)))

(PROVE-LEMMA ALL-EMPTY-ROOT-STARTED-IMPLIES-TOTAL-OUTSTANDING-0 (REWRITE)
  (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                (SETP (NODES TREE))
                (INV TREE STATE)
                (ALL-EMPTY (DOWN-LINKS (NODES TREE) TREE) STATE)
                (ALL-EMPTY (UP-LINKS (CDR (NODES TREE)) TREE) STATE)
                (EQUAL (STATUS (CAR TREE) STATE) 'STARTED))
           (EQUAL (TOTAL-OUTSTANDING (NODES TREE) TREE STATE)
                  0))
    ((INSTRUCTIONS PROMOTE (DEMOTE 3) (DIVE 1) X-DUMB TOP PROMOTE
         (DIVE 1) (REWRITE ALL-DONE-IMPLIES-TOTAL-OUTSTANDING-0) TOP S
         (REWRITE ALL-NODES-ARE-DONE (($TREE TREE))) X)))

(DEFN FULL-CHANNEL (CHANNELS STATE)
  (IF (LISTP CHANNELS)
      (IF (EMPTY (CAR CHANNELS) STATE)
```

```
          (FULL-CHANNEL (CDR CHANNELS) STATE)
        (CAR CHANNELS))
    F))

(PROVE-LEMMA NOT-ALL-EMPTY-IMPLIES-FULL-CHANNEL-FULL (REWRITE)
  (IMPLIES (AND (NOT (ALL-EMPTY CHANNELS STATE))
                (NOT (MEMBER F CHANNELS)))
           (AND (LISTP (CDR (ASSOC (FULL-CHANNEL CHANNELS STATE)
                                   STATE)))
                (MEMBER (FULL-CHANNEL CHANNELS STATE) CHANNELS)
                (FULL-CHANNEL CHANNELS STATE))))

(PROVE-LEMMA NOT-TOTAL-OUTSTANDING-0-IMPLIES-FULL-CHANNEL (REWRITE)
  (IMPLIES (AND (PROPER-TREE 'TREE TREE)
                (SETP (NODES TREE))
                (INV TREE STATE)
                (OR (EQUAL (STATUS (CAR TREE) STATE) 'STARTED)
                    (EQUAL (STATUS (CAR TREE) STATE) 'NOT-STARTED))
                (NOT (EQUAL (TOTAL-OUTSTANDING (NODES TREE)
                                               TREE STATE)
                            0)))
           (OR (EQUAL (STATUS (CAR TREE) STATE) 'NOT-STARTED)
               (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE) STATE)
               (FULL-CHANNEL (UP-LINKS (CDR (NODES TREE)) TREE)
                             STATE)))
    ((INSTRUCTIONS PROMOTE (CONTRADICT 5) (DIVE 1)
         (REWRITE ALL-EMPTY-ROOT-STARTED-IMPLIES-TOTAL-OUTSTANDING-0)
         TOP S SPLIT (CONTRADICT 6)
         (REWRITE NOT-ALL-EMPTY-IMPLIES-FULL-CHANNEL-FULL)
         (DROP 3 4 5 6 7) PROVE SPLIT (CONTRADICT 7)
         (REWRITE NOT-ALL-EMPTY-IMPLIES-FULL-CHANNEL-FULL)
         (DROP 3 4 5 6 7) PROVE SPLIT)))


(PROVE-LEMMA STATUS-ROOT-BECOMES-STARTED-OR-UNCHANGED (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (MEMBER STATEMENT (TREE-PRG TREE))
                           (N OLD NEW STATEMENT))
                      (OR (EQUAL (STATUS (CAR TREE) NEW) 'STARTED)
                          (EQUAL (STATUS (CAR TREE) NEW)
                                 (STATUS (CAR TREE) OLD))))
    ((INSTRUCTIONS PROMOTE
         (CLAIM (EQUAL (CADR STATEMENT) (CAR TREE)) 0)
         (PROVE (DISABLE TREE-PRG MIN)) (PROVE (DISABLE TREE-PRG MIN)))))


(PROVE-LEMMA ROOT-STARTED-OR-NOT-STARTED-IS-INVARIANT (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                            `(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS TREE))
                                             STATE)
                                  (NOT-STARTED (QUOTE ,(NODES TREE))
                                               STATE))
                            (TREE-PRG TREE))
                           (TREEP TREE))
                      (INVARIANT `(OR (EQUAL (STATUS (QUOTE ,(CAR TREE))
                                                     STATE)
                                             'STARTED)
                                      (EQUAL (STATUS (QUOTE ,(CAR TREE))
                                                     STATE)
                                             'NOT-STARTED))
                                 (TREE-PRG TREE)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE UNLESS-PROVES-INVARIANT
                  (($IC (LIST 'AND
                              (LIST 'ALL-EMPTY
                                    (LIST 'QUOTE (ALL-CHANNELS TREE))
                                    'STATE)
                              (LIST 'NOT-STARTED
                                    (LIST 'QUOTE (NODES TREE)) 'STATE))))
```

```
         (REWRITE HELP-PROVE-UNLESS) (DROP 1)
         (GENERALIZE
             (((EU (LIST 'OR
                         (CONS 'EQUAL
                               (CONS (CONS 'STATUS
                                      (CONS (LIST 'QUOTE (CAR TREE))
                                       '(STATE)))
                                     '('STARTED)))
                         (CONS 'EQUAL
                               (CONS (CONS 'STATUS
                                      (CONS (LIST 'QUOTE (CAR TREE))
                                       '(STATE)))
                                     '('NOT-STARTED))))
                   (TREE-PRG TREE) '(FALSE))
               STATEMENT)
              ((OLDU (LIST 'OR
                         (CONS 'EQUAL
                               (CONS (CONS 'STATUS
                                      (CONS (LIST 'QUOTE (CAR TREE))
                                       '(STATE)))
                                     '('STARTED)))
                         (CONS 'EQUAL
                               (CONS (CONS 'STATUS
                                      (CONS (LIST 'QUOTE (CAR TREE))
                                       '(STATE)))
                                     '('NOT-STARTED))))
                   (TREE-PRG TREE) '(FALSE))
               OLD)
              ((NEWU (LIST 'OR
                         (CONS 'EQUAL
                               (CONS (CONS 'STATUS
                                      (CONS (LIST 'QUOTE (CAR TREE))
                                       '(STATE)))
                                     '('STARTED)))
                         (CONS 'EQUAL
                               (CONS (CONS 'STATUS
                                      (CONS (LIST 'QUOTE (CAR TREE))
                                       '(STATE)))
                                     '('NOT-STARTED))))
                   (TREE-PRG TREE) '(FALSE))
               NEW)))
         (USE-LEMMA STATUS-ROOT-BECOMES-STARTED-OR-UNCHANGED
             ((TREE TREE) (OLD OLD) (NEW NEW)))
         (PROVE (DISABLE TREE-PRG MEMBER-TREE-PRG N))
         (PROVE (DISABLE TREE-PRG)
                (EXPAND (NOT-STARTED (NODES-REC 'TREE TREE)
                            (S (TREE-PRG TREE) 0)))))))


(PROVE-LEMMA TOTAL-OUTSTANDING-DECREASES-SUBLIST (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                (N OLD NEW STATEMENT)
                (MEMBER STATEMENT (TREE-PRG TREE))
                (SUBLISTP NODES (NODES TREE))
                (MEMBER NODE NODES)
                (LESSP (IF (EQUAL (STATUS NODE NEW) 'STARTED)
                           (OUTSTANDING NODE NEW)
                         (ADD1 (LENGTH (CHILDREN NODE TREE))))
                       (IF (EQUAL (STATUS NODE OLD) 'STARTED)
                           (OUTSTANDING NODE OLD)
                         (ADD1 (LENGTH (CHILDREN NODE TREE))))))
           (LESSP (TOTAL-OUTSTANDING NODES TREE NEW)
                  (TOTAL-OUTSTANDING NODES TREE OLD)))
    ((INSTRUCTIONS (INDUCT (MEMBER NODE NODES))
         (PROVE (DISABLE TREE-PRG MEMBER-TREE-PRG TREEP STATUS
                         OUTSTANDING CHILDREN))
         (PROVE (DISABLE TREE-PRG MEMBER-TREE-PRG TREEP STATUS
                         OUTSTANDING CHILDREN NODES N)
```

```
                (EXPAND (TOTAL-OUTSTANDING NODES TREE OLD)
                        (TOTAL-OUTSTANDING NODES TREE NEW)))
         PROMOTE PROMOTE (DEMOTE 3) (DIVE 1) (DIVE 1) S-PROP (= * T 0)
         UP S TOP PROMOTE (DROP 8 9)
         (USE-LEMMA OUTSTANDING-NON-INCREASING
             ((TREE TREE) (STATEMENT STATEMENT) (OLD OLD) (NEW NEW)
              (NODE (CAR NODES))))
         (PROVE (DISABLE TREE-PRG MEMBER-TREE-PRG TREEP STATUS
                         OUTSTANDING CHILDREN NODES N)
                (EXPAND (TOTAL-OUTSTANDING NODES TREE OLD)
                        (TOTAL-OUTSTANDING NODES TREE NEW)))
         (DROP 3 4 5 6)
         (PROVE (DISABLE STATUS OUTSTANDING NODES CHILDREN)))))

(DEFN TOU (OLD NEW NODE TREE)
  (LESSP (IF (EQUAL (STATUS NODE NEW) 'STARTED)
             (OUTSTANDING NODE NEW)
           (ADD1 (LENGTH (CHILDREN NODE TREE))))
         (IF (EQUAL (STATUS NODE OLD) 'STARTED)
             (OUTSTANDING NODE OLD)
           (ADD1 (LENGTH (CHILDREN NODE TREE))))))


(PROVE-LEMMA TOTAL-OUTSTANDING-DECREASES (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (DL (DOWN-LINKS (NODES TREE) TREE) OLD)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (TREE-PRG TREE))
                           (MEMBER NODE (NODES TREE))
                           (TOU OLD NEW NODE TREE))
                      (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE NEW)
                             (TOTAL-OUTSTANDING (NODES TREE) TREE OLD)))
   ((INSTRUCTIONS PROMOTE (DEMOTE 6) (DIVE 1) X TOP PROMOTE
         (PROVE (DISABLE TREE-PRG MEMBER-TREE-PRG N TREEP NODES)))))

(PROVE-LEMMA START-DECREASES-TOU (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (EQUAL (STATUS (CAR TREE) OLD) 'NOT-STARTED)
                (N OLD NEW (LIST 'START (CAR TREE)
                                 (RFP (CAR TREE)
                                      (CHILDREN (CAR TREE) TREE)))))
           (TOU OLD NEW (CAR TREE) TREE)))

(PROVE-LEMMA OTHERS-PRESERVE-ROOT-NOT-STARTED (REWRITE)
              (IMPLIES (AND (TREEP TREE)
                            (N OLD NEW STATEMENT)
                            (MEMBER STATEMENT (TREE-PRG TREE))
                            (NOT (EQUAL STATEMENT
                                        (LIST 'START (CAR TREE)
                                              (RFP (CAR TREE)
                                                   (CHILDREN (CAR TREE)
                                                             TREE))))))
                       (EQUAL (CDR (ASSOC (CONS 'STATUS (CAR TREE)) NEW))
                              (STATUS (CAR TREE) OLD)))
((INSTRUCTIONS
  (BASH T
        (DISABLE TREE-PRG MIN ZEROP PARENT CHILDREN NODES))
  PROMOTE
  (CLAIM (AND (NUMBERP W)
              (NUMBERP (PARENT W (CONS X Z))))
         0)
  (DIVE 1)
  (DIVE 1)
  (REWRITE ABOUT-UC
   (($B OLD)
    ($EXCPT
     (CONS
        (CONS (PARENT W (CONS X Z)) W)
        (CONS (CONS W (PARENT W (CONS X Z)))
```

```
            (CONS (CONS 'STATUS W)
                  (CONS (CONS 'FOUND-VALUE W)
                        (CONS (CONS 'OUTSTANDING W)
                              (RFP W
                                   (CHILDREN W (CONS X Z))))))))))
TOP S
(DROP 4 5 6 7 8 9 11 12)
(BASH T
      (DISABLE PARENT CHILDREN NODES))
PROMOTE
(CONTRADICT 7)
(DROP 3 4 6 7)
PROVE
(CONTRADICT 14)
(DROP 4 5 6 7 8 9 10 11 12 14)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
      (CONS (PARENT W (CONS X Z)) W)
      (CONS (CONS W (PARENT W (CONS X Z)))
            (CONS (CONS 'STATUS W)
                  (CONS (CONS 'FOUND-VALUE W)
                        (CONS (CONS 'OUTSTANDING W)
                              (RFP W
                                   (CHILDREN W (CONS X Z))))))))))
TOP S
(CLAIM (AND (NUMBERP W)
            (NUMBERP (PARENT W (CONS X Z))))
       0)
(DROP 4 5 6 7 8 10 11)
(BASH T
      (DISABLE CHILDREN PARENT NODES))
PROMOTE
(CONTRADICT 7)
(DROP 3 4 5 6 7)
PROVE
(CONTRADICT 13)
(DROP 4 5 6 7 8 9 10 11)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS Z1 W)
                        (CONS W (PARENT W (CONS X Z)))
                        (CONS 'OUTSTANDING W)
                        (CONS 'FOUND-VALUE W)))))
TOP S
(CLAIM (AND (NUMBERP Z1)
            (NUMBERP W)
            (NOT (EQUAL X W)))
       0)
(DROP 1 2 3 4 5 6 7 8 9 10 11 12)
PROVE
(CONTRADICT 13)
(DROP 4 5 6 7 8 9 10 13)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS Z1 W)
                        (CONS W (PARENT W (CONS X Z)))
                        (CONS 'OUTSTANDING W)
                        (CONS 'FOUND-VALUE W)))))
```

```
TOP S
(CLAIM (AND (NUMBERP Z1) (NUMBERP W))
       0)
(DROP 1 2 3 4 5 6 7 8 9 10 11 12)
(PROVE (DISABLE PARENT))
(CONTRADICT 13)
(DROP 4 5 6 7 8 9 10 13)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS Z1 W)
                        (CONS W (PARENT W (CONS X Z)))
                        (CONS 'OUTSTANDING W)
                        (CONS 'FOUND-VALUE W)))))
TOP S
(CLAIM (AND (NUMBERP Z1) (NUMBERP W))
       0)
(DROP 1 2 3 4 5 6 7 8 9 10 11 12)
(PROVE (DISABLE PARENT))
(CONTRADICT 13)
(DROP 4 5 6 7 8 9 10 13)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS Z1 W)
                        (CONS W (PARENT W (CONS X Z)))
                        (CONS 'OUTSTANDING W)
                        (CONS 'FOUND-VALUE W)))))
TOP S
(CLAIM (AND (NUMBERP Z1) (NUMBERP W))
       0)
(DROP 1 2 3 4 5 6 7 8 9 10 11 12)
(PROVE (DISABLE PARENT))
(CONTRADICT 13)
(DROP 4 5 6 7 8 9 10 13)
PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS V X)
                        (CONS 'OUTSTANDING X)
                        (CONS 'FOUND-VALUE X)))))
TOP S
(CLAIM (NUMBERP V) 0)
(DROP 1 2 3 4 5 6 7 8 9)
PROVE
(CONTRADICT 10)
(DROP 4 5 6 7 8 10)
PROVE)))


(PROVE-LEMMA ROOT-RECEIVE-REPORT-DECREASES-TOU (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (LISTP (CHANNEL (CONS CHILD (CAR TREE)) OLD))
                           (MEMBER CHILD (CHILDREN (CAR TREE) TREE))
                           (N OLD NEW
                              (LIST 'ROOT-RECEIVE-REPORT (CAR TREE)
                                    (CONS CHILD (CAR TREE))))
                           (INV TREE OLD))
                      (TOU OLD NEW (CAR TREE) TREE))
    ((INSTRUCTIONS PROMOTE
         (CLAIM (EQUAL (STATUS (CAR TREE) OLD) 'STARTED) 0)
         (CLAIM (NOT (ZEROP (OUTSTANDING (CAR TREE) OLD))) 0) (DROP 5)
         PROVE
```

```
        (USE-LEMMA NUMBER-NOT-REPORTED-0-IMPLIES
            ((CHILDREN (CHILDREN (CAR TREE) TREE)) (PARENT (CAR TREE))
             (STATE OLD) (NODE CHILD)))
        (CONTRADICT 2) (DROP 2) (DEMOTE 6) (DIVE 1) (DIVE 1) S
        (REWRITE NO-IMPLIES-INSTANCE-OF-NO
                 (($TREE TREE) ($NODES (NODES TREE))))
        TOP (DROP 4) DEMOTE S (DEMOTE 4) S (DROP 2 3 4 5 6) PROVE
        (CONTRADICT 6)
        (USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL
            ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))
             (UPLINK (CONS CHILD (CAR TREE))) (STATE OLD)))
        (USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
            ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE)) (STATE OLD)
             (DOWN-LINK (CONS (CAR TREE) CHILD))))
        (DROP 4 6) (DEMOTE 5) (DIVE 1) (DIVE 1) (= * T 0) TOP
        (DEMOTE 5) (DIVE 1) (DIVE 1) (= * T 0) TOP (DROP 1 3 4) DEMOTE
        PROVE (DROP 2) (DIVE 1) (DIVE 1) (= T) TOP (DROP 3) S-PROP
        (DIVE 1) (REWRITE MEMBER-DOWN-LINKS) TOP PROVE (DIVE 1)
        (DIVE 1) (DROP 5) (= T) UP (DIVE 2) (REWRITE MEMBER-UP-LINKS)
        (DROP 2 4) (DIVE 2) (DIVE 1) (DIVE 2) S (DIVE 1)
        (REWRITE PARENT-OF-CHILD (($PARENT (CAR TREE)))) TOP PROVE
        PROVE PROVE PROVE)))


(PROVE-LEMMA OTHERS-PRESERVE-UP-TO-ROOT-FULL (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (LISTP (CHANNEL (CONS CHILD (CAR TREE)) OLD))
                (MEMBER CHILD (CHILDREN (CAR TREE) TREE))
                (MEMBER STATEMENT (TREE-PRG TREE))
                (NOT (EQUAL STATEMENT
                            (LIST 'ROOT-RECEIVE-REPORT (CAR TREE)
                                  (CONS CHILD (CAR TREE)))))
                (N OLD NEW STATEMENT))
           (LISTP (CDR (ASSOC (CONS CHILD (CAR TREE)) NEW))))
((INSTRUCTIONS PROMOTE
  (CLAIM (AND (NUMBERP CHILD)
              (NUMBERP (CAR TREE)))
         0)
  (BASH T
        (DISABLE TREE-PRG MIN ZEROP PARENT CHILDREN NODES))
  PROMOTE
  (DIVE 1)
  (DIVE 1)
  (REWRITE ABOUT-UC
     (($B OLD)
      ($EXCPT (CONS (CONS 'STATUS W)
                    (CONS (CONS 'FOUND-VALUE W)
                          (CONS (CONS 'OUTSTANDING W)
                                (RFP W (CHILDREN W (CONS W Z))))))))
  TOP S
  (CLAIM (NOT (EQUAL W CHILD)) 0)
  (DROP 1 2 3 4 5 6 7 8 9 10 11)
  (PROVE (DISABLE CHILDREN))
  (CONTRADICT 14)
  (DROP 4 6 7 8 9 10 11)
  PROVE PROMOTE
  (CLAIM (AND (EQUAL D CHILD)
              (EQUAL X (PARENT D (CONS X Z))))
         0)
  (PROVE (DISABLE PARENT CHILDREN NODES))
  (DIVE 1)
  (DIVE 1)
  (REWRITE ABOUT-UC
   (($B OLD)
    ($EXCPT
     (CONS
        (CONS (PARENT D (CONS X Z)) D)
        (CONS (CONS D (PARENT D (CONS X Z)))
              (CONS (CONS 'STATUS D)
```

```
                  (CONS (CONS 'FOUND-VALUE D)
                        (CONS (CONS 'OUTSTANDING D)
                              (RFP D
                                   (CHILDREN D (CONS X Z))))))))))
TOP S
(DROP 4 7 8 9 10 11 12 13 14 15)
(BASH T
      (DISABLE PARENT CHILDREN NODES))
PROMOTE
(DIVE 1)
S
(REWRITE PARENT-NOT-GRANDCHILD)
TOP S X
(DEMOTE 1 2)
DROP PROVE
(DEMOTE 6)
S PROMOTE
(CONTRADICT 7)
(DEMOTE 1 2)
DROP PROVE PROMOTE
(DIVE 1)
(DIVE 1)
(CLAIM (EQUAL D CHILD) 0)
(CLAIM (EQUAL X (PARENT D (CONS X Z)))
       0)
(DIVE 1)
(DIVE 1)
= UP
(DIVE 2)
= UP UP UP = TOP S TOP
(CONTRADICT 18)
(DIVE 2)
S
(DIVE 1)
(REWRITE PARENT-OF-CHILD
         (($PARENT X)))
TOP S X
(DEMOTE 1)
S
(DIVE 1)
= TOP
(DEMOTE 5)
S
(CLAIM (NOT (EQUAL D X)) 0)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
      (CONS (PARENT D (CONS X Z)) D)
      (CONS (CONS D (PARENT D (CONS X Z)))
            (CONS (CONS 'STATUS D)
                  (CONS (CONS 'FOUND-VALUE D)
                        (CONS (CONS 'OUTSTANDING D)
                              (RFP D
                                   (CHILDREN D (CONS X Z))))))))))
TOP S
(DROP 4 7 8 9 10 11 12 13 14)
(PROVE (DISABLE NODES CHILDREN PARENT))
TOP
(CONTRADICT 18)
(DEMOTE 1 6)
DROP PROVE PROMOTE
(CLAIM (EQUAL D CHILD) 0)
(CLAIM (EQUAL X (PARENT D (CONS X Z)))
       0)
(DIVE 1)
(DIVE 1)
(DIVE 1)
(DIVE 1)
```

```
= UP
(DIVE 2)
= UP UP UP = TOP S
(CONTRADICT 18)
(DIVE 2)
S
(DIVE 1)
(REWRITE PARENT-OF-CHILD
         (($PARENT X)))
TOP S X
(DEMOTE 1)
S
(DIVE 1)
= TOP
(DEMOTE 5)
S
(CLAIM (NOT (EQUAL C CHILD)) 0)
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS C D)
                        (CONS D (PARENT D (CONS X Z)))
                        (CONS 'OUTSTANDING D)
                        (CONS 'FOUND-VALUE D)))))
TOP S
(DROP 5 8 9 10 11 12 13 14)
(PROVE (DISABLE CHILDREN PARENT NODES))
(CLAIM (EQUAL D X) 0)
(CONTRADICT 6)
(DIVE 1)
(DIVE 1)
= TOP
(DEMOTE 1)
DROP PROVE
(CONTRADICT 19)
(DIVE 1)
(= * (PARENT CHILD (CONS X Z)) 0)
TOP
(DIVE 1)
S
(DIVE 1)
(REWRITE PARENT-OF-CHILD
         (($PARENT X)))
TOP S X
(DEMOTE 1)
S
(DEMOTE 5)
S
(DIVE 2)
S
(DIVE 1)
(DIVE 2)
= UP
(REWRITE PARENT-OF-CHILD
         (($PARENT D)))
TOP S X
(DEMOTE 1)
S
(DEMOTE 7)
S PROMOTE
(CLAIM (EQUAL D CHILD) 0)
(CLAIM (EQUAL (PARENT D (CONS X Z)) X)
       0)
(DIVE 1)
(DIVE 1)
(DIVE 1)
(DIVE 1)
= UP
```

```
(DIVE 2)
= TOP
(DIVE 1)
= TOP S
(CONTRADICT 18)
(DIVE 1)
(DIVE 1)
= UP S
(DIVE 1)
(REWRITE PARENT-OF-CHILD
         (($PARENT X)))
TOP S X
(DEMOTE 1)
S
(DEMOTE 5)
S
(CLAIM (EQUAL C CHILD) 0)
(CONTRADICT 6)
(DIVE 1)
(DIVE 1)
(= * X 0)
TOP
(DEMOTE 1)
DROP PROVE
(DIVE 1)
(= * (PARENT CHILD (CONS X Z)) 0)
S
(DIVE 1)
(REWRITE PARENT-OF-CHILD
         (($PARENT X)))
TOP S X
(DEMOTE 1)
S
(DEMOTE 5)
S
(DIVE 2)
(DIVE 1)
= UP S
(DIVE 1)
(REWRITE PARENT-OF-CHILD
         (($PARENT D)))
TOP S X
(DEMOTE 1)
S
(DEMOTE 7)
S
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS C D)
                        (CONS D (PARENT D (CONS X Z)))
                        (CONS 'OUTSTANDING D)
                        (CONS 'FOUND-VALUE D)))))
TOP S
(DROP 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
PROVE PROMOTE
(CLAIM (EQUAL C CHILD) 0)
(CONTRADICT 17)
(DROP 4 8 9 10 11 12 13 14 17)
(BASH T)
PROMOTE
(CONTRADICT 8)
(DIVE 1)
(REWRITE PARENT-REC-CHILDREN-REC)
TOP PROVE
(CLAIM (EQUAL D CHILD) 0)
(DIVE 1)
(DIVE 1)
```

```
(DIVE 1)
(DIVE 1)
= UP
(DIVE 2)
(= * (PARENT D (CONS X Z)) 0)
UP UP UP = TOP S
(DIVE 2)
(DIVE 1)
= UP S
(DIVE 1)
(REWRITE PARENT-OF-CHILD
         (($PARENT X)))
TOP S X
(DEMOTE 1)
S
(DEMOTE 5)
S
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS C D)
                        (CONS D (PARENT D (CONS X Z)))
                        (CONS 'OUTSTANDING D)
                        (CONS 'FOUND-VALUE D)))))
TOP S
(DROP 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
(PROVE (DISABLE PARENT))
PROMOTE
(CLAIM (EQUAL C CHILD) 0)
(DROP 4 8 9 10 11 12 13 14)
(DEMOTE 6)
(DIVE 1)
S
(REWRITE PARENT-REC-CHILDREN-REC)
TOP
(BASH T)
(CLAIM (EQUAL D CHILD) 0)
(DIVE 1)
(DIVE 1)
(DIVE 1)
(DIVE 1)
= UP
(DIVE 2)
(= * (PARENT D (CONS X Z)) 0)
UP UP UP = TOP S
(DEMOTE 4)
(DIVE 1)
(DIVE 1)
(DIVE 1)
(DIVE 1)
(DIVE 2)
(= * (PARENT D (CONS X Z)) 0)
UP
(DIVE 1)
= TOP S
(DIVE 2)
S
(DIVE 1)
(REWRITE PARENT-OF-CHILD
         (($PARENT X)))
TOP S X
(DEMOTE 1)
S
(DIVE 1)
= TOP
(DEMOTE 4)
S
(DIVE 2)
```

```
  S
  (DIVE 1)
  (REWRITE PARENT-OF-CHILD
           (($PARENT X)))
  TOP S X
  (DEMOTE 1)
  S
  (DIVE 1)
  = TOP
  (DEMOTE 5)
  S
  (DIVE 1)
  (DIVE 1)
  (REWRITE ABOUT-UC
           (($B OLD)
            ($EXCPT (LIST (CONS C D)
                          (CONS D (PARENT D (CONS X Z)))
                          (CONS 'OUTSTANDING D)
                          (CONS 'FOUND-VALUE D)))))
  TOP S
  (DROP 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
  PROVE
  (CONTRADICT 7)
  (DROP 2 4 5 6 7)
  PROVE)))

(PROVE-LEMMA RECEIVE-FIND-DECREASES-TOU (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (LISTP (CHANNEL (CONS (PARENT NODE TREE) NODE) OLD))
                 (MEMBER NODE (CDR (NODES TREE)))
                 (N OLD NEW (LIST 'RECEIVE-FIND NODE
                                  (CONS (PARENT NODE TREE) NODE)
                                  (CONS NODE (PARENT NODE TREE))
                                  (RFP NODE
                                       (CHILDREN NODE TREE))))
                 (INV TREE OLD))
            (TOU OLD NEW NODE TREE))
    ((INSTRUCTIONS PROMOTE
         (CLAIM (AND (EQUAL (STATUS NODE OLD) 'NOT-STARTED)
                     (EQUAL (HEAD (CONS (PARENT NODE TREE) NODE) OLD)
                            'FIND))
                0)
         (DROP 5) (PROVE (DISABLE CHILDREN PARENT NODES))
         (CONTRADICT 6)
         (USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
             ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE))
              (DOWN-LINK (CONS (PARENT NODE TREE) NODE)) (STATE OLD)))
         (DEMOTE 7) (DIVE 1) (DIVE 1) (= * T 0) (DROP 1 3 4 5 6) TOP
         DEMOTE (PROVE (DISABLE PARENT NODES)) S SPLIT (DROP 2 4 5 6 7)
         (REWRITE MEMBER-DOWN-LINKS) (DIVE 2) (DIVE 1) S
         (REWRITE PARENT-REC-CHILDREN-REC) TOP S
         (CLAIM (MEMBER NODE (NODES-REC 'TREE TREE))) (DIVE 1)
         (REWRITE NODE-HAS-PARENT) TOP S (DIVE 2)
         (REWRITE SETP-TREE-UNIQUE-PARENT) TOP PROVE PROVE PROVE PROVE
         PROVE PROVE)))

(PROVE-LEMMA OTHERS-PRESERVE-DOWN-TO-NODE-FULL (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (LISTP (CHANNEL (CONS (PARENT NODE TREE)
                                      NODE) OLD))
                (MEMBER NODE (CDR (NODES TREE)))
                (MEMBER STATEMENT (TREE-PRG TREE))
                (NOT (EQUAL STATEMENT
                            (LIST 'RECEIVE-FIND NODE
                                  (CONS (PARENT NODE TREE) NODE)
                                  (CONS NODE (PARENT NODE TREE))
                                  (RFP NODE
                                       (CHILDREN NODE TREE)))))
                (N OLD NEW STATEMENT))
```

```
          (LISTP (CDR (ASSOC (CONS (CAR (PARENT-REC 'TREE NODE
                                                     TREE))
                                  NODE) NEW))))
((INSTRUCTIONS PROMOTE
  (CLAIM (AND (NUMBERP NODE)
              (NUMBERP (PARENT NODE TREE)))
         0)
  (DIVE 1)
  (DIVE 1)
  (DIVE 1)
  (DIVE 1)
  (= (PARENT NODE TREE))
  TOP
  (BASH T
        (DISABLE TREE-PRG MIN ZEROP PARENT CHILDREN NODES))
  PROMOTE
  (CLAIM (EQUAL (PARENT NODE (CONS X Z)) X)
         0)
  (DIVE 1)
  (DIVE 1)
  (DIVE 1)
  (DIVE 1)
  = UP UP
  (REWRITE SEND-FIND-GENERAL
           (($OLD OLD)
            ($CHANNELS (RFP X (CHILDREN X (CONS X Z))))))
  TOP S
  (REWRITE MEMBER-RFP)
  S
  (DIVE 2)
  (DIVE 2)
  = TOP
  (REWRITE PARENT-REC-CHILDREN-REC)
  (DROP 3 4 6 8 9 10 11 12 13 14 15)
  PROVE
  (DIVE 1)
  (DIVE 1)
  (REWRITE ABOUT-UC
     (($B OLD)
      ($EXCPT (CONS (CONS 'STATUS X)
                    (CONS (CONS 'FOUND-VALUE X)
                          (CONS (CONS 'OUTSTANDING X)
                                (RFP X (CHILDREN X (CONS X Z)))))))))
  TOP S-PROP
  (DROP 1 2 3 4 5 6 7 8 9 10 11 12 13)
  (PROVE (DISABLE PARENT CHILDREN))
  PROMOTE
  (CLAIM (EQUAL V NODE) 0)
  (CONTRADICT 9)
  (= V NODE)
  S
  (CLAIM (EQUAL V (PARENT NODE TREE)) 0)
  (CLAIM (MEMBER NODE (CHILDREN V TREE))
         0)
  (DIVE 1)
  (DIVE 1)
  (DIVE 1)
  (DIVE 1)
  = UP UP
  (REWRITE SEND-FIND-GENERAL
           (($OLD OLD)
            ($CHANNELS (RFP V (CHILDREN V TREE)))))
  TOP S
  (REWRITE MEMBER-RFP)
  S
  (DEMOTE 22)
  S
  (CONTRADICT 22)
  S
```

```
(REWRITE PARENT-REC-CHILDREN-REC)
(DIVE 1)
= TOP S
(DROP 3 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22)
PROVE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
    (CONS (CONS (PARENT V TREE) V)
          (CONS (CONS V (PARENT V TREE))
                (CONS (CONS 'STATUS V)
                      (CONS (CONS 'FOUND-VALUE V)
                            (CONS (CONS 'OUTSTANDING V)
                                  (RFP V (CHILDREN V TREE))))))))))
TOP S-PROP
(DROP 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18)
(PROVE (DISABLE CHILDREN PARENT))
PROMOTE
(CLAIM (EQUAL V NODE) 0)
(CONTRADICT 9)
(DIVE 1)
(DIVE 1)
= TOP S-PROP
(CLAIM (EQUAL V (PARENT NODE TREE)) 0)
(DIVE 1)
(DIVE 1)
(REWRITE SEND-FIND-GENERAL
         (($OLD OLD)
          ($CHANNELS (RFP V (CHILDREN V TREE)))))
TOP S
(DIVE 1)
(DIVE 1)
= TOP
(REWRITE MEMBER-RFP)
S
(DIVE 2)
(DIVE 2)
= TOP
(REWRITE PARENT-REC-CHILDREN-REC)
(DROP 3 4 6 8 9 10 11 12 13 14 15 16 17 18 19 20 21)
PROVE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
    (CONS (CONS (PARENT V TREE) V)
          (CONS (CONS V (PARENT V TREE))
                (CONS (CONS 'STATUS V)
                      (CONS (CONS 'FOUND-VALUE V)
                            (CONS (CONS 'OUTSTANDING V)
                                  (RFP V (CHILDREN V TREE))))))))))
TOP S-PROP
(DROP 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18)
(PROVE (DISABLE PARENT CHILDREN))
PROMOTE
(CLAIM (EQUAL NODE (PARENT V TREE)) 0)
(CLAIM (EQUAL V (PARENT NODE TREE)) 0)
(CONTRADICT 21)
(DIVE 1)
(DIVE 2)
(DIVE 1)
= TOP
(DIVE 1)
S
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S
```

```
(DEMOTE 1 2)
DROP PROVE
(REWRITE LISTP-PARENT-REC-EQUALS)
(DEMOTE 1 2 5 8)
DROP PROVE
(DEMOTE 1 2)
DROP PROVE
(DROP 3 4 6 8 9 10 11 12 13 14 15 16 17 18 19 21)
PROVE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
    (CONS (CONS (PARENT V TREE) V)
          (CONS (CONS V (PARENT V TREE))
                (CONS (CONS 'STATUS V)
                      (CONS (CONS 'FOUND-VALUE V)
                            (CONS (CONS 'OUTSTANDING V)
                                  (RFP V (CHILDREN V TREE))))))))))
TOP S-PROP
(DROP 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 18)
(PROVE (DISABLE PARENT CHILDREN))
(CLAIM (EQUAL V (PARENT NODE TREE)) 0)
(DIVE 1)
(DIVE 1)
(REWRITE SEND-FIND-GENERAL
         (($OLD OLD)
          ($CHANNELS (RFP V (CHILDREN V TREE)))))
TOP S
(REWRITE MEMBER-RFP)
S
(DIVE 1)
(DIVE 2)
= TOP S
(DIVE 2)
(DIVE 2)
= TOP
(REWRITE PARENT-REC-CHILDREN-REC)
(DEMOTE 1 2 5 7)
DROP PROMOTE
(DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT)
TOP PROVE PROVE
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
    (CONS (CONS (PARENT V TREE) V)
          (CONS (CONS V (PARENT V TREE))
                (CONS (CONS 'STATUS V)
                      (CONS (CONS 'FOUND-VALUE V)
                            (CONS (CONS 'OUTSTANDING V)
                                  (RFP V (CHILDREN V TREE))))))))))
TOP S-PROP
(DEMOTE 9 19 20 21)
DROP
(PROVE (DISABLE PARENT CHILDREN))
PROMOTE
(CLAIM (AND (NOT (EQUAL V NODE))
            (NOT (MEMBER NODE (CHILDREN V TREE)))
            (NOT (EQUAL (CONS V (PARENT V TREE))
                        (CONS (PARENT NODE TREE) NODE))))
       0)
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
```

```
  ($EXCPT
    (CONS (CONS (PARENT V TREE) V)
          (CONS (CONS V (PARENT V TREE))
                (CONS (CONS 'STATUS V)
                      (CONS (CONS 'FOUND-VALUE V)
                            (CONS (CONS 'OUTSTANDING V)
                                  (RFP V (CHILDREN V TREE))))))))))
TOP S-PROP
(DEMOTE 9 18 19 20 21)
DROP
(PROVE (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 19)
(DEMOTE 1 2 5 7 8 9 15 18)
DROP
(BASH T
      (DISABLE PARENT CHILDREN NODES))
PROMOTE
(DROP 6 7 8)
(DIVE 1)
(DIVE 1)
S UP
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S PROVE PROVE PROVE PROMOTE
(DEMOTE 6)
(DIVE 1)
(DIVE 1)
(DIVE 1)
X-DUMB TOP DROP
(PROVE (DISABLE CHILDREN))
PROMOTE
(CLAIM (AND (NOT (MEMBER NODE (CHILDREN V TREE)))
            (NOT (EQUAL (CONS V (PARENT V TREE))
                        (CONS (PARENT NODE TREE) NODE))))
       0)
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
    (CONS (CONS (PARENT V TREE) V)
          (CONS (CONS V (PARENT V TREE))
                (CONS (CONS 'STATUS V)
                      (CONS (CONS 'FOUND-VALUE V)
                            (CONS (CONS 'OUTSTANDING V)
                                  (RFP V (CHILDREN V TREE))))))))))
TOP S-PROP
(DEMOTE 9 18 19 20)
DROP
(PROVE (DISABLE PARENT CHILDREN))
(CONTRADICT 19)
(DEMOTE 1 2 5 7 8 15)
DROP PROMOTE SPLIT
(DEMOTE 6)
(DIVE 1)
(DIVE 1)
(DIVE 1)
X-DUMB TOP DROP
(PROVE (DISABLE CHILDREN))
(DROP 6)
(BASH T
      (DISABLE PARENT CHILDREN NODES))
PROMOTE
(DIVE 1)
S
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S PROVE PROVE PROVE PROMOTE
(CLAIM (AND (NOT (MEMBER NODE (CHILDREN V TREE)))
            (NOT (EQUAL (CONS V (PARENT V TREE))
                        (CONS (PARENT NODE TREE) NODE))))
```

```
       0)
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
    (CONS (CONS (PARENT V TREE) V)
          (CONS (CONS V (PARENT V TREE))
                (CONS (CONS 'STATUS V)
                      (CONS (CONS 'FOUND-VALUE V)
                            (CONS (CONS 'OUTSTANDING V)
                                  (RFP V (CHILDREN V TREE))))))))))
TOP S-PROP
(DEMOTE 9 18 19 20)
DROP
(PROVE (DISABLE PARENT CHILDREN))
(CONTRADICT 19)
(DEMOTE 1 2 5 7 8 9 15)
DROP PROMOTE SPLIT
(DEMOTE 7)
(DIVE 1)
(DIVE 1)
(DIVE 1)
X-DUMB TOP DROP
(PROVE (DISABLE CHILDREN))
(DROP 7)
(BASH T
      (DISABLE PARENT CHILDREN NODES))
PROMOTE
(DIVE 1)
S
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S PROVE PROVE PROVE PROMOTE
(CLAIM (AND (NOT (EQUAL (CONS W V)
                        (CONS (PARENT NODE TREE) NODE)))
            (NOT (EQUAL (CONS V (PARENT V TREE))
                        (CONS (PARENT NODE TREE) NODE))))
       0)
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V TREE))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DEMOTE 17 18 19)
DROP
(PROVE (DISABLE CHILDREN PARENT))
(CONTRADICT 18)
(DEMOTE 1 2 5 7 8 9)
DROP PROMOTE
(BASH T (DISABLE PARENT CHILDREN))
PROMOTE
(DIVE 1)
S
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S PROVE PROVE PROVE PROMOTE
(DIVE 1)
S
(REWRITE PARENT-IS-NOT-CHILD)
TOP S PROVE PROVE PROMOTE
(CLAIM (AND (NOT (EQUAL (CONS (PARENT NODE TREE) NODE)
                        (CONS W V)))
            (NOT (EQUAL (CONS (PARENT NODE TREE) NODE)
                        (CONS V (PARENT V TREE)))))
       0)
(DIVE 1)
```

```
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V TREE))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DEMOTE 17 18 19)
DROP
(PROVE (DISABLE PARENT))
(CONTRADICT 18)
(DEMOTE 1 2 5 7 8 9)
DROP
(BASH T)
PROMOTE
(DIVE 1)
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S X
(BASH T)
PROVE PROMOTE
(DIVE 1)
(REWRITE PARENT-IS-NOT-CHILD)
TOP S X PROVE PROMOTE
(CLAIM (AND (NOT (EQUAL (CONS W V)
                        (CONS (PARENT NODE TREE) NODE)))
            (NOT (EQUAL (CONS V (PARENT V TREE))
                        (CONS (PARENT NODE TREE) NODE))))
       0)
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V TREE))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DEMOTE 17 18 19)
DROP
(PROVE (DISABLE PARENT))
(CONTRADICT 18)
(DEMOTE 1 2 5 7 8 9)
DROP
(BASH T
      (DISABLE PARENT CHILDREN NODES))
PROMOTE
(DIVE 1)
S
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S
(DROP 4 5 6)
PROVE PROVE PROVE
(BASH T)
PROMOTE
(DIVE 1)
(REWRITE PARENT-IS-NOT-CHILD)
TOP S
(DROP 4)
PROVE PROVE PROMOTE
(CLAIM (AND (NOT (EQUAL (CONS W V)
                        (CONS (PARENT NODE TREE) NODE)))
            (NOT (EQUAL (CONS V (PARENT V TREE))
                        (CONS (PARENT NODE TREE) NODE))))
       0)
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
```

```
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V TREE))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DEMOTE 17 18 19)
DROP
(PROVE (DISABLE PARENT))
(CONTRADICT 18)
(DEMOTE 1 2 5 7 8 9)
DROP
(BASH T
      (DISABLE PARENT CHILDREN NODES))
PROMOTE
(DIVE 1)
S
(REWRITE PARENT-OF-PARENT-NOT-NODE)
TOP S
(DROP 4 5 6)
PROVE PROVE PROVE
(BASH T)
PROMOTE
(DIVE 1)
(REWRITE PARENT-IS-NOT-CHILD)
TOP S
(DROP 4)
PROVE PROVE PROMOTE
(CLAIM (NOT (EQUAL NODE X)) 0)
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W X)
                        (CONS 'OUTSTANDING X)
                        (CONS 'FOUND-VALUE X)))))
TOP S-PROP
(DEMOTE 14 15)
DROP
(PROVE (DISABLE PARENT))
(CONTRADICT 15)
(DEMOTE 1 2 5 7)
DROP PROVE
(CONTRADICT 7)
(DROP 2 5 6 7)
PROVE)))

(PROVE-LEMMA RECEIVE-REPORT-DECREASES-TOU (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (LISTP (CHANNEL (CONS CHILD NODE) OLD))
                           (MEMBER NODE (CDR (NODES TREE)))
                           (MEMBER CHILD (CHILDREN NODE TREE))
                           (N OLD NEW
                              (LIST 'RECEIVE-REPORT NODE
                                    (CONS CHILD NODE)
                                    (CONS NODE (PARENT NODE TREE))))
                           (INV TREE OLD))
                      (TOU OLD NEW NODE TREE))
    ((INSTRUCTIONS PROMOTE (CLAIM (EQUAL (STATUS NODE OLD) 'STARTED) 0)
         (CLAIM (EQUAL (STATUS NODE NEW) 'STARTED) 0)
         (CLAIM (NOT (ZEROP (OUTSTANDING NODE OLD))) 0) (DROP 1 3 4 6)
         (PROVE (DISABLE CHANGED MIN)) (CONTRADICT 9) S (DIVE 1)
         (DIVE 1)
         (REWRITE NO-IMPLIES-INSTANCE-OF-NO
                  (($NODES (NODES TREE)) ($TREE TREE)))
         TOP
         (USE-LEMMA NUMBER-NOT-REPORTED-0-IMPLIES
             ((CHILDREN (CHILDREN NODE TREE)) (PARENT NODE)
              (NODE CHILD) (STATE OLD)))
         (DIVE 3) (DIVE 1)
```

```
          (REWRITE NO-IMPLIES-INSTANCE-OF-NO
                   (($NODES (NODES TREE)) ($TREE TREE)))
          TOP (DROP 1 3 5 6 7 8 9) (DIVE 1) (DIVE 1) (DIVE 1)
          (= (CHILDREN NODE TREE)) TOP (PROVE (DISABLE CHILDREN))
          (DEMOTE 6) S (DROP 2 4 5 6 7 8 9 10) PROVE (DEMOTE 6) S
          (DROP 2 4 5 6 7 8 9) PROVE (CONTRADICT 8) (DROP 6 8) (DIVE 1)
          S (DIVE 1)
          (REWRITE ABOUT-UC
                   (($B OLD)
                    ($EXCPT (LIST (CONS CHILD NODE)
                                  (CONS NODE (PARENT NODE TREE))
                                  (CONS 'OUTSTANDING NODE)
                                  (CONS 'FOUND-VALUE NODE)))))
          TOP (DEMOTE 6) S (DROP 1 3 4 6) (PROVE (DISABLE ZEROP MIN))
          (DROP 2 5 6) PROVE (CONTRADICT 7) (DEMOTE 6) (DIVE 1) X-DUMB
          TOP PROMOTE (DROP 5 6 9)
          (USE-LEMMA UL-IMPLIES-INSTANCE-OF-UL
              ((UPLINKS (UP-LINKS (CDR (NODES TREE)) TREE))
               (UPLINK (CONS CHILD NODE)) (STATE OLD)))
          (USE-LEMMA DL-IMPLIES-INSTANCE-OF-DL
              ((DOWN-LINKS (DOWN-LINKS (NODES TREE) TREE)) (STATE OLD)
               (DOWN-LINK (CONS NODE CHILD))))
          (DEMOTE 8) (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP (DEMOTE 7)
          (DIVE 1) (DIVE 1) S-PROP (= * T 0) TOP (DROP 1 3 4 5 6) PROVE
          S (REWRITE MEMBER-UP-LINKS) (DIVE 2) (DIVE 1) (DIVE 2) S
          (DIVE 1) (REWRITE PARENT-OF-CHILD (($PARENT NODE))) TOP
          (DROP 5 6) PROVE (DEMOTE 1) S (DEMOTE 1) S (DEMOTE 4) S S
          (REWRITE MEMBER-DOWN-LINKS) (DROP 2 5 6 7) PROVE)))

(PROVE-LEMMA OTHERS-PRESERVE-UP-TO-NODE-FULL (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (LISTP (CHANNEL (CONS CHILD NODE) OLD))
                 (MEMBER NODE (CDR (NODES TREE)))
                 (MEMBER CHILD (CHILDREN NODE TREE))
                 (NOT (EQUAL STATEMENT
                             (LIST 'RECEIVE-REPORT NODE
                                   (CONS CHILD NODE)
                                   (CONS NODE (PARENT NODE TREE)))))
                 (MEMBER STATEMENT (TREE-PRG TREE))
                 (N OLD NEW STATEMENT))
            (LISTP (CDR (ASSOC (CONS CHILD NODE) NEW))))
((INSTRUCTIONS PROMOTE
  (CLAIM (MEMBER CHILD (CDR (NODES TREE)))
         0)
  (CLAIM (EQUAL NODE (PARENT CHILD TREE))
         0)
  (DEMOTE 1 2 3 4 5 6 7 8)
  (= NODE (PARENT CHILD TREE))
  DROP PROMOTE
  (CLAIM (AND (NUMBERP CHILD)
              (NUMBERP (PARENT CHILD TREE))
              (NOT (EQUAL CHILD (CAR TREE)))
              (NOT (EQUAL (PARENT CHILD TREE)
                          (CAR TREE)))
              (NUMBERP (PARENT (PARENT CHILD TREE) TREE)))
         0)
  (BASH T
        (DISABLE TREE-PRG MIN ZEROP PARENT CHILDREN NODES))
  PROMOTE
  (CLAIM (EQUAL V CHILD) 0)
  (DEMOTE 6 17 22)
  DROP PROVE
  (DIVE 1)
  (DIVE 1)
  (REWRITE ABOUT-UC
   (($B OLD)
    ($EXCPT
     (CONS
        (CONS (PARENT V (CONS X Z)) V)
```

```
      (CONS (CONS V (PARENT V (CONS X Z)))
            (CONS (CONS 'STATUS V)
                  (CONS (CONS 'FOUND-VALUE V)
                        (CONS (CONS 'OUTSTANDING V)
                              (RFP V
                                   (CHILDREN V (CONS X Z))))))))))
TOP S-PROP
(DROP 6 10 11 12 13 14 15 16 17 18)
(BASH T)
PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
 (($B OLD)
  ($EXCPT
   (CONS
      (CONS (PARENT V (CONS X Z)) V)
      (CONS (CONS V (PARENT V (CONS X Z)))
            (CONS (CONS 'STATUS V)
                  (CONS (CONS 'FOUND-VALUE V)
                        (CONS (CONS 'OUTSTANDING V)
                              (RFP V
                                   (CHILDREN V (CONS X Z))))))))))
TOP S-PROP
(DROP 6 10 11 12 13 14 15 16 17)
PROVE PROMOTE
(CLAIM (EQUAL CHILD V) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(PROVE (DISABLE PARENT CHILDREN NODES))
PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(CLAIM (AND (NUMBERP X) (NUMBERP CHILD))
       0)
(BASH T
      (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 15)
(DROP 8 9 10 11 12 13 14 15)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
```

```
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(CLAIM (NUMBERP CHILD) 0)
(DROP 6 12 13 14 15 16 17 18)
(PROVE (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 23)
(DROP 6 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE CHILDREN NODES PARENT))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(CLAIM (NUMBERP CHILD) 0)
(PROVE (DISABLE CHILDREN PARENT NODES))
(CONTRADICT 15)
(DROP 8 9 10 11 12 13 14 15)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(CLAIM (NUMBERP CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 15)
(DROP 8 9 10 11 12 13 14 15)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(CLAIM (NUMBERP CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 15)
(DROP 8 9 10 11 12 13 14 15)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
```

```
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(CLAIM (NUMBERP CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 15)
(DROP 8 9 10 11 12 13 14 15)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(CLAIM (NUMBERP CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 15)
(DROP 8 9 10 11 12 13 14 15)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(CLAIM (NUMBERP CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 15)
(DROP 8 9 10 11 12 13 14 15)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
TOP S-PROP
(DROP 6 12 13 14 15 16 17 18)
(CLAIM (NUMBERP CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(CONTRADICT 15)
(DROP 8 9 10 11 12 13 14 15)
PROVE PROMOTE
(CLAIM (EQUAL V CHILD) 0)
(PROVE (DISABLE PARENT CHILDREN NODES))
(DIVE 1)
(DIVE 1)
(REWRITE ABOUT-UC
         (($B OLD)
          ($EXCPT (LIST (CONS W V)
                        (CONS V (PARENT V (CONS X Z)))
                        (CONS 'OUTSTANDING V)
                        (CONS 'FOUND-VALUE V)))))
```

```
  TOP S-PROP
  (CLAIM (NUMBERP CHILD) 0)
  (DROP 6 12 13 14 15 16 17 18)
  (PROVE (DISABLE PARENT CHILDREN NODES))
  (CONTRADICT 23)
  (DROP 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23)
  PROVE PROMOTE
  (CLAIM (EQUAL V CHILD) 0)
  (PROVE (DISABLE PARENT CHILDREN NODES))
  (DIVE 1)
  (DIVE 1)
  (REWRITE ABOUT-UC
           (($B OLD)
            ($EXCPT (LIST (CONS W V)
                          (CONS V (PARENT V (CONS X Z)))
                          (CONS 'OUTSTANDING V)
                          (CONS 'FOUND-VALUE V)))))
  TOP S-PROP
  (DROP 6 12 13 14 15 16 17 18)
  (CLAIM (NUMBERP CHILD) 0)
  (PROVE (DISABLE PARENT CHILDREN NODES))
  (CONTRADICT 15)
  (DROP 8 9 10 11 12 13 14 15)
  PROVE
  (CONTRADICT 9)
  (DROP 2 5 6 7 9)
  PROVE
  (CONTRADICT 9)
  (DROP 2 5 6 7 9)
  PROVE
  (CONTRADICT 8)
  (DROP 2 5 6 7 8)
  PROVE)))


(DISABLE TOTAL-OUTSTANDING-DECREASES)
(DISABLE TOU)
(DISABLE TOTAL-OUTSTANDING-DECREASES-SUBLIST)
(DISABLE STATUS-ROOT-BECOMES-STARTED-OR-UNCHANGED)
(DISABLE NOT-TOTAL-OUTSTANDING-0-IMPLIES-FULL-CHANNEL)
(DISABLE NOT-ALL-EMPTY-IMPLIES-FULL-CHANNEL-FULL)
(DISABLE FULL-CHANNEL)
(DISABLE ALL-EMPTY-ROOT-STARTED-IMPLIES-TOTAL-OUTSTANDING-0)
(DISABLE ALL-DONE-IMPLIES-TOTAL-OUTSTANDING-0)
(DISABLE ALL-NODES-ARE-DONE)
(DISABLE CHILDREN-ARE-SUFFIX-OF-SUBLIST-GENERALIZED)
(DISABLE CHILD-OF-NODE-IN-SUFFIX-IS-IN-SUFFIX)
(DISABLE SUBLIST-ULNKS)
(DISABLE ALL-EMPTY-IMPLIES-ALL-EMPTY-SUBLIST)
(DISABLE ALL-DONE-AND-ALL-EMPTY-IMPLIES-NUMBER-NOT-REPORTED-0)
(DISABLE ULNKS)
(DISABLE ALL-DONE-IMPLIES-ALL-DONE-SUBLIST)
(DISABLE ALL-DONE-IMPLIES-DONE)
(DISABLE ALL-DONE)
(DISABLE LATER-POSITIONS-ARE-IN-SUFFIX)
(DISABLE SETP-LIST-SETP-SUFFIX)
(DISABLE CHILDS-POSITION-INCREASES)
(DISABLE MEMBER-SUFFIX-MEMBER-LIST)
(DISABLE SUFFIX-IMPLIES-SUFFIX-CDR)
(DISABLE SUFFIX)
(DISABLE DL-AND-ALL-EMPTY-IMPLIES-ROOT-DEFINES-STATUS)
(DISABLE PARENT-TO-ROOT-INDUCTION)
(DISABLE PARENTS-POSITION-DECREASES)
(DISABLE POSITION-APPEND)
(DISABLE TOTAL-OUTSTANDING-NON-INCREASING)
(DISABLE TOTAL-OUTSTANDING-NON-INCREASING-SUBLIST)
(DISABLE OUTSTANDING-NON-INCREASING)
(DISABLE DL-UL-NO-PRESERVES-NO-SUBLIST)
(DISABLE DL-UL-NO-PRESERVES-INSTANCE-OF-NO)
```

```
(DISABLE RECEIVE-REPORT-PRESERVES-INSTANCE-OF-NO)
(DISABLE CHILD-MEMBER-CDR-NODES)
(DISABLE RECEIVE-REPORT-PRESERVES-NO-FOR-PARENT)
(DISABLE RECEIVE-REPORT-PRESERVES-NO-FOR-NODE)
(DISABLE RECEIVE-REPORT-PRESERVES-NO-FOR-REST-OF-TREE)
(DISABLE RECEIVE-FIND-PRESERVES-INSTANCE-OF-NO)
(DISABLE DL-DOWN-LINKS-IMPLIES-DL-RFP)
(DISABLE DOWN-LINKS-1-RFP)
(DISABLE DL-OF-APPEND)
(DISABLE RECEIVE-FIND-PRESERVES-NO-FOR-PARENT-OF-NODE)
(DISABLE RECEIVE-FIND-PRESERVES-NO-FOR-NODE)
(DISABLE RECEIVE-FIND-PRESERVES-NO-FOR-REST-OF-TREE)
(DISABLE ROOT-RECEIVE-REPORT-PRESERVES-INSTANCE-OF-NO)
(DISABLE SETP-NODES-SETP-CHILDREN)
(DISABLE SETP-NODES-IMPLIES-SETP-ROOTS)
(DISABLE NUMBER-NOT-REPORTED-OF-ROOT)
(DISABLE NUMBER-NOT-REPORTED-OF-NON-ROOT)
(DISABLE MIN-OF-REPORTED-OF-NON-ROOT)
(DISABLE UPDATE-MIN-OF-REPORTED)
(DISABLE MIN-OF-REPORTED-OF-MIN)
(DISABLE MIN-COMMUTATIVE-1)
(DISABLE MIN-ASSOCIATIVE)
(DISABLE MIN-COMMUTATIVE)
(DISABLE START-PRESERVES-INSTANCE-OF-NO)
(DISABLE LENGTH-RFP)
(DISABLE START-PRESERVES-NO-FOR-REST-OF-TREE)
(DISABLE UNCHANGED-PRESERVES-NO)
(DISABLE START-PRESERVES-NO-FOR-PARENT)
(DISABLE PARENT-NOT-STARTED-IMPLIES-ALL-EMPTY-AND-NOT-STARTED)
(DISABLE DL-UL-NO-PRESERVES-UL)
(DISABLE DL-UL-NO-PRESERVES-UL-SUBLIST)
(DISABLE DL-UL-NO-PRESERVES-INSTANCE-OF-UL)
(DISABLE ZERO-NOT-REPORTED-IMPLIES-CHILDREN-REPORTED)
(DISABLE MEMBER-UP-LINKS)
(DISABLE DL-PRESERVES-DL)
(DISABLE DL-PRESERVES-SUBLIST)
(DISABLE DL-PRESERVES-INSTANCE-OF-DL)
(DISABLE ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-CHILDREN)
(DISABLE ALL-NUMBERPS-FOREST-IMPLIES-ALL-NUMBERPS-ROOTS)
(DISABLE PARENT-NOT-LITATOM)
(DISABLE ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-CAR-PARENT)
(DISABLE ALL-NUMBERPS-NODES-IMPLIES-ALL-NUMBERPS-PARENT)
(DISABLE ALL-NUMBERPS-APPEND)
(DISABLE SEND-FIND-GENERAL)
(DISABLE ASSOC-EQUAL-CONS)
(DISABLE SEND-FIND-IMPLIES)
(DISABLE MEMBER-RFP)
(DISABLE PARENT-OF-PARENT-NOT-NODE)
(DISABLE PARENT-NOT-GRANDCHILD)
(DISABLE PARENT-NOT-CHILD)
(DISABLE MEMBER-DOWN-LINKS)
(DISABLE MEMBER-DOWN-LINKS-1)
(DISABLE UL-IMPLIES-INSTANCE-OF-UL-NOT-EMPTY-UPLINK)
(DISABLE NO-IMPLIES-INSTANCE-OF-NO)
(DISABLE UL-IMPLIES-INSTANCE-OF-UL)
(DISABLE DL-IMPLIES-INSTANCE-OF-DL)
(DISABLE INV-IMPLIES-AUGMENTED-CORRECTNESS-CONDITION)
(DISABLE INITIAL-CONDITIONS-IMPLY-INVARIANT)
(DISABLE ALL-EMPTY-NOT-STARTED-IMPLIES-DL)
(DISABLE INV)
(DISABLE DL)
(DISABLE NODE-VALUES-CONSTANT-INVARIANT)
(DISABLE NODE-VALUES-CONSTANT-UNLESS-SUFFICIENT)
(DISABLE LISTP-TREE-PRG)
(DISABLE ROOT-RECEIVE-REPORT-PRG-IS-TOTAL)
(DISABLE START-PRG-IS-TOTAL)
(DISABLE RECEIVE-REPORT-PRG-IS-TOTAL)
(DISABLE RECEIVE-FIND-PRG-IS-TOTAL)
(DISABLE ROOT-RECEIVE-REPORT-FUNC-IMPLEMENTS-ROOT-RECEIVE-REPORT)
```

```
(DISABLE ROOT-RECEIVE-REPORT-FUNC)
(DISABLE START-FUNC-IMPLEMENTS-START)
(DISABLE START-FUNC)
(DISABLE RECEIVE-REPORT-FUNC-IMPLEMENTS-RECEIVE-REPORT)
(DISABLE RECEIVE-REPORT-FUNC)
(DISABLE RECEIVE-FIND-FUNC-IMPLEMENTS-RECEIVE-FIND)
(DISABLE UC-OF-SEND-FIND-FUNC)
(DISABLE TO-NODE-NOT-IN-RFP)
(DISABLE PARENT-NOT-IN-RFP)
(DISABLE ABOUT-RFP-NUMBERP)
(DISABLE ABOUT-RFP)
(DISABLE ASSOC-OF-SEND-FIND-FUNC)
(DISABLE SEND-FIND-OF-UPDATE-ASSOC)
(DISABLE CHILDREN-ARE-NOT-LITATOMS-MEMBER)
(DISABLE CHILDREN-ARE-NOT-LITATOMS)
(DISABLE PARENT-IS-NOT-A-LITATOM)
(DISABLE NODES-ARE-NOT-LITATOMS)
(DISABLE SEND-FIND-FUNC-IMPLEMENTS-SEND-FIND)
(DISABLE RECEIVE-FIND-FUNC)
(DISABLE SEND-FIND-FUNC)
(DISABLE NO-AT-TERMINATION)
(DISABLE FOUND-VALUE-MIN-VALUE-GENERALIZED)
(DISABLE MIN-OF-TWO-NODES-VALUES)
(DISABLE PROPER-TREE-TREE-IMPLIES-NODES-EXISTS)
(DISABLE NUMBER-NOT-REPORTED-0-IMPLIES)
(DISABLE TOTAL-OUTSTANDING-0-IMPLIES)
(DISABLE NO-IMPLIES)
(DISABLE FOUND-VALUE-NODE-VALUE-APPEND)
(DISABLE NATI)
(DISABLE FOUND-VALUE-NODE-VALUE)
(DISABLE DOWN-LINKS-IS-SUBLISTP)
(DISABLE CHILDREN-OF-NON-NODE)
(DISABLE SUBLISTP-DOWN-LINKS-1)
(DISABLE SUBLISTP-NOT-STARTED)
(DISABLE NODES-IN-DOWN-LINKS-IN-NODES)
(DISABLE NODES-IN-CHANNELS-APPEND)
(DISABLE NODES-IN-DOWN-LINKS-1-IN-NODES)
(DISABLE NOT-STARTED-IMPLIES-NO)
(DISABLE NODES-IN-CHANNELS)
(DISABLE ALL-EMPTY-IMPLIES-UL)
(DISABLE ALL-EMPTY-APPEND)
(DISABLE NOT-STARTED-IMPLIES-NOT-STARTED)
(DISABLE ALL-EMPTY-IMPLIES-EMPTY)
(DISABLE CORRECT)
(DISABLE MIN-NODE-VALUE)
(DISABLE ALL-EMPTY)
(DISABLE ALL-CHANNELS)
(DISABLE NOT-STARTED)
(DISABLE UP-LINKS)
(DISABLE DOWN-LINKS)
(DISABLE DOWN-LINKS-1)
(DISABLE NO)
(DISABLE MIN-OF-REPORTED)
(DISABLE NUMBER-NOT-REPORTED)
(DISABLE REPORTED)
(DISABLE UL)
(DISABLE DONE)
(DISABLE TOTAL-OUTSTANDING)
(DISABLE TREEP)
(DISABLE MEMBER-TREE-PRG)
(DISABLE EQUAL-IF)
(DISABLE TREE-PRG)
(DISABLE MEMBER-ROOT-RECEIVE-REPORT-PRG)
(DISABLE ROOT-RECEIVE-REPORT-PRG)
(DISABLE MEMBER-RRRP)
(DISABLE RRRP)
(DISABLE MEMBER-START-PRG)
(DISABLE START-PRG)
(DISABLE MEMBER-RECEIVE-REPORT-PRG)
```

```
(DISABLE RECEIVE-REPORT-PRG)
(DISABLE MEMBER-RRP)
(DISABLE RRP)
(DISABLE MEMBER-RECEIVE-FIND-PRG)
(DISABLE RECEIVE-FIND-PRG)
(DISABLE RFP)
(DISABLE ROOT-RECEIVE-REPORT)
(DISABLE START)
(DISABLE RECEIVE-REPORT)
(DISABLE MIN)
(DISABLE RECEIVE-FIND)
(DISABLE SEND-FIND)
(DISABLE NODE-VALUE)
(DISABLE OUTSTANDING)
(DISABLE FOUND-VALUE)
(DISABLE STATUS)
(DISABLE RECEIVE)
(DISABLE SEND)
(DISABLE HEAD)
(DISABLE EMPTY)
(DISABLE CHANNEL)
(DISABLE VALUE)
(DISABLE PARENT-NOT-IN-CHILDREN)
(DISABLE PARENT-IS-NOT-CHILD)
(DISABLE LISTP-PARENT-REC-EQUALS)
(DISABLE PARENT-IS-NOT-ITSELF)
(DISABLE PARENT-IS-NOT-ITSELF-GENERALIZED)
(DISABLE NODE-HAS-PARENT)
(DISABLE CHILDREN-OF-SETP-TREE)
(DISABLE MEMBER-SUBTREE-MEMBER-TREE)
(DISABLE NO-CHILDREN-IN-REST-OF-TREE)
(DISABLE NO-CHILDREN-IN-REST-OF-FOREST)
(DISABLE NOT-MEMBER-NO-CHILDREN)
(DISABLE NOT-MEMBER-SUBTREES)
(DISABLE PROPER-TREE-NEXT-LEVEL-OF-PROPER-TREE)
(DISABLE PROPER-TREE-OF-APPEND)
(DISABLE NEXT-LEVEL-OF-SUBTREES-IN-COMPLETE-SUBTREES)
(DISABLE NEXT-LEVEL-IN-SUBTREES-FOREST)
(DISABLE SUBTREES-OF-SUBTREES-IN-COMPLETE-SUBTREES)
(DISABLE SUBTREES-OF-SUBTREE-IN-COMPLETE-SUBTREES)
(DISABLE NEXT-LEVEL-OF-TREE-IN-SUBTREES)
(DISABLE NEXT-LEVEL-REDUCES-COUNT)
(DISABLE NODES-REC-FOREST-APPEND)
(DISABLE NEXT-LEVEL)
(DISABLE SUBTREEP-SUBTREES)
(DISABLE SUBTREES)
(DISABLE SUBTREEP)
(DISABLE SUBLISTP-CHILDREN)
(DISABLE SUBLISTP-CHILDREN-GENERALIZED)
(DISABLE NODE-THAT-HAS-PARENT-IS-IN-TREE)
(DISABLE NODE-THAT-HAS-CHILD-IS-IN-TREE)
(DISABLE MEMBER-PARENT-MEMBER-TREE)
(DISABLE PARENT-OF-CHILD)
(DISABLE MEMBER-PARENT-PARENT)
(DISABLE MEMBER-CHILD-TREE)
(DISABLE NOT-MEMBER-NO-PARENT)
(DISABLE PLISTP-ROOTS)
(DISABLE PLISTP-PARENT-REC)
(DISABLE PLISTP-CHILDREN-REC)
(DISABLE MEMBER-ROOTS-MEMBER-FOREST)
(DISABLE PARENT-REC-CHILDREN-REC)
(DISABLE NOT-FLAG-TREE)
(DISABLE CANONICALIZE-CHILDREN-REC-FLAG)
(DISABLE CANONICALIZE-PARENT-REC-FLAG)
(DISABLE CANONICALIZE-PROPER-TREE-FLAG)
(DISABLE CANONICALIZE-NODES-REC-FLAG)
(DISABLE PROPER-TREE)
(DISABLE PARENT)
(DISABLE PARENT-REC)
```

```
(DISABLE CHILDREN)
(DISABLE CHILDREN-REC)
(DISABLE ROOTS)
(DISABLE NODES)
(DISABLE NODES-REC)
(DISABLE SUBLISTP-IN-CONS)
(DISABLE SUBLISTP-IN-APPEND)
(DISABLE SUBLISTP-REFLEXIVE)
(DISABLE SUBLISTP-EASY)
(DISABLE SEI)
(DISABLE SUBLISTP-NORMALIZE)
(DISABLE SUBLISTP-OF-SUBLISTP-IS-SUBLISTP)
(DISABLE MEMBER-OF-SUBLISTP-IS-MEMBER)
(DISABLE SUBLISTP-APPEND)
(DISABLE SUBLISTP)
(DISABLE SETP-MEMBER-2)
(DISABLE SETP-MEMBER-1)
(DISABLE SETP-APPEND-CANONICALIZE)
(DISABLE SETP-APPEND-NOT-LISTP)
(DISABLE SETP-APPEND-CONS)
(DISABLE SETP-MEMBER)
(DISABLE SETP-APPEND)
(DISABLE SETP)
(DISABLE ALL-NUMBERPS-IMPLIES)
(DISABLE ALL-NUMBERPS)
(DISABLE NOT-LESSP-COUNT-APPEND)
(DISABLE APPEND-PLISTP-NIL)
(DISABLE PLISTP-APPEND-PLISTP)
(DISABLE PLISTP)
(DISABLE LENGTH-APPEND)
(DISABLE LISTP-APPEND)
(DISABLE CAR-APPEND)

(DISABLE N)

(PROVE-LEMMA MEMBER-CDR-NODES-MEMBER-NODES (REWRITE)
             (IMPLIES (AND (MEMBER NODE (CDR (NODES TREE)))
                           (TREEP TREE))
                      (MEMBER NODE (NODES TREE)))
             ((ENABLE TREEP PROPER-TREE)))

(PROVE-LEMMA TOTAL-OUTSTANDING-DECREASES-EXPANDED (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (INV TREE OLD)
                (N OLD NEW STATEMENT)
                (MEMBER STATEMENT (TREE-PRG TREE))
                (MEMBER NODE (NODES TREE))
                (TOU OLD NEW NODE TREE))
           (AND (EQUAL (LESSP (TOTAL-OUTSTANDING (NODES TREE)
                                                 TREE NEW)
                              (TOTAL-OUTSTANDING (NODES TREE)
                                                 TREE OLD))
                       T)
                (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE NEW)
                       (TOTAL-OUTSTANDING (NODES TREE) TREE OLD))))
   ((ENABLE INV)
    (USE (TOTAL-OUTSTANDING-DECREASES))))

(PROVE-LEMMA TOTAL-OUTSTANDING-DECREASES-EXPANDED-COUNT (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (INV TREE OLD)
                 (N OLD NEW STATEMENT)
                 (MEMBER STATEMENT (TREE-PRG TREE))
                 (MEMBER NODE (NODES TREE))
                 (TOU OLD NEW NODE TREE)
                 (EQUAL (TOTAL-OUTSTANDING (NODES TREE) TREE OLD)
                        (ADD1 COUNT)))
            (AND (EQUAL (LESSP (TOTAL-OUTSTANDING (NODES TREE)
                                                  TREE NEW)
```

```
                              (ADD1 COUNT))
                       T)
                (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE NEW)
                       (ADD1 COUNT)))))

(PROVE-LEMMA TOTAL-OUTSTANDING-NON-INCREASING-EXPANDED (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (MEMBER STATEMENT (TREE-PRG TREE))
                (N OLD NEW STATEMENT)
                (INV TREE OLD))
           (AND (EQUAL (LESSP (TOTAL-OUTSTANDING (NODES TREE)
                                                 TREE OLD)
                              (TOTAL-OUTSTANDING (NODES TREE)
                                                 TREE NEW))
                      F)
                (NOT (LESSP (TOTAL-OUTSTANDING (NODES TREE)
                                               TREE OLD)
                            (TOTAL-OUTSTANDING (NODES TREE)
                                               TREE NEW)))))
   ((ENABLE INV)
    (USE (TOTAL-OUTSTANDING-NON-INCREASING))))

(PROVE-LEMMA TOTAL-OUTSTANDING-NON-INCREASING-EXPANDED-COUNT (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (MEMBER STATEMENT (TREE-PRG TREE))
                (N OLD NEW STATEMENT)
                (INV TREE OLD)
                (EQUAL (TOTAL-OUTSTANDING (NODES TREE) TREE OLD)
                       (ADD1 COUNT)))
           (AND (EQUAL (LESSP (ADD1 COUNT)
                              (TOTAL-OUTSTANDING (NODES TREE)
                                                 TREE OLD))
                      F)
                (NOT (LESSP (ADD1 COUNT)
                            (TOTAL-OUTSTANDING (NODES TREE)
                                               TREE NEW))))))

(PROVE-LEMMA KEY-STATEMENTS-MEMBER-TREE-PRG (REWRITE)
             (AND (IMPLIES (TREEP TREE)
                           (MEMBER (LIST 'START (CAR TREE)
                                         (RFP (CAR TREE)
                                              (CHILDREN (CAR TREE)
                                                        TREE)))
                                   (TREE-PRG TREE)))
                  (IMPLIES (AND (TREEP TREE)
                                (MEMBER CHILD (CHILDREN (CAR TREE) TREE)))
                           (MEMBER (LIST 'ROOT-RECEIVE-REPORT (CAR TREE)
                                         (CONS CHILD (CAR TREE)))
                                   (TREE-PRG TREE)))
                  (IMPLIES (AND (TREEP TREE)
                                (MEMBER NODE (CDR (NODES TREE))))
                           (MEMBER (LIST 'RECEIVE-FIND NODE
                                         (CONS (PARENT NODE TREE) NODE)
                                         (CONS NODE (PARENT NODE TREE))
                                         (RFP NODE
                                              (CHILDREN NODE TREE)))
                                   (TREE-PRG TREE)))
                  (IMPLIES (AND (TREEP TREE)
                                (MEMBER NODE (CDR (NODES TREE)))
                                (MEMBER CHILD (CHILDREN NODE TREE)))
                           (MEMBER (LIST 'RECEIVE-REPORT NODE
                                         (CONS CHILD NODE)
                                         (CONS NODE (PARENT NODE TREE)))
                                   (TREE-PRG TREE))))
             ((ENABLE MEMBER-TREE-PRG)))

(PROVE-LEMMA DOWN-LINK-FULL-DECREASES-TOTAL-OUTSTANDING-ENSURES (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (MEMBER NODE (CDR (NODES TREE)))
```

```
                  (INV TREE OLD)
                  (LISTP (CHANNEL (CONS (PARENT NODE TREE) NODE) OLD))
                  (N OLD NEW
                     (LIST 'RECEIVE-FIND
                           NODE
                           (CONS (PARENT NODE TREE) NODE)
                           (CONS NODE (PARENT NODE TREE))
                           (RFP NODE (CHILDREN NODE TREE)))))
             (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE NEW)
                    (TOTAL-OUTSTANDING (NODES TREE) TREE OLD)))
     ((INSTRUCTIONS PROMOTE
          (REWRITE TOTAL-OUTSTANDING-DECREASES-EXPANDED
                   (($STATEMENT
                        (LIST 'RECEIVE-FIND NODE
                              (CONS (PARENT NODE TREE) NODE)
                              (CONS NODE (PARENT NODE TREE))
                              (RFP NODE (CHILDREN NODE TREE))))
                    ($NODE NODE)))
          PROVE PROVE PROVE)))

(PROVE-LEMMA DOWN-LINK-FULL-UNLESS (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (MEMBER NODE (CDR (NODES TREE)))
                 (LISTP (CHANNEL (CONS (PARENT NODE TREE) NODE) OLD))
                 (MEMBER STATEMENT (TREE-PRG TREE))
                 (NOT (EQUAL STATEMENT
                             (LIST 'RECEIVE-FIND
                                   NODE
                                   (CONS (PARENT NODE TREE) NODE)
                                   (CONS NODE (PARENT NODE TREE))
                                   (RFP NODE (CHILDREN NODE TREE)))))
                 (N OLD NEW STATEMENT))
            (LISTP (CHANNEL (CONS (PARENT NODE TREE) NODE) NEW)))
     ((INSTRUCTIONS PROMOTE (DIVE 1) (DIVE 1) (DIVE 1) X UP UP X X TOP
          PROVE)))

(PROVE-LEMMA DOWN-LINK-FULL-DECREASES-TOTAL-OUTSTANDING (REWRITE)
 (IMPLIES (AND (TREEP TREE)
               (MEMBER NODE (CDR (NODES TREE))))
          (LEADS-TO `(AND (INV (QUOTE ,TREE) STATE)
                          (AND (LISTP (CHANNEL
                                       (QUOTE ,(CONS (PARENT NODE
                                                             TREE)
                                                     NODE))
                                       STATE))
                               (EQUAL (TOTAL-OUTSTANDING
                                       (QUOTE ,(NODES TREE))
                                       (QUOTE ,TREE)
                                       STATE)
                                      (QUOTE ,(ADD1 COUNT)))))
                    `(LESSP (TOTAL-OUTSTANDING
                             (QUOTE ,(NODES TREE))
                             (QUOTE ,TREE)
                             STATE)
                            (QUOTE ,(ADD1 COUNT)))
                    (TREE-PRG TREE)))
 ((INSTRUCTIONS
   PROMOTE (REWRITE UNCONDITIONAL-FAIRNESS)
   (REWRITE HELP-PROVE-UNLESS)
   (CLAIM (EQUAL (EU (LIST 'AND
                           (CONS 'INV
                                 (CONS (LIST 'QUOTE TREE)
                                       '(STATE)))
                           (LIST 'AND
                                 (LIST 'LISTP
                                       (CONS 'CHANNEL
                                             (CONS
                                              (LIST 'QUOTE
                                                    (CONS (PARENT NODE
```

```
                                                                    TREE)
                                                            NODE))
                                                   '(STATE))))
                                    (LIST 'EQUAL
                                          (CONS 'TOTAL-OUTSTANDING
                                                (CONS
                                                 (LIST 'QUOTE (NODES TREE))
                                                 (CONS (LIST 'QUOTE TREE)
                                                       '(STATE))))
                                          (LIST 'QUOTE (ADD1 COUNT)))))
                        (TREE-PRG TREE)
                        (LIST 'LESSP
                              (CONS 'TOTAL-OUTSTANDING
                                    (CONS (LIST 'QUOTE (NODES TREE))
                                          (CONS (LIST 'QUOTE TREE)
                                                '(STATE))))
                              (LIST 'QUOTE (ADD1 COUNT))))
                    (LIST 'RECEIVE-FIND NODE
                          (CONS (PARENT NODE TREE) NODE)
                          (CONS NODE (PARENT NODE TREE))
                          (RFP NODE (CHILDREN NODE TREE))))
             0)
   PROVE PROVE
   (REWRITE HELP-PROVE-ENSURES
            (($STATEMENT
              (LIST 'RECEIVE-FIND NODE
                    (CONS (PARENT NODE TREE) NODE)
                    (CONS NODE (PARENT NODE TREE))
                    (RFP NODE (CHILDREN NODE TREE))))))
   PROVE PROVE)))

(PROVE-LEMMA MEMBER-CAR-TREE-NODES-TREE (REWRITE)
             (IMPLIES (TREEP TREE)
                      (MEMBER (CAR TREE) (NODES TREE)))
             ((ENABLE TREEP PROPER-TREE NODES NODES-REC)))


(PROVE-LEMMA ROOT-UP-LINK-FULL-DECREASES-TOTAL-OUTSTANDING-ENSURES
 (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (INV TREE OLD)
                           (MEMBER CHILD (CHILDREN (CAR TREE) TREE))
                           (LISTP (CHANNEL (CONS CHILD (CAR TREE)) OLD))
                           (N OLD NEW
                              (LIST 'ROOT-RECEIVE-REPORT (CAR TREE)
                                    (CONS CHILD (CAR TREE)))))
                      (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE NEW)
                             (TOTAL-OUTSTANDING (NODES TREE) TREE OLD)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE TOTAL-OUTSTANDING-DECREASES-EXPANDED
                  (($STATEMENT
                      (LIST 'ROOT-RECEIVE-REPORT (CAR TREE)
                            (CONS CHILD (CAR TREE))))
                   ($NODE (CAR TREE))))
         PROVE PROVE PROVE)))

(PROVE-LEMMA ROOT-UP-LINK-FULL-UNLESS (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (MEMBER CHILD (CHILDREN (CAR TREE) TREE))
                (LISTP (CHANNEL (CONS CHILD (CAR TREE)) OLD))
                (MEMBER STATEMENT (TREE-PRG TREE))
                (NOT (EQUAL STATEMENT
                            (LIST 'ROOT-RECEIVE-REPORT (CAR TREE)
                                  (CONS CHILD (CAR TREE)))))
                (N OLD NEW STATEMENT))
           (LISTP (CHANNEL (CONS CHILD (CAR TREE)) NEW)))
    ((INSTRUCTIONS PROMOTE (DIVE 1) X X TOP PROVE)))

(PROVE-LEMMA UP-LINK-FULL-DECREASES-TOTAL-OUTSTANDING-ENSURES (REWRITE)
```

```
             (IMPLIES (AND (TREEP TREE)
                           (INV TREE OLD)
                           (MEMBER NODE (CDR (NODES TREE)))
                           (MEMBER CHILD (CHILDREN NODE TREE))
                           (LISTP (CHANNEL (CONS CHILD NODE) OLD))
                           (N OLD NEW
                              (LIST 'RECEIVE-REPORT NODE
                                    (CONS CHILD NODE)
                                    (CONS NODE (PARENT NODE TREE)))))
                      (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE NEW)
                             (TOTAL-OUTSTANDING (NODES TREE) TREE OLD)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE TOTAL-OUTSTANDING-DECREASES-EXPANDED
                  (($STATEMENT
                       (LIST 'RECEIVE-REPORT NODE (CONS CHILD NODE)
                             (CONS NODE (PARENT NODE TREE))))))
         PROVE PROVE PROVE))

(PROVE-LEMMA UP-LINK-FULL-UNLESS (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (MEMBER NODE (CDR (NODES TREE)))
                (MEMBER CHILD (CHILDREN NODE TREE))
                (LISTP (CHANNEL (CONS CHILD NODE) OLD))
                (MEMBER STATEMENT (TREE-PRG TREE))
                (NOT (EQUAL STATEMENT
                            (LIST 'RECEIVE-REPORT NODE
                                  (CONS CHILD NODE)
                                  (CONS NODE (PARENT NODE TREE)))))
                (N OLD NEW STATEMENT))
           (LISTP (CHANNEL (CONS CHILD NODE) NEW)))
    ((INSTRUCTIONS PROMOTE (DIVE 1) X X TOP PROVE)))

(PROVE-LEMMA MEMBER-CDR-NODES-EQUALS (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (NOT (EQUAL NODE (CAR TREE))))
                      (EQUAL (MEMBER NODE (NODES TREE))
                             (MEMBER NODE (CDR (NODES TREE)))))
             ((ENABLE TREEP PROPER-TREE NODES NODES-REC)))


(PROVE-LEMMA UP-LINK-FULL-DECREASES-TOTAL-OUTSTANDING (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (MEMBER NODE (NODES TREE))
                (MEMBER CHILD (CHILDREN NODE TREE)))
           (LEADS-TO `(AND (INV (QUOTE ,TREE) STATE)
                           (AND (LISTP (CHANNEL
                                        (QUOTE ,(CONS CHILD NODE))
                                        STATE))
                                (EQUAL (TOTAL-OUTSTANDING
                                        (QUOTE ,(NODES TREE))
                                        (QUOTE ,TREE)
                                        STATE)
                                       (QUOTE ,(ADD1 COUNT)))))
                      `(LESSP (TOTAL-OUTSTANDING
                               (QUOTE ,(NODES TREE))
                               (QUOTE ,TREE)
                               STATE)
                              (QUOTE ,(ADD1 COUNT)))
                      (TREE-PRG TREE)))
    ((INSTRUCTIONS PROMOTE (CLAIM (EQUAL NODE (CAR TREE)) 0)
         (REWRITE UNCONDITIONAL-FAIRNESS) (REWRITE HELP-PROVE-UNLESS)
         (CLAIM (EQUAL (EU (LIST 'AND
                                 (CONS 'INV
                                       (CONS (LIST 'QUOTE TREE)
                                        '(STATE)))
                                 (LIST 'AND
                                       (LIST 'LISTP
                                        (CONS 'CHANNEL
                                         (CONS
```

```
                                    (LIST 'QUOTE
                                     (CONS CHILD NODE))
                                    '(STATE))))
                                 (LIST 'EQUAL
                                  (CONS 'TOTAL-OUTSTANDING
                                   (CONS
                                    (LIST 'QUOTE (NODES TREE))
                                    (CONS (LIST 'QUOTE TREE)
                                     '(STATE))))
                                  (LIST 'QUOTE (ADD1 COUNT)))))
                     (TREE-PRG TREE)
                     (LIST 'LESSP
                           (CONS 'TOTAL-OUTSTANDING
                                 (CONS (LIST 'QUOTE (NODES TREE))
                                  (CONS (LIST 'QUOTE TREE)
                                   '(STATE))))
                           (LIST 'QUOTE (ADD1 COUNT))))
                 (LIST 'ROOT-RECEIVE-REPORT (CAR TREE)
                       (CONS CHILD (CAR TREE))))
          0)
   PROVE PROVE
   (REWRITE HELP-PROVE-ENSURES
            (($STATEMENT
                 (LIST 'ROOT-RECEIVE-REPORT (CAR TREE)
                       (CONS CHILD (CAR TREE))))))
   PROVE PROVE (REWRITE UNCONDITIONAL-FAIRNESS)
   (REWRITE HELP-PROVE-UNLESS)
   (CLAIM (EQUAL (EU (LIST 'AND
                           (CONS 'INV
                                 (CONS (LIST 'QUOTE TREE)
                                  '(STATE)))
                           (LIST 'AND
                                 (LIST 'LISTP
                                  (CONS 'CHANNEL
                                   (CONS
                                    (LIST 'QUOTE
                                     (CONS CHILD NODE))
                                    '(STATE))))
                                 (LIST 'EQUAL
                                  (CONS 'TOTAL-OUTSTANDING
                                   (CONS
                                    (LIST 'QUOTE (NODES TREE))
                                    (CONS (LIST 'QUOTE TREE)
                                     '(STATE))))
                                  (LIST 'QUOTE (ADD1 COUNT)))))
                     (TREE-PRG TREE)
                     (LIST 'LESSP
                           (CONS 'TOTAL-OUTSTANDING
                                 (CONS (LIST 'QUOTE (NODES TREE))
                                  (CONS (LIST 'QUOTE TREE)
                                   '(STATE))))
                           (LIST 'QUOTE (ADD1 COUNT))))
                 (LIST 'RECEIVE-REPORT NODE (CONS CHILD NODE)
                       (CONS NODE (PARENT NODE TREE))))
          0)
   PROVE PROVE
   (REWRITE HELP-PROVE-ENSURES
            (($STATEMENT
                 (LIST 'RECEIVE-REPORT NODE (CONS CHILD NODE)
                       (CONS NODE (PARENT NODE TREE))))))
   PROVE PROVE)))

(PROVE-LEMMA NOT-STARTED-ROOT-DECREASES-TOTAL-OUTSTANDING-ENSURES
 (REWRITE)
     (IMPLIES (AND (TREEP TREE)
                   (INV TREE OLD)
                   (EQUAL (STATUS (CAR TREE) OLD) 'NOT-STARTED)
                   (N OLD NEW
                      (LIST 'START (CAR TREE)
```

```
                                        (RFP (CAR TREE)
                                             (CHILDREN (CAR TREE) TREE)))))
                         (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE NEW)
                                (TOTAL-OUTSTANDING (NODES TREE) TREE OLD)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE TOTAL-OUTSTANDING-DECREASES-EXPANDED
                  (($STATEMENT
                       (LIST 'START (CAR TREE)
                             (RFP (CAR TREE)
                                  (CHILDREN (CAR TREE) TREE))))
                   ($NODE (CAR TREE))))
         PROVE PROVE PROVE)))

(PROVE-LEMMA NOT-STARTED-ROOT-UNLESS (REWRITE)
  (IMPLIES (AND (TREEP TREE)
                (EQUAL (STATUS (CAR TREE) OLD) 'NOT-STARTED)
                (MEMBER STATEMENT (TREE-PRG TREE))
                (NOT (EQUAL STATEMENT
                            (LIST 'START (CAR TREE)
                                  (RFP (CAR TREE)
                                       (CHILDREN (CAR TREE)
                                                 TREE)))))
                (N OLD NEW STATEMENT))
           (EQUAL (STATUS (CAR TREE) NEW) 'NOT-STARTED))
             ((ENABLE STATUS VALUE)))

(PROVE-LEMMA NOT-STARTED-ROOT-DECREASES-TOTAL-OUTSTANDING (REWRITE)
   (IMPLIES (TREEP TREE)
            (LEADS-TO `(AND (INV (QUOTE ,TREE) STATE)
                            (AND (EQUAL (STATUS (QUOTE ,(CAR TREE))
                                                STATE)
                                        'NOT-STARTED)
                                 (EQUAL (TOTAL-OUTSTANDING
                                         (QUOTE ,(NODES TREE))
                                         (QUOTE ,TREE)
                                         STATE)
                                        (QUOTE ,(ADD1 COUNT)))))
                      `(LESSP (TOTAL-OUTSTANDING
                                 (QUOTE ,(NODES TREE))
                                 (QUOTE ,TREE)
                                 STATE)
                                (QUOTE ,(ADD1 COUNT)))
                      (TREE-PRG TREE)))
    ((INSTRUCTIONS PROMOTE (REWRITE UNCONDITIONAL-FAIRNESS)
         (REWRITE HELP-PROVE-UNLESS)
         (CLAIM (EQUAL (EU (LIST 'AND
                                 (CONS 'INV
                                       (CONS (LIST 'QUOTE TREE)
                                        '(STATE)))
                                 (LIST 'AND
                                       (CONS 'EQUAL
                                        (CONS
                                         (CONS 'STATUS
                                          (CONS
                                           (LIST 'QUOTE (CAR TREE))
                                           '(STATE)))
                                         '('NOT-STARTED)))
                                       (LIST 'EQUAL
                                        (CONS 'TOTAL-OUTSTANDING
                                         (CONS
                                          (LIST 'QUOTE (NODES TREE))
                                          (CONS (LIST 'QUOTE TREE)
                                           '(STATE))))
                                        (LIST 'QUOTE (ADD1 COUNT)))))
                           (TREE-PRG TREE)
                           (LIST 'LESSP
                                 (CONS 'TOTAL-OUTSTANDING
                                       (CONS (LIST 'QUOTE (NODES TREE))
                                        (CONS (LIST 'QUOTE TREE)
```

```
                                        '(STATE))))
                                (LIST 'QUOTE (ADD1 COUNT))))
                      (LIST 'START (CAR TREE)
                            (RFP (CAR TREE)
                                 (CHILDREN (CAR TREE) TREE))))
               0)
        PROVE PROVE
        (REWRITE HELP-PROVE-ENSURES
                 (($STATEMENT
                      (LIST 'START (CAR TREE)
                            (RFP (CAR TREE)
                                 (CHILDREN (CAR TREE) TREE))))))
        PROVE PROVE)))

(PROVE-LEMMA FULL-CHANNEL-NOT-F-IMPLIES (REWRITE)
             (IMPLIES (FULL-CHANNEL CHANNELS STATE)
                      (AND (MEMBER (FULL-CHANNEL CHANNELS STATE) CHANNELS)
                           (LISTP (CHANNEL (FULL-CHANNEL CHANNELS STATE)
                                           STATE))))
             ((ENABLE FULL-CHANNEL EMPTY CHANNEL VALUE)))

(PROVE-LEMMA TOTAL-OUTSTANDING-DECREASES-LEADS-TO (REWRITE)
   (IMPLIES (AND (TREEP TREE)
                 (INITIAL-CONDITION
                  `(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS TREE))
                                   STATE)
                        (NOT-STARTED (QUOTE ,(NODES TREE))
                                     STATE))
                  (TREE-PRG TREE)))
            (LEADS-TO `(EQUAL (TOTAL-OUTSTANDING
                               (QUOTE ,(NODES TREE))
                               (QUOTE ,TREE)
                               STATE)
                              (QUOTE ,(ADD1 COUNT)))
                      `(LESSP (TOTAL-OUTSTANDING
                               (QUOTE ,(NODES TREE))
                               (QUOTE ,TREE)
                               STATE)
                              (QUOTE ,(ADD1 COUNT)))
                      (TREE-PRG TREE)))
((INSTRUCTIONS PROMOTE
  (REWRITE LEADS-TO-STRENGTHEN-LEFT
   (($P
     (LIST 'OR
      (LIST 'AND
            (LIST 'INV
                  (LIST 'QUOTE TREE)
                  'STATE)
            (LIST 'AND
                  (LIST 'EQUAL
                        (LIST 'STATUS
                              (LIST 'QUOTE (CAR TREE))
                              'STATE)
                        ''NOT-STARTED)
                  (LIST 'EQUAL
                        (LIST 'TOTAL-OUTSTANDING
                              (LIST 'QUOTE (NODES TREE))
                              (LIST 'QUOTE TREE)
                              'STATE)
                        (LIST 'QUOTE (ADD1 COUNT)))))
      (LIST 'OR
          (LIST 'AND
                (LIST 'INV
                      (LIST 'QUOTE TREE)
                      'STATE)
                (LIST 'AND
                      (LIST 'FULL-CHANNEL
                            (LIST 'QUOTE
                                  (DOWN-LINKS (NODES TREE) TREE))
```

```
                         'STATE)
                   (LIST 'EQUAL
                         (LIST 'TOTAL-OUTSTANDING
                               (LIST 'QUOTE (NODES TREE))
                               (LIST 'QUOTE TREE)
                               'STATE)
                         (LIST 'QUOTE (ADD1 COUNT)))))
       (LIST 'AND
             (LIST 'INV
                   (LIST 'QUOTE TREE)
                   'STATE)
             (LIST 'AND
                   (LIST 'FULL-CHANNEL
                         (LIST 'QUOTE
                               (UP-LINKS (CDR (NODES TREE)) TREE))
                         'STATE)
                   (LIST 'EQUAL
                         (LIST 'TOTAL-OUTSTANDING
                               (LIST 'QUOTE (NODES TREE))
                               (LIST 'QUOTE TREE)
                               'STATE)
                         (LIST 'QUOTE (ADD1 COUNT))))))))))
(CLAIM
 (EVAL
  (LIST 'AND
        (LIST 'INV
              (LIST 'QUOTE TREE)
              'STATE)
        (LIST 'OR
              (LIST 'EQUAL
                    (LIST 'STATUS
                          (LIST 'QUOTE (CAR TREE))
                          'STATE)
                    ''STARTED)
              (LIST 'EQUAL
                    (LIST 'STATUS
                          (LIST 'QUOTE (CAR TREE))
                          'STATE)
                    ''NOT-STARTED)))
  (S
    (TREE-PRG TREE)
    (ILEADS (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))
            (TREE-PRG TREE)
            (LIST 'LESSP
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT))))))
 0)
(DROP 2)
(GENERALIZE
 (((S
    (TREE-PRG TREE)
    (ILEADS (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))
            (TREE-PRG TREE)
            (LIST 'LESSP
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
   STATE)))
```

```
(USE-LEMMA NOT-TOTAL-OUTSTANDING-0-IMPLIES-FULL-CHANNEL)
(PROVE (ENABLE TREEP PROPER-TREE))
(CONTRADICT 3)
(DROP 3)
(PROVE (DISABLE EVAL-OR EVAL)
       (ENABLE INV-IS-INVARIANT))
(REWRITE DISJOIN-LEFT)
(REWRITE NOT-STARTED-ROOT-DECREASES-TOTAL-OUTSTANDING)
(REWRITE DISJOIN-LEFT)
(CLAIM
 (FULL-CHANNEL
  (DOWN-LINKS (NODES TREE) TREE)
  (S
   (TREE-PRG TREE)
   (ILEADS
    (LIST 'AND
      (CONS 'INV
            (CONS (LIST 'QUOTE TREE) '(STATE)))
      (LIST 'AND
            (CONS 'FULL-CHANNEL
                  (CONS (LIST 'QUOTE
                              (DOWN-LINKS (NODES TREE) TREE))
                        '(STATE)))
            (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
    (TREE-PRG TREE)
    (LIST 'LESSP
          (CONS 'TOTAL-OUTSTANDING
                (CONS (LIST 'QUOTE (NODES TREE))
                      (CONS (LIST 'QUOTE TREE) '(STATE))))
          (LIST 'QUOTE (ADD1 COUNT))))))
 0)
(REWRITE LEADS-TO-STRENGTHEN-LEFT
 (($P
   (LIST 'AND
    (LIST 'INV
          (LIST 'QUOTE TREE)
          'STATE)
    (LIST 'AND
     (LIST 'LISTP
      (LIST 'CHANNEL
       (LIST 'QUOTE
        (CONS
         (PARENT
          (CDR
           (FULL-CHANNEL
            (DOWN-LINKS (NODES TREE) TREE)
            (S
             (TREE-PRG TREE)
             (ILEADS
              (LIST 'AND
               (CONS 'INV
                     (CONS (LIST 'QUOTE TREE) '(STATE)))
               (LIST 'AND
                (CONS 'FULL-CHANNEL
                      (CONS (LIST 'QUOTE
                                  (DOWN-LINKS (NODES TREE) TREE))
                            '(STATE)))
                (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
              (TREE-PRG TREE)
              (LIST 'LESSP
                  (CONS 'TOTAL-OUTSTANDING
```

```
                          (CONS (LIST 'QUOTE (NODES TREE))
                                (CONS (LIST 'QUOTE TREE) '(STATE))))
                    (LIST 'QUOTE (ADD1 COUNT))))))
            TREE)
           (CDR
            (FULL-CHANNEL
             (DOWN-LINKS (NODES TREE) TREE)
             (S
              (TREE-PRG TREE)
              (ILEADS
               (LIST 'AND
                (CONS 'INV
                      (CONS (LIST 'QUOTE TREE) '(STATE)))
                (LIST 'AND
                 (CONS 'FULL-CHANNEL
                       (CONS (LIST 'QUOTE
                                   (DOWN-LINKS (NODES TREE) TREE))
                             '(STATE)))
                 (LIST 'EQUAL
                    (CONS 'TOTAL-OUTSTANDING
                          (CONS (LIST 'QUOTE (NODES TREE))
                                (CONS (LIST 'QUOTE TREE) '(STATE))))
                    (LIST 'QUOTE (ADD1 COUNT)))))
               (TREE-PRG TREE)
               (LIST 'LESSP
                    (CONS 'TOTAL-OUTSTANDING
                          (CONS (LIST 'QUOTE (NODES TREE))
                                (CONS (LIST 'QUOTE TREE) '(STATE))))
                    (LIST 'QUOTE (ADD1 COUNT))))))))
         'STATE))
       (LIST 'EQUAL
             (LIST 'TOTAL-OUTSTANDING
                   (LIST 'QUOTE (NODES TREE))
                   (LIST 'QUOTE TREE)
                   'STATE)
             (LIST 'QUOTE (ADD1 COUNT))))))))
(GENERALIZE
 (((S
    (TREE-PRG TREE)
    (ILEADS
     (LIST 'AND
      (CONS 'INV
            (CONS (LIST 'QUOTE TREE) '(STATE)))
      (LIST 'AND
            (CONS 'FULL-CHANNEL
                  (CONS (LIST 'QUOTE
                              (DOWN-LINKS (NODES TREE) TREE))
                        '(STATE)))
            (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
     (TREE-PRG TREE)
     (LIST 'LESSP
           (CONS 'TOTAL-OUTSTANDING
                 (CONS (LIST 'QUOTE (NODES TREE))
                       (CONS (LIST 'QUOTE TREE) '(STATE))))
           (LIST 'QUOTE (ADD1 COUNT)))))
   STATE)))
(DROP 2)
BASH PROMOTE
(CLAIM
 (EQUAL
    (CONS (PARENT (CDR (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                                     STATE))
                  TREE)
          (CDR (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                             STATE)))
```

```
    (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                  STATE))
 0)
PROVE
(CONTRADICT 5)
(DROP 4 5)
(CLAIM (MEMBER (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                             STATE)
               (DOWN-LINKS (NODES TREE) TREE))
       0)
(DEMOTE 4)
(DIVE 1)
(REWRITE MEMBER-DOWN-LINKS)
TOP
(DROP 2 3)
(CLAIM
   (EQUAL (PARENT (CDR (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                                     STATE))
                  TREE)
          (CAR (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                             STATE)))
   0)
PROVE PROMOTE
(CONTRADICT 2)
(DROP 2)
(DIVE 1)
X
(DIVE 1)
(REWRITE PARENT-OF-CHILD
        (($PARENT (CAR (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                                     STATE)))))
TOP S
(DEMOTE 1)
(DIVE 1)
X TOP S
(DEMOTE 1)
(DIVE 1)
X
(DIVE 1)
(DIVE 1)
X TOP S
(DEMOTE 2)
(DIVE 1)
(DIVE 2)
(DIVE 1)
(DIVE 2)
X TOP S
(CONTRADICT 4)
(REWRITE FULL-CHANNEL-NOT-F-IMPLIES)
(REWRITE DOWN-LINK-FULL-DECREASES-TOTAL-OUTSTANDING)
(GENERALIZE
 (((S
    (TREE-PRG TREE)
    (ILEADS
     (LIST 'AND
      (CONS 'INV
            (CONS (LIST 'QUOTE TREE) '(STATE)))
      (LIST 'AND
            (CONS 'FULL-CHANNEL
                  (CONS (LIST 'QUOTE
                              (DOWN-LINKS (NODES TREE) TREE))
                        '(STATE)))
            (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
     (TREE-PRG TREE)
     (LIST 'LESSP
```

```
           (CONS 'TOTAL-OUTSTANDING
                 (CONS (LIST 'QUOTE (NODES TREE))
                       (CONS (LIST 'QUOTE TREE) '(STATE))))
           (LIST 'QUOTE (ADD1 COUNT)))))
   STATE)))
(DROP 2)
(CLAIM (MEMBER (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                             STATE)
               (DOWN-LINKS (NODES TREE) TREE))
       0)
(DEMOTE 3)
(DIVE 1)
(REWRITE MEMBER-DOWN-LINKS)
TOP SPLIT
(REWRITE MEMBER-OF-SUBLISTP-IS-MEMBER
   (($B (CHILDREN (CAR (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                                     STATE))
                  TREE))))
(REWRITE CHILDREN-ARE-SUFFIX-OF-SUBLIST-GENERALIZED
         (($TREE TREE)
          ($NODE (CAR (FULL-CHANNEL (DOWN-LINKS (NODES TREE) TREE)
                                    STATE)))))
(DEMOTE 1)
(DIVE 1)
X TOP S
(DEMOTE 1)
(DIVE 1)
X TOP S X
(DIVE 1)
X TOP
(REWRITE SUBLISTP-REFLEXIVE)
(CONTRADICT 3)
(REWRITE FULL-CHANNEL-NOT-F-IMPLIES)
(REWRITE LEADS-TO-STRENGTHEN-LEFT
         (($P '(FALSE))))
PROVE
(REWRITE FALSE-LEADS-TO-ANYTHING)
(CLAIM
 (FULL-CHANNEL
  (UP-LINKS (CDR (NODES TREE)) TREE)
  (S
   (TREE-PRG TREE)
   (ILEADS
    (LIST 'AND
      (CONS 'INV
            (CONS (LIST 'QUOTE TREE) '(STATE)))
      (LIST 'AND
            (CONS 'FULL-CHANNEL
                  (CONS (LIST 'QUOTE
                              (UP-LINKS (CDR (NODES TREE)) TREE))
                        '(STATE)))
            (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
    (TREE-PRG TREE)
    (LIST 'LESSP
          (CONS 'TOTAL-OUTSTANDING
                (CONS (LIST 'QUOTE (NODES TREE))
                      (CONS (LIST 'QUOTE TREE) '(STATE))))
          (LIST 'QUOTE (ADD1 COUNT))))))
 0)
(REWRITE LEADS-TO-STRENGTHEN-LEFT
 (($P
   (LIST 'AND
    (LIST 'INV
          (LIST 'QUOTE TREE)
          'STATE)
```

```
    (LIST 'AND
     (LIST 'LISTP
      (LIST 'CHANNEL
       (LIST 'QUOTE
        (CONS
         (CAR
          (FULL-CHANNEL
           (UP-LINKS (CDR (NODES TREE)) TREE)
           (S
            (TREE-PRG TREE)
            (ILEADS
             (LIST 'AND
              (CONS 'INV
                    (CONS (LIST 'QUOTE TREE) '(STATE)))
              (LIST 'AND
               (CONS 'FULL-CHANNEL
                    (CONS (LIST 'QUOTE
                                (UP-LINKS (CDR (NODES TREE)) TREE))
                          '(STATE)))
               (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
             (TREE-PRG TREE)
             (LIST 'LESSP
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))))
         (CDR
          (FULL-CHANNEL
           (UP-LINKS (CDR (NODES TREE)) TREE)
           (S
            (TREE-PRG TREE)
            (ILEADS
             (LIST 'AND
              (CONS 'INV
                    (CONS (LIST 'QUOTE TREE) '(STATE)))
              (LIST 'AND
               (CONS 'FULL-CHANNEL
                    (CONS (LIST 'QUOTE
                                (UP-LINKS (CDR (NODES TREE)) TREE))
                          '(STATE)))
               (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
             (TREE-PRG TREE)
             (LIST 'LESSP
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT))))))))))
       'STATE))
     (LIST 'EQUAL
           (LIST 'TOTAL-OUTSTANDING
                 (LIST 'QUOTE (NODES TREE))
                 (LIST 'QUOTE TREE)
                 'STATE)
           (LIST 'QUOTE (ADD1 COUNT)))))))
(GENERALIZE
 (((S
    (TREE-PRG TREE)
    (ILEADS
     (LIST 'AND
      (CONS 'INV
            (CONS (LIST 'QUOTE TREE) '(STATE)))
```

```
      (LIST 'AND
            (CONS 'FULL-CHANNEL
                  (CONS (LIST 'QUOTE
                              (UP-LINKS (CDR (NODES TREE)) TREE))
                        '(STATE)))
            (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
     (TREE-PRG TREE)
     (LIST 'LESSP
           (CONS 'TOTAL-OUTSTANDING
                 (CONS (LIST 'QUOTE (NODES TREE))
                       (CONS (LIST 'QUOTE TREE) '(STATE))))
           (LIST 'QUOTE (ADD1 COUNT)))))
   STATE)))
(DROP 2)
BASH PROMOTE
(CONTRADICT 5)
(DROP 3 4 5)
(CLAIM (MEMBER (FULL-CHANNEL (UP-LINKS (CDR (NODES TREE)) TREE)
                             STATE)
               (UP-LINKS (CDR (NODES TREE)) TREE)))
(DEMOTE 3)
(DIVE 1)
(REWRITE MEMBER-UP-LINKS)
TOP S
(REWRITE UP-LINK-FULL-DECREASES-TOTAL-OUTSTANDING)
(GENERALIZE
 (((S
    (TREE-PRG TREE)
    (ILEADS
     (LIST 'AND
      (CONS 'INV
            (CONS (LIST 'QUOTE TREE) '(STATE)))
      (LIST 'AND
            (CONS 'FULL-CHANNEL
                  (CONS (LIST 'QUOTE
                              (UP-LINKS (CDR (NODES TREE)) TREE))
                        '(STATE)))
            (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
     (TREE-PRG TREE)
     (LIST 'LESSP
           (CONS 'TOTAL-OUTSTANDING
                 (CONS (LIST 'QUOTE (NODES TREE))
                       (CONS (LIST 'QUOTE TREE) '(STATE))))
           (LIST 'QUOTE (ADD1 COUNT)))))
   STATE)))
(DROP 2)
(CLAIM (MEMBER (FULL-CHANNEL (UP-LINKS (CDR (NODES TREE)) TREE)
                             STATE)
               (UP-LINKS (CDR (NODES TREE)) TREE)))
(DEMOTE 3)
(DIVE 1)
(REWRITE MEMBER-UP-LINKS)
TOP SPLIT
(DIVE 1)
= X TOP
(DIVE 2)
X TOP
(REWRITE NODE-HAS-PARENT)
(DIVE 2)
(= * (NODES TREE) ((ENABLE NODES)))
TOP
```

```
(REWRITE MEMBER-CDR-NODES-MEMBER-NODES)
(DEMOTE 1)
(DIVE 1)
X TOP S S
(DIVE 1)
(DROP 2 3 4)
TOP
(DIVE 1)
(DIVE 2)
(= *
   (CAR (NODES TREE))
   ((ENABLE NODES NODES-REC)))
TOP
(=
  (NOT (EQUAL (CAR (FULL-CHANNEL (UP-LINKS (CDR (NODES TREE)) TREE)
                                 STATE))
              (CAR (NODES TREE)))))
(CONTRADICT 2)
(DIVE 1)
(DIVE 1)
= TOP
(DEMOTE 1)
(DIVE 1)
X
(DIVE 1)
X TOP PROVE
(GENERALIZE
 (((S
    (TREE-PRG TREE)
    (ILEADS
     (LIST 'AND
      (CONS 'INV
            (CONS (LIST 'QUOTE TREE) '(STATE)))
      (LIST 'AND
            (CONS 'FULL-CHANNEL
                  (CONS (LIST 'QUOTE
                              (UP-LINKS (CDR (NODES TREE)) TREE))
                        '(STATE)))
            (LIST 'EQUAL
                  (CONS 'TOTAL-OUTSTANDING
                        (CONS (LIST 'QUOTE (NODES TREE))
                              (CONS (LIST 'QUOTE TREE) '(STATE))))
                  (LIST 'QUOTE (ADD1 COUNT)))))
     (TREE-PRG TREE)
     (LIST 'LESSP
           (CONS 'TOTAL-OUTSTANDING
                 (CONS (LIST 'QUOTE (NODES TREE))
                       (CONS (LIST 'QUOTE TREE) '(STATE))))
           (LIST 'QUOTE (ADD1 COUNT)))))
   STATE)))
(DROP 2)
(CLAIM (MEMBER (FULL-CHANNEL (UP-LINKS (CDR (NODES TREE)) TREE)
                             STATE)
               (UP-LINKS (CDR (NODES TREE)) TREE)))
(DEMOTE 3)
(DIVE 1)
(REWRITE MEMBER-UP-LINKS)
TOP SPLIT
(DIVE 2)
(DIVE 1)
= X UP X TOP
(REWRITE PARENT-REC-CHILDREN-REC)
(DIVE 2)
(REWRITE SETP-TREE-UNIQUE-PARENT)
(DIVE 1)
(DIVE 2)
(= * (NODES TREE) ((ENABLE NODES)))
UP
(REWRITE MEMBER-CDR-NODES-MEMBER-NODES)
```

```
    UP S
    (DIVE 1)
    (= * F 0)
    TOP S X
    (= *
      (NOT (EQUAL (CAR (NODES TREE))
                  (CAR (FULL-CHANNEL (UP-LINKS (CDR (NODES TREE)) TREE)
                                     STATE))))
      ((ENABLE NODES NODES-REC)))
    (CONTRADICT 5)
    (DIVE 1)
    (DIVE 1)
    = TOP
    (DEMOTE 1)
    (DIVE 1)
    X
    (DIVE 1)
    X TOP S
    (DEMOTE 1)
    (DIVE 1)
    X TOP S
    (DEMOTE 1)
    (DIVE 1)
    X
    (DIVE 1)
    (DIVE 1)
    X TOP S
    (REWRITE LEADS-TO-STRENGTHEN-LEFT
             (($P '(FALSE))))
    PROVE
    (REWRITE FALSE-LEADS-TO-ANYTHING))))


  (PROVE-LEMMA TERMINATION-INDUCTION (REWRITE)
               (IMPLIES (AND (TREEP TREE)
                             (INITIAL-CONDITION
                              `(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS TREE))
                                               STATE)
                                    (NOT-STARTED (QUOTE ,(NODES TREE))
                                                 STATE))
                              (TREE-PRG TREE)))
                        (LEADS-TO `(LESSP (TOTAL-OUTSTANDING
                                           (QUOTE ,(NODES TREE))
                                           (QUOTE ,TREE)
                                           STATE)
                                          (QUOTE ,(ADD1 COUNT)))
                                  `(EQUAL (TOTAL-OUTSTANDING
                                           (QUOTE ,(NODES TREE))
                                           (QUOTE ,TREE)
                                           STATE)
                                          0)
                                  (TREE-PRG TREE)))
      ((INSTRUCTIONS (INDUCT (PLUS COUNT J)) PROMOTE PROMOTE
           (REWRITE LEADS-TO-STRENGTHEN-LEFT
                    (($P (CONS 'EQUAL
                               (CONS (CONS 'TOTAL-OUTSTANDING
                                      (CONS (LIST 'QUOTE (NODES TREE))
                                       (CONS (LIST 'QUOTE TREE) '(STATE))))
                                     '(0))))))
           PROVE (REWRITE Q-LEADS-TO-Q) PROMOTE PROMOTE (DEMOTE 2)
           (DIVE 1) (DIVE 1) S-PROP TOP SPLIT
           (REWRITE LEADS-TO-TRANSITIVE
                    (($Q (LIST 'LESSP
                               (CONS 'TOTAL-OUTSTANDING
                                     (CONS (LIST 'QUOTE (NODES TREE))
                                      (CONS (LIST 'QUOTE TREE) '(STATE))))
                               (LIST 'QUOTE (ADD1 (SUB1 COUNT)))))))
           (DROP 4)
           (CLAIM (NOT (LESSP (TOTAL-OUTSTANDING (NODES TREE) TREE
```

```
                              (S (TREE-PRG TREE)
                                 (ILEADS
                                  (LIST 'LESSP
                                   (CONS 'TOTAL-OUTSTANDING
                                    (CONS (LIST 'QUOTE (NODES TREE))
                                     (CONS (LIST 'QUOTE TREE)
                                      '(STATE))))
                                   (LIST 'QUOTE (ADD1 COUNT)))
                                  (TREE-PRG TREE)
                                  (LIST 'LESSP
                                   (CONS 'TOTAL-OUTSTANDING
                                    (CONS (LIST 'QUOTE (NODES TREE))
                                     (CONS (LIST 'QUOTE TREE)
                                      '(STATE))))
                                   (LIST 'QUOTE (ADD1 (SUB1 COUNT))))))
                          (ADD1 (SUB1 COUNT))))
                0)
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'EQUAL
                             (LIST 'TOTAL-OUTSTANDING
                                   (LIST 'QUOTE (NODES TREE))
                                   (LIST 'QUOTE TREE) 'STATE)
                             (LIST 'QUOTE (ADD1 (SUB1 COUNT)))))))
         (DROP 2 3) PROVE
         (REWRITE TOTAL-OUTSTANDING-DECREASES-LEADS-TO)
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'LESSP
                             (CONS 'TOTAL-OUTSTANDING
                                   (CONS (LIST 'QUOTE (NODES TREE))
                                    (CONS (LIST 'QUOTE TREE) '(STATE))))
                             (LIST 'QUOTE (ADD1 (SUB1 COUNT)))))))
         (DROP 3) PROVE (REWRITE Q-LEADS-TO-Q))))


(PROVE-LEMMA TERMINATION (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (INITIAL-CONDITION
                            '(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS TREE))
                                             STATE)
                                  (NOT-STARTED (QUOTE ,(NODES TREE))
                                               STATE))
                            (TREE-PRG TREE)))
                      (LEADS-TO '(TRUE)
                                '(EQUAL (TOTAL-OUTSTANDING
                                         (QUOTE ,(NODES TREE))
                                         (QUOTE ,TREE)
                                         STATE)
                                        0)
                                (TREE-PRG TREE)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'LESSP
                             (LIST 'TOTAL-OUTSTANDING
                                   (LIST 'QUOTE (NODES TREE))
                                   (LIST 'QUOTE TREE) 'STATE)
                             (LIST 'QUOTE
                                   (ADD1
                                    (TOTAL-OUTSTANDING (NODES TREE)
                                     TREE
                                     (S (TREE-PRG TREE)
                                      (ILEADS '(TRUE) (TREE-PRG TREE)
                                       (CONS 'EQUAL
                                        (CONS
                                         (CONS 'TOTAL-OUTSTANDING
                                          (CONS
                                           (LIST 'QUOTE (NODES TREE))
                                           (CONS (LIST 'QUOTE TREE)
                                            '(STATE))))
                                         '(0))))))))))))
```

```
          PROVE (REWRITE TERMINATION-INDUCTION))))

(PROVE-LEMMA CORRECTNESS-CONDITION (REWRITE)
             (IMPLIES (AND (TREEP TREE)
                           (INITIAL-CONDITION
                            `(AND (ALL-EMPTY (QUOTE ,(ALL-CHANNELS TREE))
                                             STATE)
                                  (NOT-STARTED (QUOTE ,(NODES TREE))
                                               STATE))
                            (TREE-PRG TREE)))
                      (LEADS-TO '(TRUE)
                                `(CORRECT (QUOTE ,TREE) STATE)
                                (TREE-PRG TREE)))
    ((INSTRUCTIONS PROMOTE
          (REWRITE LEADS-TO-WEAKEN-RIGHT
                   (($Q (LIST 'EQUAL
                              (LIST 'TOTAL-OUTSTANDING
                                    (LIST 'QUOTE (NODES TREE))
                                    (LIST 'QUOTE TREE) 'STATE)
                              0))))
          BASH PROMOTE
          (REWRITE INV-IMPLIES-AUGMENTED-CORRECTNESS-CONDITION)
          (DEMOTE 1) (DIVE 1) X TOP S (DEMOTE 1) (DIVE 1) X (DIVE 1)
          (DIVE 1) X TOP S
          (CLAIM (EVAL (LIST 'INV (LIST 'QUOTE TREE) 'STATE)
                       (S (TREE-PRG TREE)
                          (JLEADS (ILEADS '(TRUE) (TREE-PRG TREE)
                                          (CONS 'CORRECT
                                           (CONS (LIST 'QUOTE TREE)
                                            '(STATE))))
                                  (TREE-PRG TREE)
                                  (CONS 'EQUAL
                                        (CONS
                                         (CONS 'TOTAL-OUTSTANDING
                                          (CONS
                                           (LIST 'QUOTE (NODES TREE))
                                           (CONS (LIST 'QUOTE TREE)
                                            '(STATE))))
                                         '(0))))))
                 ((DISABLE EVAL) (ENABLE INV-IS-INVARIANT)))
          PROVE (REWRITE TERMINATION))))


))
```

Appendix E
Dining Philosophers Events

This appendix contains the complete events list supporting the prrof of the dining philosophers algorithm described in chapter 6.

This event list constructs the proof of the dining philosophers algorithm on top of the library created by the events in Appendix B.

```
(NOTE-LIB "INTERPRETER")

(PROVEALL "DINING" '(



;;; THE PROGRAM

(DEFN STATUS (STATE INDEX)
  (CDR (ASSOC (CONS 'S INDEX) STATE)))

(DEFN THINKING (STATE INDEX)
  (EQUAL (STATUS STATE INDEX) 'THINKING))

(DEFN HUNGRY (STATE INDEX)
  (EQUAL (STATUS STATE INDEX) 'HUNGRY))

(DEFN EATING (STATE INDEX)
  (EQUAL (STATUS STATE INDEX) 'EATING))

(DEFN FORK (STATE INDEX)
  (CDR (ASSOC (CONS 'F INDEX) STATE)))

(DEFN FREE (STATE INDEX)
  (EQUAL (FORK STATE INDEX) 'FREE))

(DEFN OWNS-LEFT (STATE INDEX)
  (EQUAL (FORK STATE INDEX) INDEX))

(DEFN OWNS-RIGHT (STATE INDEX N)
  (EQUAL (FORK STATE (ADD1-MOD N INDEX)) INDEX))

(DEFN THINKING-TO (OLD NEW INDEX)
  (IF (THINKING OLD INDEX)
      (AND (OR (THINKING NEW INDEX)
               (HUNGRY NEW INDEX))
           (CHANGED OLD NEW (LIST (CONS 'S INDEX))))
    (CHANGED OLD NEW NIL)))

(DEFN HUNGRY-LEFT (OLD NEW INDEX)
  (AND (HUNGRY OLD INDEX)
       (FREE OLD INDEX)
       (OWNS-LEFT NEW INDEX)
       (CHANGED OLD NEW (LIST (CONS 'F INDEX)))))

(DEFN HUNGRY-RIGHT (OLD NEW INDEX N)
  (AND (HUNGRY OLD INDEX)
       (FREE OLD (ADD1-MOD N INDEX))
       (OWNS-RIGHT NEW INDEX N)
       (CHANGED OLD NEW (LIST (CONS 'F (ADD1-MOD N INDEX))))))

(DEFN HUNGRY-BOTH (OLD NEW INDEX N)
  (IF (AND (HUNGRY OLD INDEX)
           (OWNS-LEFT OLD INDEX)
           (OWNS-RIGHT OLD INDEX N))
```

```
      (AND (EATING NEW INDEX)
           (CHANGED OLD NEW (LIST (CONS 'S INDEX))))
    (CHANGED OLD NEW NIL)))

(DEFN EATING-TO (OLD NEW INDEX N)
  (IF (EATING OLD INDEX)
      (AND (THINKING NEW INDEX)
           (FREE NEW INDEX)
           (FREE NEW (ADD1-MOD N INDEX))
           (CHANGED OLD NEW (LIST (CONS 'S INDEX)
                                  (CONS 'F INDEX)
                                  (CONS 'F (ADD1-MOD N INDEX)))))
    (CHANGED OLD NEW NIL)))

(DEFN PHIL (INDEX N)
  (LIST (LIST 'THINKING-TO INDEX)
        (LIST 'HUNGRY-LEFT INDEX)
        (LIST 'HUNGRY-RIGHT INDEX N)
        (LIST 'HUNGRY-BOTH INDEX N)
        (LIST 'EATING-TO INDEX N)))

(DEFN RING (INDEX N)
  (IF (ZEROP INDEX)
      NIL
    (APPEND (PHIL (SUB1 INDEX) N)
            (RING (SUB1 INDEX) N))))

(DEFN PHIL-PRG (N)
  (RING N N))

(DISABLE PHIL-PRG)
(DISABLE *1*PHIL-PRG)


;;; CORRECTNESS


(PROVE-LEMMA MEMBER-RING (REWRITE)
    (EQUAL (MEMBER STATEMENT (RING INDEX N))
           (AND (MEMBER STATEMENT (PHIL (CADR STATEMENT) N))
                (NUMBERP (CADR STATEMENT))
                (LESSP (CADR STATEMENT) INDEX)))
    ((INSTRUCTIONS INDUCT BASH
         (CLAIM (EQUAL (CADR STATEMENT) (SUB1 INDEX)) 0)
         (BASH (DISABLE PHIL)) PROMOTE (DIVE 1) (DIVE 2) X UP
         (REWRITE MEMBER-APPEND) (DIVE 1) (= F) UP S = TOP (DROP 3)
         (BASH (DISABLE PHIL)) PROMOTE (DROP 4)
         (GENERALIZE (((CADR STATEMENT) ?))) PROVE)))

(PROVE-LEMMA MEMBER-PHIL-PRG (REWRITE)
             (EQUAL (MEMBER STATEMENT (PHIL-PRG N))
                    (AND (MEMBER STATEMENT (PHIL (CADR STATEMENT) N))
                         (NUMBERP (CADR STATEMENT))
                         (LESSP (CADR STATEMENT) N)))
             ((DISABLE PHIL)
              (ENABLE PHIL-PRG)))

(DEFN PROPER-FORKS-REC (STATE INDEX N)
  (IF (ZEROP INDEX)
      T
    (AND (OR (FREE STATE (SUB1 INDEX))
             (EQUAL (FORK STATE (SUB1 INDEX)) (SUB1 INDEX))
             (EQUAL (FORK STATE (SUB1 INDEX)) (SUB1-MOD N (SUB1 INDEX))))
         (PROPER-FORKS-REC STATE (SUB1 INDEX) N))))

(PROVE-LEMMA PROPER-FORKS-REC-IMPLIES (REWRITE)
             (IMPLIES (AND (PROPER-FORKS-REC STATE INDEX N)
                           (NOT (LESSP N INDEX))
                           (LESSP 1 N)
```

```
                       (LESSP I INDEX)
                       (NUMBERP I))
                  (OR (FREE STATE I)
                      (EQUAL (FORK STATE I) I)
                      (EQUAL (FORK STATE I) (SUB1-MOD N I))))
             ((INDUCT (PROPER-FORKS-REC STATE INDEX N))))

(DEFN PROPER-FORKS (STATE N)
  (PROPER-FORKS-REC STATE N N))

(PROVE-LEMMA PROPER-FORKS-IMPLIES (REWRITE)
             (IMPLIES (AND (PROPER-FORKS STATE N)
                           (LESSP 1 N)
                           (LESSP I N)
                           (NUMBERP I))
                      (OR (FREE STATE I)
                          (EQUAL (FORK STATE I) I)
                          (EQUAL (FORK STATE I) (SUB1-MOD N I))))
             ((USE (PROPER-FORKS-REC-IMPLIES (INDEX N)))))

(DEFN PROPER-PHIL (STATE PHIL RIGHT)
  (AND (IMPLIES (THINKING STATE PHIL)
                (AND (NOT (EQUAL (FORK STATE PHIL) PHIL))
                     (NOT (EQUAL (FORK STATE RIGHT) PHIL))))
       (IMPLIES (EATING STATE PHIL)
                (AND (EQUAL (FORK STATE PHIL) PHIL)
                     (EQUAL (FORK STATE RIGHT) PHIL)))
       (OR (THINKING STATE PHIL)
           (HUNGRY STATE PHIL)
           (EATING STATE PHIL))))

(DEFN PROPER-PHILS-REC (STATE INDEX N)
  (IF (ZEROP INDEX)
      T
    (AND (PROPER-PHIL STATE (SUB1 INDEX) (ADD1-MOD N (SUB1 INDEX)))
         (PROPER-PHILS-REC STATE (SUB1 INDEX) N))))

(PROVE-LEMMA PROPER-PHILS-REC-IMPLIES-PROPER-PHIL (REWRITE)
             (IMPLIES (AND (PROPER-PHILS-REC STATE INDEX N)
                           (LESSP 1 N)
                           (NOT (LESSP N INDEX))
                           (LESSP PHIL INDEX)
                           (NUMBERP PHIL)
                           (EQUAL RIGHT (ADD1-MOD N PHIL)))
                      (PROPER-PHIL STATE PHIL RIGHT))
             ((INDUCT (PROPER-PHILS-REC STATE INDEX N))
              (DISABLE PROPER-PHIL)))

(PROVE-LEMMA PROPER-PHILS-REC-IMPLIES (REWRITE)
             (IMPLIES (AND (PROPER-PHILS-REC STATE INDEX N)
                           (NOT (LESSP N INDEX))
                           (LESSP 1 N)
                           (LESSP PHIL INDEX)
                           (NUMBERP PHIL))
                      (AND (IMPLIES (THINKING STATE PHIL)
                                    (AND (NOT (OWNS-LEFT STATE PHIL))
                                         (NOT (OWNS-RIGHT STATE PHIL N))))
                           (IMPLIES (EATING STATE PHIL)
                                    (AND (OWNS-LEFT STATE PHIL)
                                         (OWNS-RIGHT STATE PHIL N)))
                           (OR (THINKING STATE PHIL)
                               (HUNGRY STATE PHIL)
                               (EATING STATE PHIL))))
             ((USE (PROPER-PHILS-REC-IMPLIES-PROPER-PHIL
                    (RIGHT (ADD1-MOD N PHIL))))
              (DISABLE PROPER-PHILS-REC-IMPLIES-PROPER-PHIL)))

(DEFN PROPER-PHILS (STATE N)
  (PROPER-PHILS-REC STATE N N))
```

```
(PROVE-LEMMA PROPER-PHILS-IMPLIES-PROPER-PHIL (REWRITE)
             (IMPLIES (AND (PROPER-PHILS STATE N)
                           (LESSP 1 N)
                           (LESSP PHIL N)
                           (NUMBERP PHIL)
                           (EQUAL RIGHT (ADD1-MOD N PHIL)))
                      (PROPER-PHIL STATE PHIL RIGHT))
             ((USE (PROPER-PHILS-REC-IMPLIES-PROPER-PHIL (INDEX N)))
              (DISABLE PROPER-PHIL
                       PROPER-PHILS-REC-IMPLIES-PROPER-PHIL)))



(DEFN ALL-LEFTS (STATE INDEX)
  (IF (ZEROP INDEX)
      T
    (AND (OWNS-LEFT STATE (SUB1 INDEX))
         (HUNGRY STATE (SUB1 INDEX))
         (ALL-LEFTS STATE (SUB1 INDEX)))))

(PROVE-LEMMA ALL-LEFTS-IMPLIES (REWRITE)
             (IMPLIES (AND (LESSP I N)
                           (NUMBERP I)
                           (ALL-LEFTS STATE N))
                      (AND (HUNGRY STATE I)
                           (OWNS-LEFT STATE I)))
             ((INDUCT (ALL-LEFTS STATE N))))


(DEFN ALL-RIGHTS-REC (STATE INDEX N)
  (IF (ZEROP INDEX)
      T
    (AND (OWNS-RIGHT STATE (SUB1 INDEX) N)
         (HUNGRY STATE (SUB1 INDEX))
         (ALL-RIGHTS-REC STATE (SUB1 INDEX) N))))

(PROVE-LEMMA ALL-RIGHTS-REC-IMPLIES (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (NOT (LESSP N INDEX))
                           (LESSP I INDEX)
                           (NUMBERP I)
                           (ALL-RIGHTS-REC STATE INDEX N))
                      (AND (HUNGRY STATE I)
                           (OWNS-RIGHT STATE I N)))
             ((INDUCT (ALL-RIGHTS-REC STATE INDEX N))))

(DEFN ALL-RIGHTS (STATE N)
  (ALL-RIGHTS-REC STATE N N))

(PROVE-LEMMA ALL-RIGHTS-IMPLIES (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP I N)
                           (NUMBERP I)
                           (ALL-RIGHTS STATE N))
                      (AND (HUNGRY STATE I)
                           (OWNS-RIGHT STATE I N)))
             ((USE (ALL-RIGHTS-REC-IMPLIES (INDEX N)))))


;(DEFN INITIAL (STATE N)
;  (AND (PROPER-FORKS STATE N)
;       (PROPER-PHILS STATE N)))

(PROVE-LEMMA LISTP-PHIL-PRG (REWRITE)
             (EQUAL (LISTP (PHIL-PRG N))
                    (NOT (ZEROP N)))
             ((ENABLE PHIL-PRG N)))
```

```
(DEFN PFUSI (INDEX N STATEMENT)
  (IF (ZEROP INDEX)
      T
    (IF (EQUAL (SUB1 INDEX) (CADR STATEMENT))
        (PFUSI (SUB1 INDEX) N STATEMENT)
      (IF (EQUAL (SUB1 INDEX) (ADD1-MOD N (CADR STATEMENT)))
          (PFUSI (SUB1 INDEX) N STATEMENT)
        (PFUSI (SUB1 INDEX) N STATEMENT)))))

(PROVE-LEMMA PROPER-FORKS-REC-UNLESS-SUFFICIENT (REWRITE)
             (IMPLIES (AND (NOT (LESSP N INDEX))
                           (LESSP 1 N))
                      (UNLESS-SUFFICIENT STATEMENT
                                         (PHIL-PRG N)
                                         OLD NEW
                                         '(PROPER-FORKS-REC STATE
                                                            (QUOTE ,INDEX)
                                                            (QUOTE ,N))
                                         '(FALSE)))
             ((INDUCT (PFUSI INDEX N STATEMENT))))


(DEFN PROPER-TRIPLE (STATE PHIL N)
  (AND (PROPER-PHIL STATE (SUB1-MOD N PHIL)
                    (ADD1-MOD N (SUB1-MOD N PHIL)))
       (PROPER-PHIL STATE PHIL (ADD1-MOD N PHIL))
       (PROPER-PHIL STATE (ADD1-MOD N PHIL)
                    (ADD1-MOD N (ADD1-MOD N PHIL)))))

(DEFN BUT-TRIPLE (STATE INDEX PHIL N)
  (IF (ZEROP INDEX)
      T
    (IF (OR (EQUAL (SUB1 INDEX) (SUB1-MOD N PHIL))
            (EQUAL (SUB1 INDEX) PHIL)
            (EQUAL (SUB1 INDEX) (ADD1-MOD N PHIL)))
        (BUT-TRIPLE STATE (SUB1 INDEX) PHIL N)
      (AND (PROPER-PHIL STATE (SUB1 INDEX) (ADD1-MOD N (SUB1 INDEX)))
           (BUT-TRIPLE STATE (SUB1 INDEX) PHIL N)))))

(PROVE-LEMMA LESSP-1 (REWRITE)
             (EQUAL (LESSP 1 N)
                    (AND (NOT (ZEROP N))
                         (NOT (EQUAL N 1)))))

(PROVE-LEMMA LESSP-2 (REWRITE)
             (EQUAL (LESSP X 2)
                    (OR (EQUAL X 1)
                        (ZEROP X))))

(PROVE-LEMMA LESSP-2-2 (REWRITE)
             (EQUAL (LESSP 2 X)
                    (AND (NOT (ZEROP X))
                         (NOT (EQUAL X 1))
                         (NOT (EQUAL X 2)))))

(PROVE-LEMMA PROPER-TRIPLE-PRESERVED (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP PHIL N)
                           (NUMBERP PHIL)
                           (MEMBER STATEMENT (PHIL PHIL N))
                           (N OLD NEW STATEMENT)
                           (PROPER-TRIPLE OLD PHIL N))
                      (PROPER-TRIPLE NEW PHIL N))
  ((INSTRUCTIONS
    PROMOTE
    (BOOKMARK (BEGIN ELIM))
    (CLAIM (EQUAL STATEMENT
                  (CONS (CAR STATEMENT) (CDR STATEMENT)))
           0)
```

```
    (CHANGE-GOAL (MAIN . 1)) (CONTRADICT 7) S (CHANGE-GOAL MAIN)
    (GENERALIZE (((CAR STATEMENT) X) ((CDR STATEMENT) Z)))
    (SUBV (STATEMENT (CONS X Z))) (DROP 7) (BOOKMARK (END ELIM))
    (CLAIM (OR (AND (EQUAL X 'THINKING-TO) (EQUAL Z (LIST PHIL)))
               (AND (EQUAL X 'HUNGRY-LEFT) (EQUAL Z (LIST PHIL)))
               (AND (EQUAL X 'HUNGRY-RIGHT)
                    (EQUAL Z (LIST PHIL N)))
               (AND (EQUAL X 'HUNGRY-BOTH)
                    (EQUAL Z (LIST PHIL N)))
               (AND (EQUAL X 'EATING-TO) (EQUAL Z (LIST PHIL N))))
           0)
    (DROP 4)
    (CLAIM (LESSP 2 N) 0)
    (CLAIM (EQUAL PHIL (SUB1 N)) 0)
    (CLAIM (EQUAL PHIL 0) 0)
    (CONTRADICT 7)
    (DROP 4 5 6 7)
    PROVE
    PROVE
    (CLAIM (EQUAL PHIL 0) 0)
    PROVE
    PROVE
    PROVE
    (CONTRADICT 7)
    (DROP 1 2 3 5 6 7)
    PROVE
    (DROP 1 2 3 5 6 7) PROVE)))

(PROVE-LEMMA BUT-TRIPLE-PRESERVED (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (NOT (LESSP N INDEX))
                           (LESSP PHIL N)
                           (NUMBERP PHIL)
                           (MEMBER STATEMENT (PHIL PHIL N))
                           (N OLD NEW STATEMENT)
                           (BUT-TRIPLE OLD INDEX PHIL N))
                      (BUT-TRIPLE NEW INDEX PHIL N))
  ((INSTRUCTIONS
    (INDUCT (BUT-TRIPLE NEW INDEX PHIL N)) PROVE PROMOTE
    PROMOTE X-DUMB (DIVE 3) (DIVE 1) (= * T 0) TOP S-PROP
    (DEMOTE 10) (DIVE 1) X-DUMB (DIVE 3) (DIVE 1) (= * T 0) UP UP
    S-PROP TOP PROMOTE (DEMOTE 3) (DIVE 1) (DIVE 1)
    (= * T ((DISABLE LESSP-1 ZEROP PHIL N ADD1-MOD SUB1-MOD))) TOP
    S-PROP (DROP 1 3 4 5 6 7 8 9 10) PROVE
    (DROP 1 3 4 5 6 7 8 9 10 11) PROVE PROMOTE PROMOTE (DEMOTE 10)
    (DIVE 1) X-DUMB (DIVE 3) (DIVE 1) (= * F 0) UP UP S-PROP TOP
    PROMOTE X-DUMB (DIVE 3) (DIVE 1) (= * F 0) TOP S-PROP
    (DEMOTE 3) (DIVE 1) (DIVE 1)
    (= * (BUT-TRIPLE OLD (SUB1 INDEX) PHIL N) 0) TOP (DEMOTE 9)
    S-PROP SPLIT (DROP 12 13) PROVE (DROP 9 2)
    (PROVE (DISABLE PHIL N)) (DROP 1 3 4 5 6 7 8 9 10 11) PROVE
    (DROP 1 3 4 5 6 7 8 9 10) PROVE)))

(PROVE-LEMMA BUT-TRIPLE-AND-TRIPLE-ALL (REWRITE)
  (IMPLIES
   (AND (NUMBERP PHIL)
        (NOT (LESSP N INDEX)))
   (EQUAL
    (PROPER-PHILS-REC STATE INDEX N)
    (AND (BUT-TRIPLE STATE INDEX PHIL N)
         (IF (LESSP (ADD1-MOD N PHIL) INDEX)
             (IF (LESSP PHIL INDEX)
                 (IF (LESSP (SUB1-MOD N PHIL) INDEX)
                     (PROPER-TRIPLE STATE PHIL N)
                   (AND (PROPER-PHIL STATE PHIL
                                     (ADD1-MOD N PHIL))
                        (PROPER-PHIL STATE (ADD1-MOD N
                                                     PHIL)
                                     (ADD1-MOD N (ADD1-MOD N PHIL)))))
```

```
                (IF (LESSP (SUB1-MOD N PHIL) INDEX)
                    (AND (PROPER-PHIL STATE (SUB1-MOD N PHIL)
                                      (ADD1-MOD N (SUB1-MOD N PHIL)))
                         (PROPER-PHIL STATE (ADD1-MOD N PHIL)
                                      (ADD1-MOD N (ADD1-MOD N PHIL))))
                  (PROPER-PHIL STATE (ADD1-MOD N PHIL)
                               (ADD1-MOD N (ADD1-MOD N PHIL)))))
            (IF (LESSP PHIL INDEX)
                (IF (LESSP (SUB1-MOD N PHIL) INDEX)
                    (AND (PROPER-PHIL STATE PHIL (ADD1-MOD N PHIL))
                         (PROPER-PHIL STATE (SUB1-MOD N PHIL)
                                      (ADD1-MOD N (SUB1-MOD N PHIL))))
                  (PROPER-PHIL STATE PHIL (ADD1-MOD N PHIL)))
              (IF (LESSP (SUB1-MOD N PHIL) INDEX)
                  (PROPER-PHIL STATE (SUB1-MOD N PHIL)
                               (ADD1-MOD N (SUB1-MOD N PHIL)))
                T))))))
 ((DISABLE ADD1-MOD SUB1-MOD PROPER-PHIL)
  (INDUCT (BUT-TRIPLE STATE INDEX PHIL N))))

(PROVE-LEMMA PROPER-PHILS-PRESERVED (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (MEMBER STATEMENT (PHIL-PRG N))
                           (N OLD NEW STATEMENT)
                           (PROPER-PHILS OLD N))
                      (PROPER-PHILS NEW N))
             ((DISABLE PROPER-TRIPLE PHIL PROPER-PHIL N)))

(DISABLE BUT-TRIPLE-AND-TRIPLE-ALL)
(DISABLE BUT-TRIPLE-PRESERVED)
(DISABLE PROPER-TRIPLE-PRESERVED)
(DISABLE PROPER-PHILS)


(PROVE-LEMMA PROPER-PHILS-UNLESS-SUFFICIENT (REWRITE)
    (IMPLIES (LESSP 1 N)
             (UNLESS-SUFFICIENT STATEMENT (PHIL-PRG N)
                                OLD NEW
                                '(PROPER-PHILS STATE (QUOTE ,N))
                                '(FALSE)))
    ((DISABLE MEMBER-PHIL-PRG N)))

(PROVE-LEMMA ALL-LEFTS-UNCHANGED (REWRITE)
             (IMPLIES (AND (ALL-LEFTS OLD N)
                           (CHANGED OLD NEW NIL))
                      (ALL-LEFTS NEW N)))

(PROVE-LEMMA ALL-LEFTS-UNLESS-SUFFICIENT (REWRITE)
             (IMPLIES (LESSP 1 N)
                      (UNLESS-SUFFICIENT STATEMENT
                                         (PHIL-PRG N)
                                         OLD NEW
                                         '(ALL-LEFTS STATE (QUOTE ,N))
                                         '(FALSE)))
             ((USE (ALL-LEFTS-IMPLIES (I (CADR STATEMENT)) (STATE OLD))
                   (ALL-LEFTS-IMPLIES (I (ADD1-MOD N (CADR STATEMENT)))
                                      (STATE OLD)))
              (DISABLE ALL-LEFTS-IMPLIES)))

(DISABLE ALL-LEFTS-IMPLIES)
(DISABLE ALL-LEFTS-UNCHANGED)

(PROVE-LEMMA ALL-RIGHTS-REC-UNCHANGED (REWRITE)
             (IMPLIES (AND (ALL-RIGHTS-REC OLD INDEX N)
                           (CHANGED OLD NEW NIL)
                           (NOT (LESSP N INDEX)))
                      (ALL-RIGHTS-REC NEW INDEX N)))

(PROVE-LEMMA ALL-RIGHTS-UNLESS-SUFFICIENT (REWRITE)
```

```
              (IMPLIES (LESSP 1 N)
                       (UNLESS-SUFFICIENT STATEMENT
                                          (PHIL-PRG N)
                                          OLD NEW
                                          '(ALL-RIGHTS STATE (QUOTE ,N))
                                          '(FALSE)))
              ((USE (ALL-RIGHTS-IMPLIES (I (CADR STATEMENT)) (STATE OLD)
                                        (N N))
                    (ALL-RIGHTS-IMPLIES (I (SUB1-MOD N (CADR STATEMENT)))
                                        (STATE OLD) (N N)))
               (DISABLE ALL-RIGHTS-IMPLIES)))

(DISABLE ALL-RIGHTS-IMPLIES)
(DISABLE ALL-RIGHTS-REC-UNCHANGED)

(PROVE-LEMMA PROPER-FORKS-REC-INV (REWRITE)
    (IMPLIES (LESSP 1 N)
             (UNLESS '(PROPER-FORKS-REC STATE (QUOTE ,N)
                                              (QUOTE ,N))
               '(FALSE)
               (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE (REWRITE HELP-PROVE-UNLESS)
         (REWRITE PROPER-FORKS-REC-UNLESS-SUFFICIENT) PROVE)))

(PROVE-LEMMA PROPER-FORKS-INV (REWRITE)
             (IMPLIES (LESSP 1 N)
                      (UNLESS '(PROPER-FORKS STATE (QUOTE ,N))
                              '(FALSE)
                              (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE UNLESS-EQUAL-P
                  (($Q (LIST 'PROPER-FORKS-REC 'STATE (LIST 'QUOTE N)
                             (LIST 'QUOTE N)))))
         (REWRITE PROPER-FORKS-REC-INV) PROVE)))

(PROVE-LEMMA PROPER-PHILS-INV (REWRITE)
             (IMPLIES (LESSP 1 N)
                      (UNLESS '(PROPER-PHILS STATE (QUOTE ,N))
                              '(FALSE)
                              (PHIL-PRG N))))


(PROVE-LEMMA ALL-LEFTS-INV (REWRITE)
    (IMPLIES (LESSP 1 N)
             (UNLESS '(ALL-LEFTS STATE (QUOTE ,N))
               '(FALSE)
               (PHIL-PRG N))))

(PROVE-LEMMA ALL-RIGHTS-INV (REWRITE)
    (IMPLIES (LESSP 1 N)
             (UNLESS '(ALL-RIGHTS STATE (QUOTE ,N))
                     '(FALSE)
                     (PHIL-PRG N))))

(PROVE-LEMMA PHIL-PRG-INVARIANT-1 (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (INITIAL-CONDITION
                            '(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N)))
                      (AND (INVARIANT '(PROPER-PHILS STATE (QUOTE ,N))
                                      (PHIL-PRG N))
                           (INVARIANT '(PROPER-FORKS STATE (QUOTE ,N))
                                      (PHIL-PRG N))
                           (INVARIANT '(AND (PROPER-PHILS STATE
                                                          (QUOTE ,N))
                                            (PROPER-FORKS STATE
                                                          (QUOTE ,N)))
                                      (PHIL-PRG N))))
```

```
    ((INSTRUCTIONS (DISABLE EVAL) SPLIT (PROVE (DISABLE EVAL))
         (REWRITE UNLESS-PROVES-INVARIANT
                  (($IC (LIST 'AND
                              (LIST 'PROPER-PHILS 'STATE
                                    (LIST 'QUOTE N))
                              (LIST 'PROPER-FORKS 'STATE
                                    (LIST 'QUOTE N))))))
         (REWRITE PROPER-FORKS-INV) (PROVE (DISABLE EVAL))
         (PROVE (DISABLE EVAL))
         (REWRITE INVARIANT-CONSEQUENCE
                  (($P (LIST 'PROPER-PHILS 'STATE (LIST 'QUOTE N)))))
         (PROVE (DISABLE EVAL)) (BASH (DISABLE EVAL)) PROMOTE
         (REWRITE INVARIANT-IMPLIES)
         (REWRITE UNLESS-PROVES-INVARIANT
                  (($IC (LIST 'AND
                              (LIST 'PROPER-PHILS 'STATE
                                    (LIST 'QUOTE N))
                              (LIST 'PROPER-FORKS 'STATE
                                    (LIST 'QUOTE N))))))
         (PROVE (DISABLE EVAL)) (PROVE (DISABLE EVAL))
         (PROVE (DISABLE EVAL)))))

(PROVE-LEMMA PROPER-PHIL-INVARIANT (REWRITE)
  (IMPLIES (AND (LESSP INDEX N)
                (NUMBERP INDEX)
                (LESSP 1 N)
                (INITIAL-CONDITION
                 `(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N)))
           (INVARIANT `(PROPER-PHIL STATE (QUOTE ,INDEX)
                                    (QUOTE ,(ADD1-MOD N INDEX)))
                      (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE INVARIANT-CONSEQUENCE
                  (($P (LIST 'PROPER-PHILS 'STATE (LIST 'QUOTE N)))))
         PROVE (PROVE (DISABLE PROPER-PHIL)))))

(DEFN PROPER-FORK (STATE INDEX LEFT)
  (OR (FREE STATE INDEX)
      (EQUAL (FORK STATE INDEX) INDEX)
      (EQUAL (FORK STATE INDEX) LEFT)))

(PROVE-LEMMA PROPER-FORK-INVARIANT (REWRITE)
  (IMPLIES (AND (LESSP INDEX N)
                (NUMBERP INDEX)
                (LESSP 1 N)
                (INITIAL-CONDITION
                 `(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N)))
           (INVARIANT `(PROPER-FORK STATE (QUOTE ,INDEX)
                                    (QUOTE ,(SUB1-MOD N INDEX)))
                      (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE INVARIANT-CONSEQUENCE
                  (($P (LIST 'PROPER-FORKS 'STATE (LIST 'QUOTE N)))))
         PROVE (DROP 4)
         (USE-LEMMA PROPER-FORKS-IMPLIES
             ((N N) (I INDEX)
              (STATE (S (PHIL-PRG N)
                        (II (LIST 'PROPER-FORK 'STATE
                                  (LIST 'QUOTE INDEX)
                                  (LIST 'QUOTE (SUB1-MOD N INDEX)))
                            (PHIL-PRG N))))))
         PROVE)))


(PROVE-LEMMA HUNGRY-UNLESS-OWNS-LEFT (REWRITE)
```

```
              (IMPLIES (AND (LESSP 1 N)
                            (LESSP INDEX N)
                            (NUMBERP INDEX))
                       (UNLESS '(HUNGRY STATE (QUOTE ,INDEX))
                               '(OWNS-LEFT STATE (QUOTE ,INDEX))
                               (PHIL-PRG N)))
              ((INSTRUCTIONS PROMOTE
                             (REWRITE HELP-PROVE-UNLESS)
                             (GENERALIZE (((EU (LIST 'HUNGRY
                                                     'STATE
                                                     (LIST 'QUOTE INDEX))
                                               (PHIL-PRG N)
                                               (LIST 'OWNS-LEFT
                                                     'STATE
                                                     (LIST 'QUOTE INDEX)))
                                           STATEMENT)))
                             (CLAIM (EQUAL (CADR STATEMENT) INDEX)
                                    0)
                             PROVE PROVE)))

(PROVE-LEMMA HUNGRY-UNLESS-OWNS-RIGHT (REWRITE)
   (IMPLIES (AND (LESSP 1 N)
                 (LESSP INDEX N)
                 (NUMBERP INDEX))
            (UNLESS '(HUNGRY STATE (QUOTE ,INDEX))
                    '(OWNS-RIGHT STATE (QUOTE ,INDEX) (QUOTE ,N))
                    (PHIL-PRG N)))
              ((INSTRUCTIONS PROMOTE
                             (REWRITE HELP-PROVE-UNLESS)
                             (GENERALIZE (((EU (LIST 'HUNGRY
                                                     'STATE
                                                     (LIST 'QUOTE INDEX))
                                               (PHIL-PRG N)
                                               (LIST 'OWNS-RIGHT
                                                     'STATE
                                                     (LIST 'QUOTE INDEX)
                                                     (LIST 'QUOTE N)))
                                           STATEMENT)))
                             (CLAIM (EQUAL (CADR STATEMENT) INDEX)
                                    0)
                             PROVE PROVE)))

(PROVE-LEMMA HUNGRY-LEFT-FREE-E-ENSURES-OWNS-LEFT (REWRITE)
    (IMPLIES (AND (LESSP 1 N)
                  (LESSP INDEX N)
                  (NUMBERP INDEX))
             (E-ENSURES '(HUNGRY STATE (QUOTE ,INDEX))
                        '(OWNS-LEFT STATE (QUOTE ,INDEX))
                        '(AND (HUNGRY STATE (QUOTE ,INDEX))
                              (FREE STATE (QUOTE ,INDEX)))
                        (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE HELP-PROVE-E-ENSURES
                  (($E (LIST 'HUNGRY-LEFT INDEX))
                   ($NEW (UPDATE-ASSOC (CONS 'F INDEX) INDEX
                              (OLDEEE-1
                                  (LIST 'AND
                                        (LIST 'HUNGRY 'STATE
                                         (LIST 'QUOTE INDEX))
                                        (LIST 'FREE 'STATE
                                         (LIST 'QUOTE INDEX)))
                                  (LIST 'HUNGRY-LEFT INDEX))))))
         PROVE PROVE PROVE PROVE)))

(PROVE-LEMMA HUNGRY-RIGHT-FREE-E-ENSURES-OWNS-RIGHT (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP INDEX N)
                (NUMBERP INDEX))
           (E-ENSURES '(HUNGRY STATE (QUOTE ,INDEX))
```

```
                        '(OWNS-RIGHT STATE (QUOTE ,INDEX) (QUOTE ,N))
                        '(AND (HUNGRY STATE (QUOTE ,INDEX))
                              (FREE STATE
                                    (QUOTE ,(ADD1-MOD N INDEX))))
                        (PHIL-PRG N)))
  ((INSTRUCTIONS
    PROMOTE
    (REWRITE HELP-PROVE-E-ENSURES
             (($E (LIST 'HUNGRY-RIGHT INDEX N))
              ($NEW (UPDATE-ASSOC (CONS 'F (ADD1-MOD N INDEX))
                                  INDEX
                                  (OLDEEE-1
                                   (LIST 'AND
                                         (LIST 'HUNGRY 'STATE
                                               (LIST 'QUOTE INDEX))
                                         (LIST 'FREE 'STATE
                                               (LIST 'QUOTE
                                                     (ADD1-MOD N INDEX))))
                                   (LIST 'HUNGRY-RIGHT INDEX N))))))
    PROVE PROVE PROVE PROVE)))


(PROVE-LEMMA HUNGRY-UNLESS-EATING (REWRITE)
    (IMPLIES (AND (LESSP 1 N)
                  (LESSP INDEX N)
                  (NUMBERP INDEX))
             (UNLESS '(HUNGRY STATE (QUOTE ,INDEX))
                     '(EATING STATE (QUOTE ,INDEX))
               (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE (REWRITE HELP-PROVE-UNLESS)
         (GENERALIZE
             (((EU (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX))
                   (PHIL-PRG N)
                   (LIST 'EATING 'STATE (LIST 'QUOTE INDEX)))
               STATEMENT)))
         (CLAIM (EQUAL (CADR STATEMENT) INDEX) 0) PROVE PROVE)))

(PROVE-LEMMA OWNS-LEFT-UNLESS-SUFFICIENT (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX))
                      (UNLESS-SUFFICIENT
                       STATEMENT
                       (PHIL-PRG N)
                       OLD NEW
                       '(AND (PROPER-PHILS STATE (QUOTE ,N))
                             (OWNS-LEFT STATE (QUOTE ,INDEX)))
                       '(EATING STATE (QUOTE ,INDEX))))
    ((INSTRUCTIONS PROMOTE X-DUMB PROMOTE
         (= * (OR (OWNS-LEFT NEW INDEX) (EATING NEW INDEX)) 0)
         (CLAIM (PROPER-PHIL OLD (SUB1-MOD N INDEX) INDEX) 0)
         (CLAIM (OWNS-LEFT OLD INDEX) 0) (DROP 6)
         (CLAIM (EQUAL (CADR STATEMENT) (SUB1-MOD N INDEX)) 0) PROVE
         (DROP 7) (CLAIM (EQUAL (CADR STATEMENT) INDEX) 0) PROVE
         (DROP 6) PROVE (CONTRADICT 9) (DROP 1 2 3 4 5 7 8 9)
         (PROVE (DISABLE OWNS-LEFT)) (CONTRADICT 8) (DROP 4 5 7 8)
         (DEMOTE 4) (DIVE 1)
         (= * (AND (PROPER-PHILS OLD N) (OWNS-LEFT OLD INDEX))
            ((DISABLE OWNS-LEFT)))
         TOP PROMOTE
         (REWRITE PROPER-PHILS-IMPLIES-PROPER-PHIL (($N N)))
         (DROP 2 3 4 5) PROVE (DROP 1 4 5) PROVE (DROP 2 3 7)
         (PROVE (DISABLE N MEMBER-PHIL-PRG OWNS-LEFT EATING
                         PROPER-PHILS)))))

(PROVE-LEMMA OWNS-RIGHT-UNLESS-SUFFICIENT (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX))
```

```
                    (UNLESS-SUFFICIENT
                     STATEMENT
                     (PHIL-PRG N)
                     OLD NEW
                     '(AND (PROPER-PHILS STATE (QUOTE ,N))
                           (OWNS-RIGHT STATE (QUOTE ,INDEX) (QUOTE ,N)))
                     '(EATING STATE (QUOTE ,INDEX))))
  ((INSTRUCTIONS PROMOTE X-DUMB PROMOTE
       (= * (OR (OWNS-RIGHT NEW INDEX N) (EATING NEW INDEX)) 0)
       (CLAIM (PROPER-PHIL OLD (ADD1-MOD N INDEX)
                   (ADD1-MOD N (ADD1-MOD N INDEX)))
               0)
       (CLAIM (OWNS-RIGHT OLD INDEX N) 0) (DROP 6)
       (CLAIM (EQUAL (CADR STATEMENT) (ADD1-MOD N INDEX)) 0) PROVE
       (DROP 7) (CLAIM (EQUAL (CADR STATEMENT) INDEX) 0) PROVE
       (DROP 6) PROVE (CONTRADICT 9) (DROP 1 2 3 4 5 7 8 9)
       (PROVE (DISABLE OWNS-RIGHT)) (CONTRADICT 8) (DROP 4 5 7 8)
       (REWRITE PROPER-PHILS-IMPLIES-PROPER-PHIL (($INDEX N) ($N N)))
       (PROVE (DISABLE PROPER-PHIL)) (DROP 1 3 4) PROVE (DROP 7)
       (PROVE (DISABLE N OWNS-RIGHT EATING MEMBER-PHIL-PRG)))))

(PROVE-LEMMA OWNS-RIGHT-UNLESS-EATING (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP INDEX N)
                (NUMBERP INDEX))
           (UNLESS '(AND (PROPER-PHILS STATE (QUOTE ,N))
                         (OWNS-RIGHT STATE (QUOTE ,INDEX)
                                     (QUOTE ,N)))
                   '(EATING STATE (QUOTE ,INDEX))
                   (PHIL-PRG N)))
             ((DISABLE PHIL-PRG UNLESS-SUFFICIENT)))


(PROVE-LEMMA OWNS-LEFT-UNLESS-EATING (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX))
                      (UNLESS '(AND (PROPER-PHILS STATE (QUOTE ,N))
                                    (OWNS-LEFT STATE (QUOTE ,INDEX)))
                              '(EATING STATE (QUOTE ,INDEX))
                              (PHIL-PRG N)))
             ((DISABLE UNLESS-SUFFICIENT PHIL-PRG)))

(PROVE-LEMMA OWNS-BOTH-UNLESS-EATING (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX))
                      (UNLESS '(AND (OWNS-LEFT STATE (QUOTE ,INDEX))
                                    (AND (OWNS-RIGHT STATE (QUOTE ,INDEX)
                                                     (QUOTE ,N))
                                         (PROPER-PHILS STATE (QUOTE ,N))))
                              '(EATING STATE (QUOTE ,INDEX))
                              (PHIL-PRG N)))
  ((INSTRUCTIONS PROMOTE
       (REWRITE UNLESS-CONJUNCTION
                (($P-1 (LIST 'AND
                             (LIST 'PROPER-PHILS 'STATE
                                   (LIST 'QUOTE N))
                             (LIST 'OWNS-LEFT 'STATE
                                   (LIST 'QUOTE INDEX))))
                 ($P-2 (LIST 'AND
                             (LIST 'PROPER-PHILS 'STATE
                                   (LIST 'QUOTE N))
                             (LIST 'OWNS-RIGHT 'STATE
                                   (LIST 'QUOTE INDEX)
                                   (LIST 'QUOTE N))))))
       (REWRITE OWNS-LEFT-UNLESS-EATING)
       (REWRITE OWNS-RIGHT-UNLESS-EATING) (PROVE (DISABLE EVAL)))))
```

```
(PROVE-LEMMA OWNS-BOTH-E-ENSURES-EATING (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP INDEX N)
                (NUMBERP INDEX))
           (E-ENSURES `(AND (OWNS-LEFT STATE (QUOTE ,INDEX))
                            (AND (OWNS-RIGHT STATE (QUOTE ,INDEX)
                                             (QUOTE ,N))
                                 (PROPER-PHILS STATE (QUOTE ,N))))
                      `(EATING STATE (QUOTE ,INDEX))
                      `(TRUE)
                      (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE PROVE-E-ENSURES (($E (LIST 'HUNGRY-BOTH INDEX N))))
         PROVE
         (REWRITE PROVE-ENABLING-CONDITION
                  (($NEW (IF (AND (HUNGRY (OLDC-1 '(TRUE)
                                           (LIST 'HUNGRY-BOTH INDEX N))
                                          INDEX)
                                  (OWNS-LEFT
                                      (OLDC-1 '(TRUE)
                                       (LIST 'HUNGRY-BOTH INDEX N))
                                      INDEX)
                                  (OWNS-RIGHT
                                      (OLDC-1 '(TRUE)
                                       (LIST 'HUNGRY-BOTH INDEX N))
                                      INDEX N))
                             (UPDATE-ASSOC (CONS 'S INDEX) 'EATING
                                 (OLDC-1 '(TRUE)
                                         (LIST 'HUNGRY-BOTH INDEX N)))
                             (OLDC-1 '(TRUE)
                                     (LIST 'HUNGRY-BOTH INDEX N))))))
         PROVE PROVE PROVE
         (CLAIM (HUNGRY (OLDEE (LIST 'HUNGRY-BOTH INDEX N)
                               (LIST 'AND
                                     (LIST 'OWNS-LEFT 'STATE
                                      (LIST 'QUOTE INDEX))
                                     (LIST 'AND
                                      (LIST 'OWNS-RIGHT 'STATE
                                       (LIST 'QUOTE INDEX)
                                       (LIST 'QUOTE N))
                                      (LIST 'PROPER-PHILS 'STATE
                                       (LIST 'QUOTE N))))
                               (LIST 'EATING 'STATE
                                     (LIST 'QUOTE INDEX)))
                        INDEX)
                0)
         (PROVE (DISABLE HUNGRY EATING OWNS-LEFT OWNS-RIGHT))
         (CONTRADICT 4)
         (USE-LEMMA PROPER-PHILS-IMPLIES-PROPER-PHIL
             ((PHIL INDEX) (N N)
              (STATE (OLDEE (LIST 'HUNGRY-BOTH INDEX N)
                            (LIST 'AND
                                  (LIST 'OWNS-LEFT 'STATE
                                        (LIST 'QUOTE INDEX))
                                  (LIST 'AND
                                        (LIST 'OWNS-RIGHT 'STATE
                                         (LIST 'QUOTE INDEX)
                                         (LIST 'QUOTE N))
                                        (LIST 'PROPER-PHILS 'STATE
                                         (LIST 'QUOTE N))))
                            (LIST 'EATING 'STATE (LIST 'QUOTE INDEX))))
              (RIGHT (ADD1-MOD N INDEX))))
         (PROVE (DISABLE ADD1-MOD PROPER-PHILS-IMPLIES-PROPER-PHIL)))))

(PROVE-LEMMA OWNS-BOTH-LEADS-TO-EATING (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX)
                           (INITIAL-CONDITION
```

```
                              '(AND (PROPER-PHILS STATE (QUOTE ,N))
                                    (PROPER-FORKS STATE (QUOTE ,N)))
                              (PHIL-PRG N)))
                        (LEADS-TO '(AND (OWNS-LEFT STATE (QUOTE ,INDEX))
                                        (OWNS-RIGHT STATE (QUOTE ,INDEX)
                                                    (QUOTE ,N)))
                                  '(EATING STATE (QUOTE ,INDEX))
                                  (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE WEAK-FAIRNESS-GENERAL
                  (($Q-1 (LIST 'EATING 'STATE (LIST 'QUOTE INDEX)))
                   ($Q-2 (LIST 'EATING 'STATE (LIST 'QUOTE INDEX)))
                   ($P-1 (LIST 'AND
                               (LIST 'OWNS-LEFT 'STATE
                                     (LIST 'QUOTE INDEX))
                               (LIST 'AND
                                     (LIST 'OWNS-RIGHT 'STATE
                                      (LIST 'QUOTE INDEX)
                                      (LIST 'QUOTE N))
                                     (LIST 'PROPER-PHILS 'STATE
                                      (LIST 'QUOTE N)))))
                   ($P-2 (LIST 'AND
                               (LIST 'OWNS-LEFT 'STATE
                                     (LIST 'QUOTE INDEX))
                               (LIST 'AND
                                     (LIST 'OWNS-RIGHT 'STATE
                                      (LIST 'QUOTE INDEX)
                                      (LIST 'QUOTE N))
                                     (LIST 'PROPER-PHILS 'STATE
                                      (LIST 'QUOTE N)))))
                   ($C '(TRUE))))
         (REWRITE OWNS-BOTH-UNLESS-EATING)
         (REWRITE OWNS-BOTH-E-ENSURES-EATING) (BASH (DISABLE EVAL))
         (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL))
         (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)))))

(PROVE-LEMMA EATING-UNLESS-FREE (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX))
                      (UNLESS '(EATING STATE (QUOTE ,INDEX))
                              '(AND (FREE STATE (QUOTE ,INDEX))
                                    (FREE STATE
                                          (QUOTE ,(ADD1-MOD N INDEX))))
                                (PHIL-PRG N)))
  ((INSTRUCTIONS
    PROMOTE
    (REWRITE HELP-PROVE-UNLESS)
    (GENERALIZE (((EU (LIST 'EATING
                            'STATE
                            (LIST 'QUOTE INDEX))
                      (PHIL-PRG N)
                      (LIST 'AND
                            (LIST 'FREE
                                  'STATE
                                  (LIST 'QUOTE INDEX))
                            (LIST 'FREE
                                  'STATE
                                  (LIST 'QUOTE (ADD1-MOD N INDEX)))))
                  STATEMENT)))
    (CLAIM (EQUAL (CADR STATEMENT) INDEX)
           0)
    PROVE PROVE)))

(PROVE-LEMMA EATING-E-ENSURES-FREE (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX))
                      (E-ENSURES '(EATING STATE (QUOTE ,INDEX))
```

```
                                ‘(AND (FREE STATE (QUOTE ,INDEX))
                                      (FREE STATE
                                            (QUOTE ,(ADD1-MOD N INDEX))))
                                ’(TRUE)
                                (PHIL-PRG N)))
 ((INSTRUCTIONS
   PROMOTE
   (REWRITE PROVE-E-ENSURES
            (($E (LIST ’EATING-TO INDEX N))))
   PROVE
   (REWRITE PROVE-ENABLING-CONDITION
            (($NEW
              (IF
               (EATING (OLDC-1 ’(TRUE)
                               (LIST ’EATING-TO INDEX N))
                       INDEX)
               (UPDATE-ASSOC
                (CONS ’S INDEX)
                ’THINKING
                (UPDATE-ASSOC
                 (CONS ’F INDEX)
                 ’FREE
                 (UPDATE-ASSOC (CONS ’F (ADD1-MOD N INDEX))
                               ’FREE
                               (OLDC-1 ’(TRUE)
                                       (LIST ’EATING-TO INDEX N)))))
               (OLDC-1 ’(TRUE)
                       (LIST ’EATING-TO INDEX N))))))
   PROVE PROVE PROVE
   (PROVE (DISABLE EATING THINKING FREE ADD1-MOD)))))


(PROVE-LEMMA EATING-LEADS-TO-FREE (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX)
                           (INITIAL-CONDITION
                            ‘(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N)))
                      (LEADS-TO ‘(EATING STATE (QUOTE ,INDEX))
                                ‘(AND (FREE STATE (QUOTE ,INDEX))
                                      (FREE STATE
                                            (QUOTE ,(ADD1-MOD N INDEX))))
                                (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE WEAK-FAIRNESS-GENERAL
                  (($Q-1 (LIST ’AND
                               (LIST ’FREE ’STATE (LIST ’QUOTE INDEX))
                               (LIST ’FREE ’STATE
                                     (LIST ’QUOTE (ADD1-MOD N INDEX)))))
                   ($Q-2 (LIST ’AND
                               (LIST ’FREE ’STATE (LIST ’QUOTE INDEX))
                               (LIST ’FREE ’STATE
                                     (LIST ’QUOTE (ADD1-MOD N INDEX)))))
                   ($P-1 (LIST ’EATING ’STATE (LIST ’QUOTE INDEX)))
                   ($P-2 (LIST ’EATING ’STATE (LIST ’QUOTE INDEX)))
                   ($C ’(TRUE))))
         (REWRITE EATING-UNLESS-FREE) (REWRITE EATING-E-ENSURES-FREE)
         (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL))
         (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL))
         (BASH (DISABLE EVAL)))))

(PROVE-LEMMA LEADS-TO-EXPANDED-RIGHT-IMPLIES (REWRITE)
   (IMPLIES (AND (LESSP 1 N)
                 (LESSP INDEX N)
                 (NUMBERP INDEX)
                 (INITIAL-CONDITION
                  ‘(AND (PROPER-PHILS STATE (QUOTE ,N))
```

```
                   (PROPER-FORKS STATE (QUOTE ,N)))
             (PHIL-PRG N))
            (LEADS-TO ‘(TRUE)
                       ‘(OR (OWNS-RIGHT
                             STATE (QUOTE ,(SUB1-MOD N INDEX))
                             (QUOTE ,N))
                            (OR (FREE STATE (QUOTE ,INDEX))
                                (OR (EATING STATE
                                            (QUOTE ,INDEX))
                                    (THINKING
                                     STATE
                                     (QUOTE ,INDEX)))))
                       (PHIL-PRG N)))
       (LEADS-TO ‘(TRUE)
                  ‘(OR (FREE STATE (QUOTE ,INDEX))
                       (OWNS-RIGHT STATE
                                   (QUOTE ,(SUB1-MOD N INDEX))
                                   (QUOTE ,N)))
                  (PHIL-PRG N)))
((INSTRUCTIONS PROMOTE
     (REWRITE CANCELLATION-LEADS-TO-GENERAL
              (($P-1 ’(TRUE))
               ($B (LIST ’EATING ’STATE (LIST ’QUOTE INDEX)))
               ($R-1 (LIST ’FREE ’STATE (LIST ’QUOTE INDEX)))
               ($D (LIST ’OR
                         (LIST ’OWNS-RIGHT ’STATE
                               (LIST ’QUOTE (SUB1-MOD N INDEX))
                               (LIST ’QUOTE N))
                         (LIST ’OR
                               (LIST ’FREE ’STATE
                                (LIST ’QUOTE INDEX))
                               (LIST ’OR
                                (LIST ’EATING ’STATE
                                 (LIST ’QUOTE INDEX))
                                (LIST ’THINKING ’STATE
                                 (LIST ’QUOTE INDEX))))))))
     (REWRITE LEADS-TO-WEAKEN-RIGHT
              (($Q (LIST ’AND
                         (LIST ’FREE ’STATE (LIST ’QUOTE INDEX))
                         (LIST ’FREE ’STATE
                               (LIST ’QUOTE (ADD1-MOD N INDEX)))))))
     (PROVE (DISABLE EVAL)) (REWRITE EATING-LEADS-TO-FREE)
     (CLAIM (EVAL (LIST ’PROPER-PHIL ’STATE (LIST ’QUOTE INDEX)
                        (LIST ’QUOTE (ADD1-MOD N INDEX)))
                  (S (PHIL-PRG N)
                     (JLEADS (ILEADS ’(TRUE) (PHIL-PRG N)
                                     (LIST ’OR
                                      (LIST ’FREE ’STATE
                                       (LIST ’QUOTE INDEX))
                                      (LIST ’OWNS-RIGHT ’STATE
                                       (LIST ’QUOTE
                                        (SUB1-MOD N INDEX))
                                       (LIST ’QUOTE N))))
                             (PHIL-PRG N)
                             (LIST ’OR
                                   (LIST ’OWNS-RIGHT ’STATE
                                    (LIST ’QUOTE
                                     (SUB1-MOD N INDEX))
                                    (LIST ’QUOTE N))
                                   (LIST ’OR
                                    (LIST ’FREE ’STATE
                                     (LIST ’QUOTE INDEX))
                                    (LIST ’OR
                                     (LIST ’EATING ’STATE
                                      (LIST ’QUOTE INDEX))
                                     (LIST ’THINKING ’STATE
                                      (LIST ’QUOTE INDEX)))))))
            ((DISABLE EVAL ADD1-MOD)))
     (CLAIM (EVAL (LIST ’PROPER-FORK ’STATE (LIST ’QUOTE INDEX)
```

```
                              (LIST 'QUOTE (SUB1-MOD N INDEX)))
                        (S (PHIL-PRG N)
                           (JLEADS (ILEADS '(TRUE) (PHIL-PRG N)
                                           (LIST 'OR
                                            (LIST 'FREE 'STATE
                                             (LIST 'QUOTE INDEX))
                                            (LIST 'OWNS-RIGHT 'STATE
                                             (LIST 'QUOTE
                                              (SUB1-MOD N INDEX))
                                             (LIST 'QUOTE N))))
                                   (PHIL-PRG N)
                                   (LIST 'OR
                                         (LIST 'OWNS-RIGHT 'STATE
                                          (LIST 'QUOTE
                                           (SUB1-MOD N INDEX))
                                          (LIST 'QUOTE N))
                                         (LIST 'OR
                                          (LIST 'FREE 'STATE
                                           (LIST 'QUOTE INDEX))
                                          (LIST 'OR
                                           (LIST 'EATING 'STATE
                                            (LIST 'QUOTE INDEX))
                                           (LIST 'THINKING 'STATE
                                            (LIST 'QUOTE INDEX)))))))
                  ((DISABLE EVAL SUB1-MOD)))
         (GENERALIZE
             (((S (PHIL-PRG N)
                  (JLEADS (ILEADS '(TRUE) (PHIL-PRG N)
                                  (LIST 'OR
                                        (LIST 'FREE 'STATE
                                         (LIST 'QUOTE INDEX))
                                        (LIST 'OWNS-RIGHT 'STATE
                                         (LIST 'QUOTE
                                          (SUB1-MOD N INDEX))
                                         (LIST 'QUOTE N))))
                          (PHIL-PRG N)
                          (LIST 'OR
                                (LIST 'OWNS-RIGHT 'STATE
                                      (LIST 'QUOTE (SUB1-MOD N INDEX))
                                      (LIST 'QUOTE N))
                                (LIST 'OR
                                      (LIST 'FREE 'STATE
                                       (LIST 'QUOTE INDEX))
                                      (LIST 'OR
                                       (LIST 'EATING 'STATE
                                        (LIST 'QUOTE INDEX))
                                       (LIST 'THINKING 'STATE
                                        (LIST 'QUOTE INDEX)))))))
               STATE)))
         (DROP 4 5) BASH (PROVE (DISABLE EVAL)) S)))

(PROVE-LEMMA LEADS-TO-EXPANDED-LEFT-IMPLIES (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP INDEX N)
                (NUMBERP INDEX)
                (INITIAL-CONDITION
                 `(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N))
                (LEADS-TO `(TRUE)
                          `(OR (OWNS-LEFT
                                STATE
                                (QUOTE ,(ADD1-MOD N INDEX)))
                               (OR (FREE
                                    STATE
                                    (QUOTE ,(ADD1-MOD N INDEX)))
                                   (OR (EATING STATE (QUOTE ,INDEX))
                                       (THINKING STATE
                                                 (QUOTE ,INDEX)))))
```

```
                              (PHIL-PRG N)))
           (LEADS-TO '(TRUE)
                      '(OR (FREE STATE (QUOTE ,(ADD1-MOD N INDEX)))
                           (OWNS-LEFT STATE
                                      (QUOTE ,(ADD1-MOD N INDEX))))
                      (PHIL-PRG N)))
((INSTRUCTIONS PROMOTE
  (REWRITE CANCELLATION-LEADS-TO-GENERAL
           (($P-1 '(TRUE))
            ($B (LIST 'EATING
                      'STATE
                      (LIST 'QUOTE INDEX)))
            ($R-1 (LIST 'AND
                        (LIST 'FREE
                              'STATE
                              (LIST 'QUOTE INDEX))
                        (LIST 'FREE
                              'STATE
                              (LIST 'QUOTE (ADD1-MOD N INDEX)))))
            ($D (LIST 'OR
                      (LIST 'OWNS-LEFT
                            'STATE
                            (LIST 'QUOTE (ADD1-MOD N INDEX)))
                      (LIST 'OR
                            (LIST 'FREE
                                  'STATE
                                  (LIST 'QUOTE (ADD1-MOD N INDEX)))
                            (LIST 'OR
                                  (LIST 'EATING
                                        'STATE
                                        (LIST 'QUOTE INDEX))
                                  (LIST 'THINKING
                                        'STATE
                                        (LIST 'QUOTE INDEX))))))))
  (REWRITE EATING-LEADS-TO-FREE)
  (CLAIM
   (EVAL
    (LIST 'PROPER-PHIL
          'STATE
          (LIST 'QUOTE INDEX)
          (LIST 'QUOTE (ADD1-MOD N INDEX)))
    (S
     (PHIL-PRG N)
     (JLEADS (ILEADS '(TRUE)
                     (PHIL-PRG N)
                     (LIST 'OR
                           (LIST 'FREE
                                 'STATE
                                 (LIST 'QUOTE (ADD1-MOD N INDEX)))
                           (LIST 'OWNS-LEFT
                                 'STATE
                                 (LIST 'QUOTE (ADD1-MOD N INDEX)))))
             (PHIL-PRG N)
             (LIST 'OR
                   (LIST 'OWNS-LEFT
                         'STATE
                         (LIST 'QUOTE (ADD1-MOD N INDEX)))
                   (LIST 'OR
                         (LIST 'FREE
                               'STATE
                               (LIST 'QUOTE (ADD1-MOD N INDEX)))
                         (LIST 'OR
                               (LIST 'EATING
                                     'STATE
                                     (LIST 'QUOTE INDEX))
                               (LIST 'THINKING
                                     'STATE
                                     (LIST 'QUOTE INDEX))))))))
   ((DISABLE EVAL ADD1-MOD))
```

```
(CLAIM
 (EVAL
  (LIST 'PROPER-FORK
        'STATE
        (LIST 'QUOTE (ADD1-MOD N INDEX))
        (LIST 'QUOTE
              (SUB1-MOD N (ADD1-MOD N INDEX))))
  (S
   (PHIL-PRG N)
   (JLEADS (ILEADS '(TRUE)
                   (PHIL-PRG N)
                   (LIST 'OR
                         (LIST 'FREE
                               'STATE
                               (LIST 'QUOTE (ADD1-MOD N INDEX)))
                         (LIST 'OWNS-LEFT
                               'STATE
                               (LIST 'QUOTE (ADD1-MOD N INDEX)))))
           (PHIL-PRG N)
           (LIST 'OR
                 (LIST 'OWNS-LEFT
                       'STATE
                       (LIST 'QUOTE (ADD1-MOD N INDEX)))
                 (LIST 'OR
                       (LIST 'FREE
                             'STATE
                             (LIST 'QUOTE (ADD1-MOD N INDEX)))
                       (LIST 'OR
                             (LIST 'EATING
                                   'STATE
                                   (LIST 'QUOTE INDEX))
                             (LIST 'THINKING
                                   'STATE
                                   (LIST 'QUOTE INDEX))))))))
 ((DISABLE EVAL SUB1-MOD)))
(GENERALIZE
 (((S
    (PHIL-PRG N)
    (JLEADS
           (ILEADS '(TRUE)
                   (PHIL-PRG N)
                   (LIST 'OR
                         (LIST 'FREE
                               'STATE
                               (LIST 'QUOTE (ADD1-MOD N INDEX)))
                         (LIST 'OWNS-LEFT
                               'STATE
                               (LIST 'QUOTE (ADD1-MOD N INDEX)))))
           (PHIL-PRG N)
           (LIST 'OR
                 (LIST 'OWNS-LEFT
                       'STATE
                       (LIST 'QUOTE (ADD1-MOD N INDEX)))
                 (LIST 'OR
                       (LIST 'FREE
                             'STATE
                             (LIST 'QUOTE (ADD1-MOD N INDEX)))
                       (LIST 'OR
                             (LIST 'EATING
                                   'STATE
                                   (LIST 'QUOTE INDEX))
                             (LIST 'THINKING
                                   'STATE
                                   (LIST 'QUOTE INDEX)))))))
   STATE)))
(DROP 4 5)
BASH
(PROVE (DISABLE EVAL))
S)))
```

```
(PROVE-LEMMA TRUE-LEADS-TO-LEFT-FREE-IMPLIES (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP INDEX N)
                (NUMBERP INDEX)
                (STRONGLY-FAIR (PHIL-PRG N))
                (INITIAL-CONDITION
                 '(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N))
                (LEADS-TO '(TRUE)
                          '(OR (FREE STATE (QUOTE ,INDEX))
                               (OWNS-LEFT STATE (QUOTE ,INDEX)))
                          (PHIL-PRG N)))
           (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
                     '(OWNS-LEFT STATE (QUOTE ,INDEX))
                     (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE STRONG-FAIRNESS
                  (($C (LIST 'AND
                             (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX))
                             (LIST 'FREE 'STATE (LIST 'QUOTE INDEX))))))
         (REWRITE HUNGRY-UNLESS-OWNS-LEFT)
         (REWRITE HUNGRY-LEFT-FREE-E-ENSURES-OWNS-LEFT)
         (REWRITE PSP-GENERAL
                  (($P '(TRUE))
                   ($Q (LIST 'OR
                             (LIST 'FREE 'STATE (LIST 'QUOTE INDEX))
                             (LIST 'OWNS-LEFT 'STATE
                                   (LIST 'QUOTE INDEX))))
                   ($R (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX)))
                   ($B (LIST 'OWNS-LEFT 'STATE (LIST 'QUOTE INDEX)))))
         (REWRITE HUNGRY-UNLESS-OWNS-LEFT) (PROVE (DISABLE EVAL))
         (PROVE (DISABLE EVAL)) PROVE)))

(PROVE-LEMMA TRUE-LEADS-TO-RIGHT-FREE-IMPLIES (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP INDEX N)
                (NUMBERP INDEX)
                (STRONGLY-FAIR (PHIL-PRG N))
                (INITIAL-CONDITION
                 '(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N))
                (LEADS-TO '(TRUE)
                          '(OR (FREE STATE
                                     (QUOTE ,(ADD1-MOD N INDEX)))
                               (OWNS-RIGHT STATE (QUOTE ,INDEX)
                                           (QUOTE ,N)))
                          (PHIL-PRG N)))
           (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
                     '(OWNS-RIGHT STATE (QUOTE ,INDEX)
                                  (QUOTE ,N))
                     (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE STRONG-FAIRNESS
                  (($C (LIST 'AND
                             (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX))
                             (LIST 'FREE 'STATE
                                   (LIST 'QUOTE (ADD1-MOD N INDEX)))))))
         (REWRITE HUNGRY-UNLESS-OWNS-RIGHT)
         (REWRITE HUNGRY-RIGHT-FREE-E-ENSURES-OWNS-RIGHT)
         (REWRITE PSP-GENERAL
                  (($P '(TRUE))
                   ($Q (LIST 'OR
                             (LIST 'FREE 'STATE
                                   (LIST 'QUOTE (ADD1-MOD N INDEX)))
                             (LIST 'OWNS-RIGHT 'STATE
                                   (LIST 'QUOTE INDEX)
                                   (LIST 'QUOTE N))))
```

```
                    ($R (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX)))
                    ($B (LIST 'OWNS-RIGHT 'STATE (LIST 'QUOTE INDEX)
                              (LIST 'QUOTE N)))))
          (REWRITE HUNGRY-UNLESS-OWNS-RIGHT) (PROVE (DISABLE EVAL))
          (PROVE (DISABLE EVAL)) PROVE)))

(PROVE-LEMMA HUNGRY-LEADS-TO-OWNS-RIGHT-IMPLIES (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX)
                           (STRONGLY-FAIR (PHIL-PRG N))
                           (INITIAL-CONDITION
                            '(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N))
                           (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
                                     '(OWNS-RIGHT STATE (QUOTE ,INDEX)
                                                  (QUOTE ,N))
                                     (PHIL-PRG N)))
                      (LEADS-TO '(AND (HUNGRY STATE (QUOTE ,INDEX))
                                      (OWNS-LEFT STATE (QUOTE ,INDEX)))
                                '(EATING STATE (QUOTE ,INDEX))
                                (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
          (REWRITE CANCELLATION-LEADS-TO-GENERAL
                   (($P-1 (LIST 'AND
                                (LIST 'HUNGRY 'STATE
                                      (LIST 'QUOTE INDEX))
                                (LIST 'OWNS-LEFT 'STATE
                                      (LIST 'QUOTE INDEX))))
                    ($B (LIST 'AND
                              (LIST 'OWNS-LEFT 'STATE
                                    (LIST 'QUOTE INDEX))
                              (LIST 'OWNS-RIGHT 'STATE
                                    (LIST 'QUOTE INDEX) (LIST 'QUOTE N))))
                    ($R-1 (LIST 'EATING 'STATE (LIST 'QUOTE INDEX)))
                    ($D (LIST 'OR
                              (LIST 'AND
                                    (LIST 'OWNS-LEFT 'STATE
                                     (LIST 'QUOTE INDEX))
                                    (LIST 'OWNS-RIGHT 'STATE
                                     (LIST 'QUOTE INDEX)
                                     (LIST 'QUOTE N)))
                              (LIST 'EATING 'STATE (LIST 'QUOTE INDEX))))))
          (REWRITE PSP-GENERAL
                   (($P (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX)))
                    ($Q (LIST 'OWNS-RIGHT 'STATE (LIST 'QUOTE INDEX)
                              (LIST 'QUOTE N)))
                    ($R (LIST 'AND
                              (LIST 'PROPER-PHILS 'STATE
                                    (LIST 'QUOTE N))
                              (LIST 'OWNS-LEFT 'STATE
                                    (LIST 'QUOTE INDEX))))
                    ($B (LIST 'EATING 'STATE (LIST 'QUOTE INDEX)))))
          (REWRITE OWNS-LEFT-UNLESS-EATING) (BASH (DISABLE EVAL))
          (PROVE (DISABLE EVAL)) PROVE
          (REWRITE OWNS-BOTH-LEADS-TO-EATING) (PROVE (DISABLE EVAL)) S
          S)))

(PROVE-LEMMA HUNGRY-LEADS-TO-OWNS-LEFT-IMPLIES (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX)
                           (STRONGLY-FAIR (PHIL-PRG N))
                           (INITIAL-CONDITION
                            '(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N))
                           (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
```

```
                                        ‘(OWNS-LEFT STATE (QUOTE ,INDEX))
                                        (PHIL-PRG N)))
                         (LEADS-TO ‘(AND (HUNGRY STATE (QUOTE ,INDEX))
                                         (OWNS-RIGHT STATE (QUOTE ,INDEX)
                                                     (QUOTE ,N)))
                                   ‘(EATING STATE (QUOTE ,INDEX))
                                   (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE CANCELLATION-LEADS-TO-GENERAL
                  (($P-1 (LIST ’AND
                               (LIST ’HUNGRY ’STATE
                                     (LIST ’QUOTE INDEX))
                               (LIST ’OWNS-RIGHT ’STATE
                                     (LIST ’QUOTE INDEX)
                                     (LIST ‘QUOTE N))))
                   ($B (LIST ’AND
                             (LIST ’OWNS-LEFT ’STATE
                                   (LIST ’QUOTE INDEX))
                             (LIST ’OWNS-RIGHT ’STATE
                                   (LIST ’QUOTE INDEX) (LIST ’QUOTE N))))
                   ($R-1 (LIST ’EATING ’STATE (LIST ’QUOTE INDEX)))
                   ($D (LIST ’OR
                             (LIST ’AND
                                   (LIST ’OWNS-LEFT ’STATE
                                    (LIST ’QUOTE INDEX))
                                   (LIST ’OWNS-RIGHT ’STATE
                                    (LIST ’QUOTE INDEX)
                                    (LIST ’QUOTE N)))
                             (LIST ’EATING ’STATE (LIST ’QUOTE INDEX))))))
         (REWRITE PSP-GENERAL
                  (($P (LIST ’HUNGRY ’STATE (LIST ’QUOTE INDEX)))
                   ($Q (LIST ’OWNS-LEFT ’STATE (LIST ’QUOTE INDEX)))
                   ($R (LIST ’AND
                             (LIST ’PROPER-PHILS ’STATE
                                   (LIST ’QUOTE N))
                             (LIST ’OWNS-RIGHT ’STATE
                                   (LIST ’QUOTE INDEX)
                                   (LIST ‘QUOTE N))))
                   ($B (LIST ’EATING ’STATE (LIST ’QUOTE INDEX)))))
         (REWRITE OWNS-RIGHT-UNLESS-EATING) (BASH (DISABLE EVAL))
         (PROVE (DISABLE EVAL)) PROVE
         (REWRITE OWNS-BOTH-LEADS-TO-EATING) (PROVE (DISABLE EVAL)) S
         S)))

(DEFN LEFT-CHAIN (I J N STATE)
  (IF (AND (NUMBERP I)
           (NUMBERP J)
           (LESSP I N)
           (LESSP J N))
      (IF (EQUAL I J)
          (AND (HUNGRY STATE I)
               (OWNS-LEFT STATE I))
        (AND (HUNGRY STATE I)
             (OWNS-LEFT STATE I)
             (LEFT-CHAIN (SUB1-MOD N I) J N STATE)))
    F)
  ((LESSP (IF (NOT (LESSP I J))
              (DIFFERENCE (ADD1 I) J)
            (PLUS (ADD1 I) (DIFFERENCE N J))))))

(DEFN RIGHT-CHAIN (I J N STATE)
  (IF (AND (NUMBERP I)
           (NUMBERP J)
           (LESSP I N)
           (LESSP J N))
      (IF (EQUAL I J)
          (AND (HUNGRY STATE I)
               (OWNS-RIGHT STATE I N))
        (AND (HUNGRY STATE I)
```

```
              (OWNS-RIGHT STATE I N)
              (RIGHT-CHAIN (ADD1-MOD N I) J N STATE)))
    F)
  ((LESSP (IF (NOT (LESSP J I))
               (DIFFERENCE (ADD1 J) I)
             (PLUS (ADD1 J) (DIFFERENCE N I))))))

(PROVE-LEMMA BREAK-LEFT-CHAIN (REWRITE)
              (IMPLIES (AND (LESSP I N)
                            (LESSP J N)
                            (NUMBERP I)
                            (NUMBERP J)
                            (LESSP I J))
                       (EQUAL (LEFT-CHAIN I J N STATE)
                              (AND (LEFT-CHAIN I 0 N STATE)
                                   (LEFT-CHAIN (SUB1 N) J N STATE)))))

(DEFN LCC (I J K)
  (IF (AND (NUMBERP I)
           (NUMBERP J)
           (NUMBERP K)
           (LESSP I K)
           (LESSP J K)
           (NOT (EQUAL I J))
           (NOT (ZEROP K)))
      (IF (EQUAL (ADD1 I) K)
          (LCC (SUB1 I) J (SUB1 K))
        (IF (EQUAL (ADD1 J) K)
            (LCC I (SUB1 J) (SUB1 K))
          (LCC I J (SUB1 K))))
    T))

(PROVE-LEMMA LEFT-CHAIN-COMBINE (REWRITE)
              (IMPLIES (AND (NUMBERP I)
                            (NUMBERP J)
                            (NUMBERP K)
                            (LESSP K N)
                            (LESSP I K)
                            (LESSP J K))
                       (EQUAL (AND (LEFT-CHAIN I 0 N STATE)
                                   (LEFT-CHAIN K (ADD1 I) N STATE))
                              (AND (LEFT-CHAIN J 0 N STATE)
                                   (LEFT-CHAIN K (ADD1 J) N STATE))))
              ((INDUCT (LCC I J K))
               (DISABLE HUNGRY OWNS-LEFT
                        BREAK-LEFT-CHAIN)))

(PROVE-LEMMA LESSP-1-1 (REWRITE)
              (EQUAL (LESSP N 1)
                     (ZEROP N)))

(DISABLE LESSP-1-1)

(PROVE-LEMMA LEFT-CHAIN-CANONICALIZE (REWRITE)
              (IMPLIES (AND (NUMBERP I)
                            (LESSP I N))
                       (EQUAL (LEFT-CHAIN I (ADD1-MOD N I) N STATE)
                              (LEFT-CHAIN (SUB1 N) 0 N STATE)))
              ((USE (LEFT-CHAIN-COMBINE (K (SUB1 N)) (I (SUB1 (SUB1 N)))
                                         (J I) (STATE STATE))
                    (BREAK-LEFT-CHAIN (I I) (J (ADD1-MOD N I))
                                       (N N) (STATE STATE)))
               (DISABLE LEFT-CHAIN-COMBINE BREAK-LEFT-CHAIN
                        ALL-LEFTS-IMPLIES HUNGRY OWNS-LEFT LESSP-1)
               (ENABLE LESSP-1-1)))

(PROVE-LEMMA LEFT-CHAIN-CANONICALIZE-ALL-LEFTS (REWRITE)
              (IMPLIES (AND (NOT (ZEROP INDEX))
                            (NOT (LESSP N INDEX)))
```

```
                      (EQUAL (LEFT-CHAIN (SUB1 INDEX) 0 N STATE)
                             (ALL-LEFTS STATE INDEX)))
             ((DISABLE HUNGRY OWNS-LEFT)
              (INDUCT (ALL-LEFTS STATE INDEX))))

(PROVE-LEMMA LEFT-CHAIN-COMPLETE (REWRITE)
             (IMPLIES (AND (NUMBERP I)
                           (LESSP I N))
                      (EQUAL (LEFT-CHAIN I (ADD1-MOD N I) N STATE)
                             (ALL-LEFTS STATE N)))
    ((INSTRUCTIONS PROMOTE (DIVE 1) (REWRITE LEFT-CHAIN-CANONICALIZE)
         (REWRITE LEFT-CHAIN-CANONICALIZE-ALL-LEFTS) TOP S PROVE PROVE
         PROVE)))

(DISABLE LEFT-CHAIN-CANONICALIZE-ALL-LEFTS)
(DISABLE LEFT-CHAIN-CANONICALIZE)
(DISABLE LEFT-CHAIN-COMBINE)
(DISABLE BREAK-LEFT-CHAIN)

(PROVE-LEMMA BREAK-RIGHT-CHAIN (REWRITE)
             (IMPLIES (AND (LESSP I N)
                           (LESSP J N)
                           (NUMBERP I)
                           (NUMBERP J)
                           (LESSP J I))
                      (EQUAL (RIGHT-CHAIN I J N STATE)
                             (AND (RIGHT-CHAIN I (SUB1 N) N STATE)
                                  (RIGHT-CHAIN 0 J N STATE)))))

(DEFN RCC (I J K N)
  (IF (AND (LESSP I N)
           (LESSP J N)
           (NUMBERP K)
           (LESSP K I)
           (LESSP K J)
           (NOT (EQUAL I J))
           (NOT (EQUAL K N)))
      (IF (EQUAL K (SUB1 I))
          (RCC (ADD1 I) J (ADD1 K) N)
        (IF (EQUAL K (SUB1 J))
            (RCC I (ADD1 J) (ADD1 K) N)
          (RCC I J (ADD1 K) N)))
    T)
  ((LESSP (DIFFERENCE N K))))

(PROVE-LEMMA RIGHT-CHAIN-COMBINE (REWRITE)
             (IMPLIES (AND (NUMBERP K)
                           (LESSP I N)
                           (LESSP J N)
                           (LESSP K I)
                           (LESSP K J))
                      (EQUAL (AND (RIGHT-CHAIN I (SUB1 N) N STATE)
                                  (RIGHT-CHAIN K (SUB1 I) N STATE))
                             (AND (RIGHT-CHAIN J (SUB1 N) N STATE)
                                  (RIGHT-CHAIN K (SUB1 J) N STATE))))
    ((INSTRUCTIONS (DISABLE HUNGRY OWNS-RIGHT BREAK-RIGHT-CHAIN)
         (INDUCT (RCC I J K N)) PROMOTE PROMOTE (DIVE 1) (DIVE 1) X UP
         (DIVE 2) X TOP (DIVE 2) (DIVE 2) X TOP (DEMOTE 9) (DIVE 1)
         (DIVE 2) (DIVE 1) (DIVE 2) X TOP
         (PROVE (DISABLE BREAK-RIGHT-CHAIN OWNS-RIGHT HUNGRY)) PROMOTE
         PROMOTE (DIVE 1) (DIVE 2) X TOP (DIVE 2) (DIVE 1) X UP
         (DIVE 2) X TOP (DEMOTE 10) (DIVE 1) (DIVE 2) (DIVE 2) (DIVE 2)
         X TOP PROMOTE
         (PROVE (DISABLE BREAK-RIGHT-CHAIN OWNS-RIGHT HUNGRY)) PROMOTE
         PROMOTE (DIVE 1) (DIVE 2) X TOP (DIVE 2) (DIVE 2) X TOP
         (PROVE (DISABLE BREAK-RIGHT-CHAIN OWNS-RIGHT HUNGRY))
         (PROVE (DISABLE BREAK-RIGHT-CHAIN OWNS-RIGHT HUNGRY)))))

(PROVE-LEMMA RIGHT-CHAIN-CANONICALIZE (REWRITE)
```

```
             (IMPLIES (AND (NUMBERP I)
                           (LESSP I N))
                      (EQUAL (RIGHT-CHAIN I (SUB1-MOD N I) N STATE)
                             (RIGHT-CHAIN 0 (SUB1 N) N STATE)))
             ((USE (RIGHT-CHAIN-COMBINE (K 0) (I 1)
                                        (J I) (STATE STATE))
                   (BREAK-RIGHT-CHAIN (I I) (J (SUB1-MOD N I))
                                      (N N) (STATE STATE)))
              (DISABLE RIGHT-CHAIN-COMBINE BREAK-RIGHT-CHAIN
                       ALL-RIGHTS-IMPLIES HUNGRY OWNS-RIGHT LESSP-1)
              (ENABLE LESSP-1-1)))

(PROVE-LEMMA RIGHT-CHAIN-EXTEND-RIGHT (REWRITE)
             (IMPLIES (AND (NUMBERP I)
                           (NUMBERP J)
                           (LESSP I N)
                           (LESSP J N)
                           (NOT (EQUAL I J)))
                      (EQUAL (RIGHT-CHAIN I J N STATE)
                             (AND (RIGHT-CHAIN I (SUB1-MOD N J) N STATE)
                                  (HUNGRY STATE J)
                                  (OWNS-RIGHT STATE J N))))
             ((DISABLE OWNS-RIGHT HUNGRY)
              (INDUCT (RIGHT-CHAIN I J N STATE))))


(PROVE-LEMMA RIGHT-CHAIN-CANONICALIZE-ALL-RIGHTS (REWRITE)
             (IMPLIES (AND (NOT (ZEROP INDEX))
                           (NOT (LESSP N INDEX)))
                      (EQUAL (RIGHT-CHAIN 0 (SUB1 INDEX) N STATE)
                             (ALL-RIGHTS-REC STATE INDEX N)))
    ((INSTRUCTIONS (DISABLE OWNS-RIGHT HUNGRY)
         (INDUCT (ALL-RIGHTS-REC STATE INDEX N))
         (PROVE (DISABLE HUNGRY OWNS-RIGHT)) PROMOTE PROMOTE (DEMOTE 2)
         (DIVE 1) (DIVE 1) (DIVE 2)
         (= * T ((DISABLE HUNGRY OWNS-RIGHT))) TOP S-PROP S DIVE
         (= * T ((DISABLE HUNGRY OWNS-RIGHT))) TOP DIVE (DIVE 1)
         (REWRITE RIGHT-CHAIN-EXTEND-RIGHT) TOP
         (PROVE (DISABLE HUNGRY OWNS-RIGHT)) PROVE PROVE PROVE)))

(PROVE-LEMMA RIGHT-CHAIN-COMPLETE (REWRITE)
             (IMPLIES (AND (NUMBERP I)
                           (LESSP I N))
                      (EQUAL (RIGHT-CHAIN I (SUB1-MOD N I) N STATE)
                             (ALL-RIGHTS STATE N)))
  ((INSTRUCTIONS PROMOTE (DIVE 1) (REWRITE RIGHT-CHAIN-CANONICALIZE)
                 (REWRITE RIGHT-CHAIN-CANONICALIZE-ALL-RIGHTS)
                 TOP S PROVE
                 PROVE PROVE)))

(DISABLE RIGHT-CHAIN-CANONICALIZE-ALL-RIGHTS)
(DISABLE RIGHT-CHAIN-EXTEND-RIGHT)
(DISABLE RIGHT-CHAIN-CANONICALIZE)
(DISABLE RIGHT-CHAIN-COMBINE)
(DISABLE BREAK-RIGHT-CHAIN)

(PROVE-LEMMA NOT-EVENTUALLY-LEFT-IMPLIES (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP INDEX N)
                (NUMBERP INDEX)
                (INITIAL-CONDITION
                 '(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N))
                (NOT (EVENTUALLY-INVARIANT
                      '(AND (HUNGRY STATE (QUOTE ,INDEX))
                            (OWNS-LEFT STATE (QUOTE ,INDEX)))
                      (PHIL-PRG N))))
           (LEADS-TO '(TRUE)
```

```
                    '(OR (OWNS-RIGHT
                          STATE (QUOTE ,(SUB1-MOD N INDEX))
                          (QUOTE ,N))
                         (OR (FREE STATE (QUOTE ,INDEX))
                             (OR (EATING STATE
                                         (QUOTE ,INDEX))
                                 (THINKING
                                  STATE
                                  (QUOTE ,INDEX)))))
                    (PHIL-PRG N)))
 ((INSTRUCTIONS
   PROMOTE
   (REWRITE NOT-EVENTUALLY-INVARIANT-PROVES-LEADS-TO
            (($NOT-Q (LIST 'AND
                           (LIST 'HUNGRY 'STATE
                                 (LIST 'QUOTE INDEX))
                           (LIST 'OWNS-LEFT 'STATE
                                 (LIST 'QUOTE INDEX))))))
   (CLAIM
    (EVAL (LIST 'PROPER-PHIL 'STATE (LIST 'QUOTE INDEX)
                (LIST 'QUOTE (ADD1-MOD N INDEX)))
          (S (PHIL-PRG N)
             (JES (ILEADS
                   '(TRUE) (PHIL-PRG N)
                   (LIST 'OR
                         (LIST 'OWNS-RIGHT 'STATE
                               (LIST 'QUOTE
                                     (SUB1-MOD N INDEX))
                               (LIST 'QUOTE N))
                         (LIST 'OR
                               (LIST 'FREE 'STATE
                                     (LIST 'QUOTE INDEX))
                               (LIST 'OR
                                     (LIST 'EATING 'STATE
                                           (LIST 'QUOTE INDEX))
                                     (LIST 'THINKING 'STATE
                                           (LIST 'QUOTE INDEX))))))
                  (PHIL-PRG N)
                  (LIST 'AND
                        (LIST 'HUNGRY 'STATE
                              (LIST 'QUOTE INDEX))
                        (LIST 'OWNS-LEFT 'STATE
                              (LIST 'QUOTE INDEX))))))
    ((DISABLE EVAL ADD1-MOD)))
   (CLAIM
    (EVAL (LIST 'PROPER-FORK 'STATE (LIST 'QUOTE INDEX)
                (LIST 'QUOTE (SUB1-MOD N INDEX)))
          (S (PHIL-PRG N)
             (JES (ILEADS '(TRUE) (PHIL-PRG N)
                          (LIST 'OR
                                (LIST 'OWNS-RIGHT 'STATE
                                      (LIST 'QUOTE
                                            (SUB1-MOD N INDEX))
                                      (LIST 'QUOTE N))
                                (LIST 'OR
                                      (LIST 'FREE 'STATE
                                            (LIST 'QUOTE INDEX))
                                      (LIST 'OR
                                            (LIST 'EATING 'STATE
                                                  (LIST 'QUOTE INDEX))
                                            (LIST 'THINKING 'STATE
                                                  (LIST 'QUOTE
                                                        INDEX))))))
                  (PHIL-PRG N)
                  (LIST 'AND
                        (LIST 'HUNGRY 'STATE
                              (LIST 'QUOTE INDEX))
                        (LIST 'OWNS-LEFT 'STATE
                              (LIST 'QUOTE INDEX))))))
```

```
 ((DISABLE EVAL SUB1-MOD)))
(DROP 4 5)
(GENERALIZE
 (((S (PHIL-PRG N)
      (JES (ILEADS '(TRUE) (PHIL-PRG N)
                   (LIST 'OR
                         (LIST 'OWNS-RIGHT 'STATE
                               (LIST 'QUOTE (SUB1-MOD N INDEX))
                               (LIST 'QUOTE N))
                         (LIST 'OR
                               (LIST 'FREE 'STATE
                                     (LIST 'QUOTE INDEX))
                               (LIST 'OR
                                     (LIST 'EATING 'STATE
                                           (LIST 'QUOTE INDEX))
                                     (LIST 'THINKING 'STATE
                                           (LIST 'QUOTE INDEX))))))
           (PHIL-PRG N)
           (LIST 'AND
                 (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX))
                 (LIST 'OWNS-LEFT 'STATE
                       (LIST 'QUOTE INDEX)))))
   STATE)))
(BASH (DISABLE EATING HUNGRY THINKING FORK)))))


(PROVE-LEMMA NOT-EVENTUALLY-RIGHT-IMPLIES (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP INDEX N)
                           (NUMBERP INDEX)
                           (INITIAL-CONDITION
                            `(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N))
                           (NOT (EVENTUALLY-INVARIANT
                                 `(AND (HUNGRY STATE (QUOTE ,INDEX))
                                       (OWNS-RIGHT STATE (QUOTE ,INDEX)
                                                   (QUOTE ,N)))
                                 (PHIL-PRG N))))
                      (LEADS-TO `(TRUE)
                                `(OR (OWNS-LEFT
                                      STATE
                                      (QUOTE ,(ADD1-MOD N INDEX)))
                                     (OR (FREE
                                          STATE
                                          (QUOTE ,(ADD1-MOD N INDEX)))
                                         (OR (EATING STATE (QUOTE ,INDEX))
                                             (THINKING STATE
                                                       (QUOTE ,INDEX)))))
                                (PHIL-PRG N)))
 ((INSTRUCTIONS PROMOTE
      (REWRITE NOT-EVENTUALLY-INVARIANT-PROVES-LEADS-TO
               (($NOT-Q (LIST 'AND
                              (LIST 'HUNGRY 'STATE
                                    (LIST 'QUOTE INDEX))
                              (LIST 'OWNS-RIGHT 'STATE
                                    (LIST 'QUOTE INDEX)
                                    (LIST 'QUOTE N))))))
      (CLAIM (EVAL (LIST 'PROPER-PHIL 'STATE (LIST 'QUOTE INDEX)
                         (LIST 'QUOTE (ADD1-MOD N INDEX)))
                   (S (PHIL-PRG N)
                      (JES (ILEADS '(TRUE) (PHIL-PRG N)
                                   (LIST 'OR
                                    (LIST 'OWNS-LEFT 'STATE
                                     (LIST 'QUOTE
                                      (ADD1-MOD N INDEX)))
                                    (LIST 'OR
                                     (LIST 'FREE 'STATE
                                      (LIST 'QUOTE
```

```
                                        (ADD1-MOD N INDEX)))
                                      (LIST 'OR
                                       (LIST 'EATING 'STATE
                                        (LIST 'QUOTE INDEX))
                                       (LIST 'THINKING 'STATE
                                        (LIST 'QUOTE INDEX))))))
                          (PHIL-PRG N)
                          (LIST 'AND
                                (LIST 'HUNGRY 'STATE
                                 (LIST 'QUOTE INDEX))
                                (LIST 'OWNS-RIGHT 'STATE
                                 (LIST 'QUOTE INDEX)
                                 (LIST 'QUOTE N))))))
            ((DISABLE EVAL ADD1-MOD)))
     (CLAIM (EVAL (LIST 'PROPER-FORK 'STATE
                        (LIST 'QUOTE (ADD1-MOD N INDEX))
                        (LIST 'QUOTE
                              (SUB1-MOD N (ADD1-MOD N INDEX))))
                  (S (PHIL-PRG N)
                     (JES (ILEADS '(TRUE) (PHIL-PRG N)
                                  (LIST 'OR
                                   (LIST 'OWNS-LEFT 'STATE
                                    (LIST 'QUOTE
                                     (ADD1-MOD N INDEX)))
                                   (LIST 'OR
                                    (LIST 'FREE 'STATE
                                     (LIST 'QUOTE
                                      (ADD1-MOD N INDEX)))
                                    (LIST 'OR
                                     (LIST 'EATING 'STATE
                                      (LIST 'QUOTE INDEX))
                                     (LIST 'THINKING 'STATE
                                      (LIST 'QUOTE INDEX))))))
                          (PHIL-PRG N)
                          (LIST 'AND
                                (LIST 'HUNGRY 'STATE
                                 (LIST 'QUOTE INDEX))
                                (LIST 'OWNS-RIGHT 'STATE
                                 (LIST 'QUOTE INDEX)
                                 (LIST 'QUOTE N))))))
            ((DISABLE EVAL SUB1-MOD)))
     (DROP 4 5)
     (GENERALIZE
         (((S (PHIL-PRG N)
              (JES (ILEADS '(TRUE) (PHIL-PRG N)
                           (LIST 'OR
                                 (LIST 'OWNS-LEFT 'STATE
                                  (LIST 'QUOTE (ADD1-MOD N INDEX)))
                                 (LIST 'OR
                                  (LIST 'FREE 'STATE
                                   (LIST 'QUOTE (ADD1-MOD N INDEX)))
                                  (LIST 'OR
                                   (LIST 'EATING 'STATE
                                    (LIST 'QUOTE INDEX))
                                   (LIST 'THINKING 'STATE
                                    (LIST 'QUOTE INDEX))))))
                   (PHIL-PRG N)
                   (LIST 'AND
                         (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX))
                         (LIST 'OWNS-RIGHT 'STATE
                               (LIST 'QUOTE INDEX) (LIST 'QUOTE N)))))
           STATE)))
     (BASH (DISABLE EATING HUNGRY THINKING FORK)))))

(PROVE-LEMMA RIGHT-CHAIN-INDUCTION (REWRITE)
  (IMPLIES (AND (STRONGLY-FAIR (PHIL-PRG N))
                (LESSP 1 N)
                (LESSP I N)
                (LESSP J N)
```

```
                 (NUMBERP I)
                 (NUMBERP J)
                 (INITIAL-CONDITION
                  '(AND (PROPER-PHILS STATE (QUOTE ,N))
                        (PROPER-FORKS STATE (QUOTE ,N)))
                  (PHIL-PRG N))
                 (EVENTUALLY-INVARIANT '(AND (HUNGRY STATE (QUOTE ,J))
                                             (OWNS-RIGHT STATE
                                                         (QUOTE ,J)
                                                         (QUOTE ,N)))
                                       (PHIL-PRG N)))
            (EVENTUALLY-INVARIANT '(RIGHT-CHAIN (QUOTE ,I)
                                                (QUOTE ,J)
                                                (QUOTE ,N)
                                                STATE)
                                  (PHIL-PRG N)))
    ((INSTRUCTIONS (INDUCT (RIGHT-CHAIN I J N STATE)) PROMOTE PROMOTE
         (REWRITE EVENTUALLY-INVARIANT-WEAKEN
                  (($P (LIST 'AND (LIST 'HUNGRY 'STATE (LIST 'QUOTE J))
                             (LIST 'OWNS-RIGHT 'STATE (LIST 'QUOTE J)
                                   (LIST 'QUOTE N))))))
         (DROP 6 8 9) (BASH (DISABLE HUNGRY OWNS-RIGHT)) PROMOTE
         PROMOTE
         (REWRITE EVENTUALLY-INVARIANT-CONJUNCTION
                  (($P (CONS 'RIGHT-CHAIN
                             (CONS (LIST 'QUOTE (ADD1-MOD N I))
                                   (CONS (LIST 'QUOTE J)
                                    (CONS (LIST 'QUOTE N) '(STATE))))))
                   ($Q (LIST 'AND (LIST 'HUNGRY 'STATE (LIST 'QUOTE I))
                             (LIST 'OWNS-RIGHT 'STATE (LIST 'QUOTE I)
                                   (LIST 'QUOTE N))))))
         (BASH (DISABLE EVAL)) (DEMOTE 6) (DIVE 1) (DIVE 1)
         (= * T ((DISABLE EVAL))) TOP SPLIT (CONTRADICT 10) (DIVE 1)
         (REWRITE EVENTUALLY-INVARIANT-FALSE
                  (($Q (LIST 'EATING 'STATE
                             (LIST 'QUOTE (ADD1-MOD N I))))))
         TOP S
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'AND
                             (LIST 'HUNGRY 'STATE
                                   (LIST 'QUOTE (ADD1-MOD N I)))
                             (LIST 'OWNS-RIGHT 'STATE
                                   (LIST 'QUOTE (ADD1-MOD N I))
                                   (LIST 'QUOTE N))))))
         (DROP 8 9 10) BASH (CLAIM (LESSP (ADD1-MOD N I) N))
         (REWRITE HUNGRY-LEADS-TO-OWNS-LEFT-IMPLIES)
         (REWRITE TRUE-LEADS-TO-LEFT-FREE-IMPLIES)
         (REWRITE LEADS-TO-EXPANDED-LEFT-IMPLIES)
         (REWRITE NOT-EVENTUALLY-RIGHT-IMPLIES) (DROP 6 8 9 10) BASH
         (DROP 6 7 9 10) BASH PROVE)))


(PROVE-LEMMA EVENTUALLY-INVARIANT-RIGHT-IMPLIES-ALL-RIGHTS (REWRITE)
    (IMPLIES (AND (LESSP 1 N)
                  (LESSP J N)
                  (NUMBERP J)
                  (STRONGLY-FAIR (PHIL-PRG N))
                  (INITIAL-CONDITION
                   '(AND (PROPER-PHILS STATE (QUOTE ,N))
                         (PROPER-FORKS STATE (QUOTE ,N)))
                   (PHIL-PRG N))
                  (EVENTUALLY-INVARIANT '(AND (HUNGRY STATE (QUOTE ,J))
                                              (OWNS-RIGHT STATE
                                                          (QUOTE ,J)
                                                          (QUOTE ,N)))
                                        (PHIL-PRG N)))
             (EVENTUALLY-INVARIANT '(ALL-RIGHTS STATE
                                                (QUOTE ,N))
                                   (PHIL-PRG N)))
```

```
 ((INSTRUCTIONS
   PROMOTE
   (REWRITE EVENTUALLY-INVARIANT-WEAKEN
            (($P (LIST 'RIGHT-CHAIN (LIST 'QUOTE (ADD1-MOD N J))
                       (LIST 'QUOTE J) (LIST 'QUOTE N) 'STATE))))
   (REWRITE RIGHT-CHAIN-INDUCTION) PROVE (DROP 4 5 6)
   (USE-LEMMA
    RIGHT-CHAIN-COMPLETE
    ((I (ADD1-MOD N J)) (N N)
     (STATE (S (PHIL-PRG N)
               (JES (IES (PHIL-PRG N)
                         (CONS 'RIGHT-CHAIN
                               (CONS
                                (LIST 'QUOTE (ADD1-MOD N J))
                                (CONS (LIST 'QUOTE J)
                                      (CONS (LIST 'QUOTE N)
                                            '(STATE))))))
                    (PHIL-PRG N)
                    (LIST 'ALL-RIGHTS 'STATE (LIST 'QUOTE N)))))))
   (BASH (DISABLE ALL-RIGHTS)))))

(PROVE-LEMMA NEVER-ALL-RIGHTS (REWRITE)
             (IMPLIES (AND (DEADLOCK-FREE (PHIL-PRG N))
                           (LESSP 1 N))
                      (INVARIANT `(NOT (ALL-RIGHTS STATE (QUOTE ,N)))
                                 (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE LEADS-TO-FALSE-INVARIANT
                  (($P (LIST 'ALL-RIGHTS 'STATE (LIST 'QUOTE N)))))
         (REWRITE DEADLOCK-FREEDOM
                  (($C '(AND (HUNGRY STATE '0) (FREE STATE '0)))
                   ($E '(HUNGRY-LEFT 0))))
         (REWRITE ALL-RIGHTS-INV)
         (REWRITE PROVE-ENABLING-CONDITION
                  (($NEW (UPDATE-ASSOC (CONS 'F 0) 0
                             (OLDC-1 (LIST 'AND
                                      (LIST 'HUNGRY 'STATE
                                       (LIST 'QUOTE 0))
                                      (LIST 'FREE 'STATE
                                       (LIST 'QUOTE 0)))
                                     (LIST 'HUNGRY-LEFT 0))))))
         PROVE PROVE PROVE (DROP 1) BASH PROMOTE (DEMOTE 4) (DIVE 1) X
         TOP BASH (BASH (DISABLE EVAL)))))

(PROVE-LEMMA NOT-EVENTUALLY-OWNS-RIGHT (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (LESSP J N)
                           (NUMBERP J)
                           (STRONGLY-FAIR (PHIL-PRG N))
                           (DEADLOCK-FREE (PHIL-PRG N))
                           (INITIAL-CONDITION
                            `(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N)))
                      (NOT (EVENTUALLY-INVARIANT
                            `(AND (HUNGRY STATE (QUOTE ,J))
                                  (OWNS-RIGHT STATE
                                              (QUOTE ,J)
                                              (QUOTE ,N)))
                            (PHIL-PRG N))))
    ((INSTRUCTIONS PROMOTE
         (USE-LEMMA EVENTUALLY-INVARIANT-RIGHT-IMPLIES-ALL-RIGHTS)
         (DEMOTE 7) (DIVE 1) (DIVE 2)
         (REWRITE EVENTUALLY-INVARIANT-FALSE (($Q '(TRUE)))) TOP
         (BASH (DISABLE EVAL))
         (REWRITE LEADS-TO-STRENGTHEN-LEFT (($P '(FALSE))))
         (CLAIM (EVAL (LIST 'NOT
                            (LIST 'ALL-RIGHTS 'STATE (LIST 'QUOTE N)))
                      (S (PHIL-PRG N)
```

```
                        (ILEADS (LIST 'ALL-RIGHTS 'STATE
                                      (LIST 'QUOTE N))
                                (PHIL-PRG N) '(TRUE))))
                ((DISABLE EVAL EVAL-NOT)))
         (BASH (DISABLE EVAL)) (REWRITE LEADS-TO-TRUE)
         (CLAIM (EVAL (LIST 'NOT
                          (LIST 'ALL-RIGHTS 'STATE (LIST 'QUOTE N)))
                     (S (PHIL-PRG N)
                        (JLEADS (IES (PHIL-PRG N)
                                     (LIST 'ALL-RIGHTS 'STATE
                                      (LIST 'QUOTE N)))
                                (PHIL-PRG N) '(TRUE))))
                ((DISABLE EVAL EVAL-NOT)))
         (BASH (DISABLE EVAL)))))

(PROVE-LEMMA HUNGRY-LEADS-TO-OWNS-LEFT (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (NUMBERP INDEX)
                           (LESSP INDEX N)
                           (INITIAL-CONDITION
                            '(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N))
                           (STRONGLY-FAIR (PHIL-PRG N))
                           (DEADLOCK-FREE (PHIL-PRG N)))
                      (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
                                '(OWNS-LEFT STATE (QUOTE ,INDEX))
                                (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE (REWRITE TRUE-LEADS-TO-LEFT-FREE-IMPLIES)
         (USE-LEMMA LEADS-TO-EXPANDED-LEFT-IMPLIES
             ((INDEX (SUB1-MOD N INDEX))))
         (DEMOTE 7) (DIVE 1) (DIVE 1) (DIVE 2 2 2 2)
         (REWRITE NOT-EVENTUALLY-RIGHT-IMPLIES) TOP BASH PROVE BASH)))

(PROVE-LEMMA LEFT-CHAIN-INDUCTION (REWRITE)
  (IMPLIES (AND (STRONGLY-FAIR (PHIL-PRG N))
                (LESSP 1 N)
                (LESSP I N)
                (LESSP J N)
                (NUMBERP I)
                (NUMBERP J)
                (INITIAL-CONDITION
                 '(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N))
                (EVENTUALLY-INVARIANT '(AND (HUNGRY STATE (QUOTE ,J))
                                            (OWNS-LEFT STATE
                                                       (QUOTE ,J)))
                                      (PHIL-PRG N)))
           (EVENTUALLY-INVARIANT '(LEFT-CHAIN (QUOTE ,I)
                                              (QUOTE ,J)
                                              (QUOTE ,N)
                                              STATE)
                                 (PHIL-PRG N)))
    ((INSTRUCTIONS (INDUCT (LEFT-CHAIN I J N STATE)) PROMOTE PROMOTE
         (REWRITE EVENTUALLY-INVARIANT-WEAKEN
                  (($P (LIST 'AND (LIST 'HUNGRY 'STATE (LIST 'QUOTE J))
                             (LIST 'OWNS-LEFT 'STATE (LIST 'QUOTE J))))))
         (DROP 6 8 9) (BASH (DISABLE HUNGRY OWNS-LEFT)) PROMOTE PROMOTE
         (REWRITE EVENTUALLY-INVARIANT-CONJUNCTION
                  (($P (CONS 'LEFT-CHAIN
                             (CONS (LIST 'QUOTE (SUB1-MOD N I))
                                   (CONS (LIST 'QUOTE J)
                                    (CONS (LIST 'QUOTE N) '(STATE))))))
                   ($Q (LIST 'AND (LIST 'HUNGRY 'STATE (LIST 'QUOTE I))
                             (LIST 'OWNS-LEFT 'STATE (LIST 'QUOTE I))))))
         (BASH (DISABLE EVAL)) (DEMOTE 6) (DIVE 1) (DIVE 1)
         (= * T ((DISABLE EVAL))) TOP SPLIT (CONTRADICT 10) (DIVE 1)
         (REWRITE EVENTUALLY-INVARIANT-FALSE
```

```
               (($Q (LIST 'EATING 'STATE
                          (LIST 'QUOTE (SUB1-MOD N I))))))
         TOP S
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
               (($P (LIST 'AND
                          (LIST 'HUNGRY 'STATE
                                (LIST 'QUOTE (SUB1-MOD N I)))
                          (LIST 'OWNS-LEFT 'STATE
                                (LIST 'QUOTE (SUB1-MOD N I)))))))
         (DROP 8 9 10) BASH (CLAIM (LESSP (SUB1-MOD N I) N))
         (REWRITE HUNGRY-LEADS-TO-OWNS-RIGHT-IMPLIES)
         (REWRITE TRUE-LEADS-TO-RIGHT-FREE-IMPLIES) (DIVE 2) (DIVE 2)
         (DIVE 1) (DIVE 2) (DIVE 2) (DIVE 1) (DIVE 2) (DIVE 1) (= I)
         TOP (REWRITE LEADS-TO-EXPANDED-RIGHT-IMPLIES)
         (REWRITE NOT-EVENTUALLY-LEFT-IMPLIES) (DROP 6 8 9 10) BASH
         (DROP 6 7 9 10) BASH PROVE)))


(PROVE-LEMMA EVENTUALLY-INVARIANT-LEFT-IMPLIES-ALL-LEFTS (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP J N)
                (NUMBERP J)
                (STRONGLY-FAIR (PHIL-PRG N))
                (INITIAL-CONDITION
                 `(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N))
                (EVENTUALLY-INVARIANT `(AND (HUNGRY STATE (QUOTE ,J))
                                            (OWNS-LEFT STATE
                                                       (QUOTE ,J)))
                                      (PHIL-PRG N)))
           (EVENTUALLY-INVARIANT `(ALL-LEFTS STATE
                                             (QUOTE ,N))
                                 (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE EVENTUALLY-INVARIANT-WEAKEN
               (($P (LIST 'LEFT-CHAIN (LIST 'QUOTE (SUB1-MOD N J))
                          (LIST 'QUOTE J) (LIST 'QUOTE N) 'STATE))))
         (REWRITE LEFT-CHAIN-INDUCTION) PROVE (DROP 4 5 6)
         (USE-LEMMA LEFT-CHAIN-COMPLETE
             ((I (SUB1-MOD N J)) (N N)
              (STATE (S (PHIL-PRG N)
                        (JES (IES (PHIL-PRG N)
                                  (CONS 'LEFT-CHAIN
                                        (CONS
                                         (LIST 'QUOTE (SUB1-MOD N J))
                                         (CONS (LIST 'QUOTE J)
                                          (CONS (LIST 'QUOTE N)
                                           '(STATE))))))
                          (PHIL-PRG N)
                          (LIST 'ALL-LEFTS 'STATE (LIST 'QUOTE N)))))))
         (BASH (DISABLE ALL-LEFTS)))))

(PROVE-LEMMA NEVER-ALL-LEFTS (REWRITE)
             (IMPLIES (AND (DEADLOCK-FREE (PHIL-PRG N))
                           (LESSP 1 N))
                      (INVARIANT `(NOT (ALL-LEFTS STATE (QUOTE ,N)))
                                 (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE LEADS-TO-FALSE-INVARIANT
               (($P (LIST 'ALL-LEFTS 'STATE (LIST 'QUOTE N)))))
         (REWRITE DEADLOCK-FREEDOM
               (($C (LIST 'AND
                          (LIST 'HUNGRY 'STATE
                                (LIST 'QUOTE (SUB1 (SUB1 N))))
                          (LIST 'FREE 'STATE (LIST 'QUOTE (SUB1 N)))))
                ($E (LIST 'HUNGRY-RIGHT (SUB1 (SUB1 N)) N))))
         (REWRITE ALL-LEFTS-INV)
         (REWRITE PROVE-ENABLING-CONDITION
```

```
              (($NEW (UPDATE-ASSOC (CONS 'F (SUB1 N))
                           (SUB1 (SUB1 N))
                           (OLDC-1 (LIST 'AND
                                    (LIST 'HUNGRY 'STATE
                                     (LIST 'QUOTE (SUB1 (SUB1 N))))
                                    (LIST 'FREE 'STATE
                                     (LIST 'QUOTE (SUB1 N))))
                                   (LIST 'HUNGRY-RIGHT
                                    (SUB1 (SUB1 N)) N))))))
         BASH BASH BASH (DROP 1) BASH S)))

(PROVE-LEMMA NOT-EVENTUALLY-OWNS-LEFT (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (LESSP J N)
                (NUMBERP J)
                (STRONGLY-FAIR (PHIL-PRG N))
                (DEADLOCK-FREE (PHIL-PRG N))
                (INITIAL-CONDITION
                 '(AND (PROPER-PHILS STATE (QUOTE ,N))
                       (PROPER-FORKS STATE (QUOTE ,N)))
                 (PHIL-PRG N)))
           (NOT (EVENTUALLY-INVARIANT '(AND (HUNGRY STATE (QUOTE ,J))
                                            (OWNS-LEFT STATE
                                                       (QUOTE ,J)))
                                       (PHIL-PRG N))))
    ((INSTRUCTIONS PROMOTE
         (USE-LEMMA EVENTUALLY-INVARIANT-LEFT-IMPLIES-ALL-LEFTS)
         (DEMOTE 7) (DIVE 1) (DIVE 2)
         (REWRITE EVENTUALLY-INVARIANT-FALSE (($Q '(TRUE)))) TOP
         (BASH (DISABLE EVAL))
         (REWRITE LEADS-TO-STRENGTHEN-LEFT (($P '(FALSE))))
         (CLAIM (EVAL (LIST 'NOT
                            (LIST 'ALL-LEFTS 'STATE (LIST 'QUOTE N)))
                      (S (PHIL-PRG N)
                         (ILEADS (LIST 'ALL-LEFTS 'STATE
                                       (LIST 'QUOTE N))
                                 (PHIL-PRG N) '(TRUE))))
                ((DISABLE EVAL EVAL-NOT)))
         (BASH (DISABLE EVAL)) (REWRITE LEADS-TO-TRUE)
         (CLAIM (EVAL (LIST 'NOT
                            (LIST 'ALL-LEFTS 'STATE (LIST 'QUOTE N)))
                      (S (PHIL-PRG N)
                         (JLEADS (IES (PHIL-PRG N)
                                      (LIST 'ALL-LEFTS 'STATE
                                       (LIST 'QUOTE N)))
                                 (PHIL-PRG N) '(TRUE))))
                ((DISABLE EVAL EVAL-NOT)))
         (BASH (DISABLE EVAL)))))

(PROVE-LEMMA HUNGRY-LEADS-TO-OWNS-RIGHT (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (NUMBERP INDEX)
                           (LESSP INDEX N)
                           (INITIAL-CONDITION
                            '(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N))
                           (STRONGLY-FAIR (PHIL-PRG N))
                           (DEADLOCK-FREE (PHIL-PRG N)))
                      (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
                                '(OWNS-RIGHT STATE (QUOTE ,INDEX)
                                              (QUOTE ,N))
                                (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE (REWRITE TRUE-LEADS-TO-RIGHT-FREE-IMPLIES)
         (USE-LEMMA LEADS-TO-EXPANDED-RIGHT-IMPLIES
             ((INDEX (ADD1-MOD N INDEX))))
         (DEMOTE 7) (DIVE 1) (DIVE 1) (DIVE 2 2 2 2)
         (REWRITE NOT-EVENTUALLY-LEFT-IMPLIES) TOP BASH PROVE BASH)))
```

```
(PROVE-LEMMA CORRECTNESS (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (NUMBERP INDEX)
                           (LESSP INDEX N)
                           (INITIAL-CONDITION
                            '(AND (PROPER-PHILS STATE (QUOTE ,N))
                                  (PROPER-FORKS STATE (QUOTE ,N)))
                            (PHIL-PRG N))
                           (STRONGLY-FAIR (PHIL-PRG N))
                           (DEADLOCK-FREE (PHIL-PRG N)))
                      (LEADS-TO '(HUNGRY STATE (QUOTE ,INDEX))
                                '(EATING STATE (QUOTE ,INDEX))
                                (PHIL-PRG N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE LEADS-TO-TRANSITIVE
                  (($Q (LIST 'OR
                             (LIST 'EATING 'STATE (LIST 'QUOTE INDEX))
                             (LIST 'AND
                                   (LIST 'HUNGRY 'STATE
                                    (LIST 'QUOTE INDEX))
                                   (LIST 'OWNS-LEFT 'STATE
                                    (LIST 'QUOTE INDEX)))))))
         (REWRITE PSP-GENERAL
                  (($P (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX)))
                   ($Q (LIST 'OWNS-LEFT 'STATE (LIST 'QUOTE INDEX)))
                   ($R (LIST 'HUNGRY 'STATE (LIST 'QUOTE INDEX)))
                   ($B (LIST 'EATING 'STATE (LIST 'QUOTE INDEX)))))
         (REWRITE HUNGRY-LEADS-TO-OWNS-LEFT)
         (REWRITE HUNGRY-UNLESS-EATING) (BASH (DISABLE EVAL))
         (BASH (DISABLE EVAL)) BASH (REWRITE DISJOIN-LEFT)
         (REWRITE Q-LEADS-TO-Q)
         (REWRITE HUNGRY-LEADS-TO-OWNS-RIGHT-IMPLIES)
         (REWRITE HUNGRY-LEADS-TO-OWNS-RIGHT))))


))
```

Appendix F
FIFO Circuit Events

This appendix contains the complete events list supporting the proof of the delay insensitive FIFO circuit described in chapter 7.

This events list constructs the proof of the delay insensitive FIFO circuit on top of the proof system presented in Appendix B.

```
(NOTE-LIB "INTERPRETER")

(PROVEALL "FIFO" '(


(DEFN VALUE (ALIST KEY)
  (CDR (ASSOC KEY ALIST)))

(DEFN CT (I)
  (CONS 'CT I))

(DEFN CF (I)
  (CONS 'CF I))

(DEFN TEMP (I)
  (CONS 'TEMP I))

(DEFN C-ELEMENT (OLD NEW A B C)
  (IF (IFF (VALUE OLD A)
           (VALUE OLD B))
      (AND (IFF (VALUE NEW C) (VALUE OLD A))
           (CHANGED OLD NEW (LIST C)))
    (CHANGED OLD NEW NIL)))

(DEFN NOR-GATE (OLD NEW A B C)
  (AND (IFF (VALUE NEW C)
            (NOT (OR (VALUE OLD A)
                     (VALUE OLD B))))
       (CHANGED OLD NEW (LIST C))))


(DEFN FIFO-NODE (I)
  (LIST (LIST 'C-ELEMENT (CT (ADD1 I)) (TEMP I) (CT I))
        (LIST 'C-ELEMENT (CF (ADD1 I)) (TEMP I) (CF I))
        (LIST 'NOR-GATE  (CT (SUB1 I)) (CF (SUB1 I)) (TEMP I))))

(DEFN EMPTY-NODE (STATE I)
  (AND (NOT (VALUE STATE (CT I)))
       (NOT (VALUE STATE (CF I)))))


(DEFN TRUE-NODE (STATE I)
  (AND (VALUE STATE (CT I))
       (NOT (VALUE STATE (CF I)))))

(DEFN FALSE-NODE (STATE I)
  (AND (NOT (VALUE STATE (CT I)))
       (VALUE STATE (CF I))))

(DEFN IN-NODE (OLD NEW I)
  (IF (IFF (VALUE OLD (TEMP I))
           (EMPTY-NODE OLD I))
      (IF (EMPTY-NODE OLD I)
          (OR (CHANGED OLD NEW NIL)
              (AND (OR (TRUE-NODE NEW I)
```

```
                          (FALSE-NODE NEW I))
                      (EQUAL (VALUE NEW 'INPUT)
                             (CONS (TRUE-NODE NEW I)
                                   (VALUE OLD 'INPUT)))
                      (CHANGED OLD NEW (LIST (CT I) (CF I) 'INPUT))))
           (OR (CHANGED OLD NEW NIL)
               (AND (EMPTY-NODE NEW I)
                    (CHANGED OLD NEW (LIST (CT I) (CF I))))))
       (CHANGED OLD NEW NIL)))


(DEFN OUT-NODE (OLD NEW)
  (AND (IFF (VALUE NEW (CT 0))
            (VALUE OLD (CT 1)))
       (IFF (VALUE NEW (CF 0))
            (VALUE OLD (CF 1)))
       (IF (AND (EMPTY-NODE OLD 0)
                (NOT (EMPTY-NODE NEW 0)))
           (EQUAL (VALUE NEW 'OUTPUT)
                  (CONS (TRUE-NODE NEW 0)
                        (VALUE OLD 'OUTPUT)))
         (EQUAL (VALUE NEW 'OUTPUT) (VALUE OLD 'OUTPUT)))
       (CHANGED OLD NEW (LIST (CT 0) (CF 0) 'OUTPUT))))

(DEFN INTERNAL-NODES (N)
  (IF (ZEROP N)
      NIL
    (APPEND (FIFO-NODE N)
            (INTERNAL-NODES (SUB1 N)))))

(DEFN EXTERNAL-NODES (N)
  (LIST (LIST 'IN-NODE N)
        (LIST 'OUT-NODE)
        (LIST 'NOR-GATE  (CT (SUB1 N)) (CF (SUB1 N)) (TEMP N))))

(DEFN FIFO-QUEUE (N)
  (APPEND (EXTERNAL-NODES N)
          (INTERNAL-NODES (SUB1 N))))

(PROVE-LEMMA MEMBER-INTERNAL-NODES (REWRITE)
  (EQUAL (MEMBER STATEMENT (INTERNAL-NODES N))
         (AND (MEMBER STATEMENT (FIFO-NODE (CDADDDR STATEMENT)))
              (NOT (LESSP N (CDADDDR STATEMENT)))
              (NOT (ZEROP (CDADDDR STATEMENT))))))


(PROVE-LEMMA MEMBER-FIFO-QUEUE (REWRITE)
  (IMPLIES (LESSP 1 N)
           (EQUAL (MEMBER STATEMENT (FIFO-QUEUE N))
                  (OR (MEMBER STATEMENT (INTERNAL-NODES (SUB1 N)))
                      (MEMBER STATEMENT (EXTERNAL-NODES N)))))
             ((DISABLE MEMBER-INTERNAL-NODES)))

(PROVE-LEMMA LISTP-FIFO-QUEUE (REWRITE)
             (LISTP (FIFO-QUEUE N)))

(DISABLE FIFO-QUEUE)
(DISABLE *1*FIFO-QUEUE)


(DEFN PROPER-NODE (STATE I)
  (AND (IMPLIES (AND (NOT (EMPTY-NODE STATE I))
                     (EMPTY-NODE STATE (SUB1 I)))
                (VALUE STATE (TEMP I)))
       (IMPLIES (AND (EMPTY-NODE STATE I)
                     (NOT (EMPTY-NODE STATE (SUB1 I))))
                (NOT (VALUE STATE (TEMP I))))
       (OR (TRUE-NODE STATE I)
           (FALSE-NODE STATE I)
```

```
         (EMPTY-NODE STATE I))
       (IMPLIES (NOT (EMPTY-NODE STATE I))
                (OR (EMPTY-NODE STATE (SUB1 I))
                    (IF (TRUE-NODE STATE I)
                        (TRUE-NODE STATE (SUB1 I))
                      (FALSE-NODE STATE (SUB1 I)))))))

(DEFN PROPER-NODES (STATE N)
  (IF (ZEROP N)
      T
    (AND (PROPER-NODE STATE N)
         (PROPER-NODES STATE (SUB1 N)))))

(PROVE-LEMMA PROPER-NODES-IMPLIES (REWRITE)
             (IMPLIES (AND (PROPER-NODES STATE N)
                           (NOT (LESSP N I))
                           (NOT (ZEROP I)))
                      (PROPER-NODE STATE I))
             ((DISABLE PROPER-NODE)))

(PROVE-LEMMA PROPER-NODES-PRESERVED-GENERAL (REWRITE)
             (IMPLIES (AND (N OLD NEW STATEMENT)
                           (PROPER-NODES OLD N)
                           (MEMBER STATEMENT (FIFO-QUEUE N))
                           (NOT (LESSP N I))
                           (LESSP 1 N))
                      (PROPER-NODES NEW I))
    ((INSTRUCTIONS (INDUCT (PROPER-NODES NEW I))
         (BASH (DISABLE MEMBER-FIFO-QUEUE))
         (BASH (DISABLE PROPER-NODE MEMBER-FIFO-QUEUE N)) PROMOTE
         (DROP 3)
         (CLAIM (PROPER-NODE OLD I)
                ((DISABLE PROPER-NODE MEMBER-FIFO-QUEUE N)))
         (CLAIM (EQUAL I N) 0)
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4 5)
         PROVE
         (CLAIM (NOT (EQUAL I (CDADDDR STATEMENT)))
                ((DISABLE PROPER-NODE)))
         (CLAIM (EQUAL I (ADD1 (CDADDDR STATEMENT))) 0) PROVE PROVE
         (CLAIM (EQUAL I 1) 0)
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4 5)
         (CLAIM (NOT (EQUAL N 0))) PROVE
         (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         (CLAIM (PROPER-NODE OLD (ADD1 I)) ((DISABLE PROPER-NODE)))
         (DROP 4) PROVE (DROP 4) PROVE
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4 5)
         (CLAIM (AND (NOT (EQUAL (SUB1 I) N)) (NOT (EQUAL (SUB1 I) 0)))
                ((DISABLE PROPER-NODE EXTERNAL-NODES N)))
         PROVE (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         (CLAIM (PROPER-NODE OLD (ADD1 I))
                ((DISABLE PROPER-NODE EXTERNAL-NODES
                          MEMBER-INTERNAL-NODES N)))
         (DROP 4) PROVE (CLAIM (EQUAL I (ADD1 (CDADDDR STATEMENT))) 0)
         (DROP 4) PROVE (DROP 4) PROVE)))

(PROVE-LEMMA PROPER-NODES-PRESERVED (REWRITE)
             (IMPLIES (AND (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (FIFO-QUEUE N))
                           (PROPER-NODES OLD N)
                           (LESSP 1 N))
                      (PROPER-NODES NEW N))
             ((DISABLE MEMBER-FIFO-QUEUE N)))

(DISABLE PROPER-NODES-PRESERVED-GENERAL)

(PROVE-LEMMA PROPER-NODES-UNLESS-FALSE (REWRITE)
             (IMPLIES (LESSP 1 N)
                      (UNLESS '(PROPER-NODES STATE (QUOTE ,N))
                              '(FALSE)
```

```
                              (FIFO-QUEUE N)))
             ((DISABLE MEMBER-FIFO-QUEUE N)))


(PROVE-LEMMA PROPER-NODES-INVARIANT (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (INITIAL-CONDITION '(PROPER-NODES STATE (QUOTE ,N))
                                    (FIFO-QUEUE N)))
           (INVARIANT '(PROPER-NODES STATE (QUOTE ,N))
                       (FIFO-QUEUE N))))

(PROVE-LEMMA PROPER-NODE-INVARIANT (REWRITE)
  (IMPLIES (AND (LESSP 1 N)
                (NOT (ZEROP I))
                (NOT (LESSP N I))
                (INITIAL-CONDITION '(PROPER-NODES STATE (QUOTE ,N))
                                    (FIFO-QUEUE N)))
           (INVARIANT '(PROPER-NODE STATE (QUOTE ,I))
                       (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE INVARIANT-CONSEQUENCE
                  (($P (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N)))))
         (REWRITE PROPER-NODES-INVARIANT) (BASH (DISABLE PROPER-NODE)))))

(DEFN QUEUE-VALUES (STATE N)
  (IF (ZEROP N)
      NIL
    (IF (AND (NOT (EMPTY-NODE STATE N))
             (EMPTY-NODE STATE (SUB1 N)))
        (CONS (TRUE-NODE STATE N)
              (QUEUE-VALUES STATE (SUB1 N)))
      (QUEUE-VALUES STATE (SUB1 N)))))

(PROVE-LEMMA EMPTY-EMPTY-EMPTY (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (FIFO-QUEUE N))
                           (EMPTY-NODE OLD (ADD1 I))
                           (EMPTY-NODE OLD I)
                           (NUMBERP I)
                           (LESSP I N))
                      (EMPTY-NODE NEW I))
    ((INSTRUCTIONS PROMOTE (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         PROVE (CLAIM (NOT (EQUAL I N))) PROVE)))

(PROVE-LEMMA FULL-FULL-FULL (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (FIFO-QUEUE N))
                           (NOT (EMPTY-NODE OLD (ADD1 I)))
                           (NOT (EMPTY-NODE OLD I))
                           (OR (IFF (TRUE-NODE OLD (ADD1 I))
                                    (TRUE-NODE OLD I))
                               (IFF (FALSE-NODE OLD (ADD1 I))
                                    (FALSE-NODE OLD I)))
                           (NUMBERP I)
                           (LESSP I N))
                      (NOT (EMPTY-NODE NEW I)))
    ((INSTRUCTIONS PROMOTE (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         PROVE (CLAIM (NOT (EQUAL I N))) PROVE)))

(PROVE-LEMMA QUEUE-EMPTY-PRESERVED (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (FIFO-QUEUE N))
                           (PROPER-NODES OLD N)
                           (EQUAL (QUEUE-VALUES OLD (ADD1 I)) NIL)
                           (LESSP I N))
                      (EQUAL (QUEUE-VALUES NEW I) NIL))
```

```
    ((INSTRUCTIONS (INDUCT (QUEUE-VALUES OLD I))
         (DISABLE TRUE-NODE FALSE-NODE EMPTY-NODE N MEMBER-FIFO-QUEUE)
         (BASH (DISABLE MEMBER-FIFO-QUEUE N EMPTY-NODE FALSE-NODE
                        TRUE-NODE))
         (BASH (DISABLE MEMBER-FIFO-QUEUE N EMPTY-NODE FALSE-NODE
                        TRUE-NODE))
         PROMOTE PROMOTE
         (CLAIM (PROPER-NODE OLD I)
                ((DISABLE MEMBER-FIFO-QUEUE N EMPTY-NODE FALSE-NODE
                          TRUE-NODE PROPER-NODE)))
         (BASH (DISABLE MEMBER-FIFO-QUEUE N EMPTY-NODE FALSE-NODE
                        TRUE-NODE PROPER-NODES-IMPLIES)))))

(PROVE-LEMMA VALUE-MOVES-FORWARD (REWRITE)
             (IMPLIES (AND (LESSP 1 N)
                           (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (FIFO-QUEUE N))
                           (PROPER-NODES OLD N)
                           (NOT (EMPTY-NODE OLD (ADD1 I)))
                           (EMPTY-NODE OLD I)
                           (LESSP I N)
                           (NOT (ZEROP I)))
                      (OR (AND (NOT (EMPTY-NODE NEW (ADD1 I)))
                               (EMPTY-NODE NEW I))
                          (AND (NOT (EMPTY-NODE NEW I))
                               (EMPTY-NODE NEW (SUB1 I))))))
   ((INSTRUCTIONS PROMOTE
                  (CLAIM (PROPER-NODE OLD I)
                         ((DISABLE PROPER-NODE MEMBER-FIFO-QUEUE N
                                   EMPTY-NODE)))
                  (CLAIM (PROPER-NODE OLD (ADD1 I))
                         ((DISABLE PROPER-NODE MEMBER-FIFO-QUEUE N
                                   EMPTY-NODE)))
                  (CLAIM (EQUAL (ADD1 I) N) 0)
                  (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N))
                         0)
                  (DROP 3 4)
                  PROVE
                  (CLAIM (EQUAL I (CDADDDR STATEMENT))
                         0)
                  (DROP 4)
                  PROVE
                  (CLAIM (NOT (EQUAL (ADD1 I) (CDADDDR STATEMENT)))
                         ((DISABLE PROPER-NODE)))
                  (DROP 4)
                  PROVE
                  (CLAIM (NOT (EQUAL I N))
                         ((DISABLE MEMBER-FIFO-QUEUE N PROPER-NODE
                                   EMPTY-NODE)))
                  (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N))
                         0)
                  (DROP 3 4)
                  PROVE
                  (CLAIM (EQUAL (ADD1 I) (CDADDDR STATEMENT))
                         0)
                  (DROP 4)
                  PROVE
                  (CLAIM (EQUAL I (CDADDDR STATEMENT))
                         0)
                  (DROP 4)
                  PROVE
                  (DROP 4)
                  PROVE)))

(PROVE-LEMMA NOT-LISTP-QUEUE-VALUES-EQUALS (REWRITE)
             (EQUAL (LISTP (QUEUE-VALUES STATE N))
                    (NOT (EQUAL (QUEUE-VALUES STATE N) NIL))))

(PROVE-LEMMA FULL-EMPTY-REST-UNLESS-MOVES-FORWARD (REWRITE)
```

```
(IMPLIES (AND (LESSP 1 N)
              (LESSP I N)
              (NOT (ZEROP I)))
         (UNLESS `(AND (PROPER-NODES STATE (QUOTE ,N))
                       (AND (NOT (EMPTY-NODE STATE
                                             (QUOTE ,(ADD1 I))))
                            (AND (EMPTY-NODE STATE (QUOTE ,I))
                                 (NOT (LISTP (QUEUE-VALUES
                                              STATE
                                              (QUOTE ,I)))))))
                 `(AND (NOT (EMPTY-NODE STATE (QUOTE ,I)))
                       (AND (EMPTY-NODE STATE (QUOTE ,(SUB1 I)))
                            (NOT (LISTP (QUEUE-VALUES
                                         STATE
                                         (QUOTE ,(SUB1 I)))))))
                 (FIFO-QUEUE N)))
  ((INSTRUCTIONS PROMOTE (REWRITE HELP-PROVE-UNLESS)
       (GENERALIZE
           (((EU (LIST 'AND
                       (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                       (LIST 'AND
                             (LIST 'NOT
                                   (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE (ADD1 I))))
                             (LIST 'AND
                                   (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE I))
                                   (LIST 'NOT
                                    (LIST 'LISTP
                                     (LIST 'QUEUE-VALUES 'STATE
                                      (LIST 'QUOTE I)))))))
                 (FIFO-QUEUE N)
                 (LIST 'AND
                       (LIST 'NOT
                             (LIST 'EMPTY-NODE 'STATE
                                   (LIST 'QUOTE I)))
                       (LIST 'AND
                             (LIST 'EMPTY-NODE 'STATE
                                   (LIST 'QUOTE (SUB1 I)))
                             (LIST 'NOT
                                   (LIST 'LISTP
                                    (LIST 'QUEUE-VALUES 'STATE
                                     (LIST 'QUOTE (SUB1 I))))))))
             STATEMENT)
            ((OLDU (LIST 'AND
                         (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                         (LIST 'AND
                               (LIST 'NOT
                                     (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE (ADD1 I))))
                               (LIST 'AND
                                     (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE I))
                                     (LIST 'NOT
                                      (LIST 'LISTP
                                       (LIST 'QUEUE-VALUES 'STATE
                                        (LIST 'QUOTE I)))))))
                   (FIFO-QUEUE N)
                   (LIST 'AND
                         (LIST 'NOT
                               (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE I)))
                         (LIST 'AND
                               (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE (SUB1 I)))
                               (LIST 'NOT
                                     (LIST 'LISTP
                                      (LIST 'QUEUE-VALUES 'STATE
                                       (LIST 'QUOTE (SUB1 I))))))))
```

```
              OLD)
             ((NEWU (LIST 'AND
                          (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                          (LIST 'AND
                                (LIST 'NOT
                                      (LIST 'EMPTY-NODE 'STATE
                                       (LIST 'QUOTE (ADD1 I))))
                                (LIST 'AND
                                      (LIST 'EMPTY-NODE 'STATE
                                       (LIST 'QUOTE I))
                                      (LIST 'NOT
                                       (LIST 'LISTP
                                        (LIST 'QUEUE-VALUES 'STATE
                                         (LIST 'QUOTE I)))))))
                    (FIFO-QUEUE N)
                    (LIST 'AND
                          (LIST 'NOT
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE I)))
                          (LIST 'AND
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE (SUB1 I)))
                                (LIST 'NOT
                                      (LIST 'LISTP
                                       (LIST 'QUEUE-VALUES 'STATE
                                        (LIST 'QUOTE (SUB1 I))))))))
              NEW)))
        (USE-LEMMA VALUE-MOVES-FORWARD NIL)
        (BASH (DISABLE MEMBER-FIFO-QUEUE N EMPTY-NODE)))))

(PROVE-LEMMA OUTPUT-ONLY-ADDS-BOOLEAN (REWRITE)
             (IMPLIES (LESSP 1 N)
                      (UNLESS '(EQUAL (VALUE STATE (QUOTE OUTPUT))
                                      (QUOTE ,K))
                              '(OR (EQUAL (VALUE STATE (QUOTE OUTPUT))
                                          (CONS (TRUE) (QUOTE ,K)))
                                   (EQUAL (VALUE STATE (QUOTE OUTPUT))
                                          (CONS (FALSE) (QUOTE ,K))))
                              (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
                   (REWRITE HELP-PROVE-UNLESS)
        (GENERALIZE (((EU (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                (LIST 'QUOTE K))
                          (FIFO-QUEUE N)
                          (LIST 'OR
                                (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                      (LIST 'CONS '(TRUE)
                                       (LIST 'QUOTE K)))
                                (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                      (LIST 'CONS '(FALSE)
                                       (LIST 'QUOTE K)))))
                      STATEMENT)
                     ((OLDU (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                  (LIST 'QUOTE K))
                            (FIFO-QUEUE N)
                            (LIST 'OR
                                  (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                   (LIST 'CONS '(TRUE)
                                    (LIST 'QUOTE K)))
                                  (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                   (LIST 'CONS '(FALSE)
                                    (LIST 'QUOTE K)))))
                      OLD)
                     ((NEWU (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                  (LIST 'QUOTE K))
                            (FIFO-QUEUE N)
                            (LIST 'OR
                                  (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                   (LIST 'CONS '(TRUE)
```

```
                                      (LIST 'QUOTE K)))
                                    (LIST 'EQUAL '(VALUE STATE 'OUTPUT)
                                     (LIST 'CONS '(FALSE)
                                      (LIST 'QUOTE K)))))
                        NEW)))
          PROVE)))

(PROVE-LEMMA INPUT-ONLY-ADDS-BOOLEAN (REWRITE)
             (IMPLIES (LESSP 1 N)
                      (UNLESS `(EQUAL (VALUE STATE (QUOTE INPUT))
                                      (QUOTE ,K))
                              `(OR (EQUAL (VALUE STATE (QUOTE INPUT))
                                          (CONS (TRUE) (QUOTE ,K)))
                                   (EQUAL (VALUE STATE (QUOTE INPUT))
                                          (CONS (FALSE) (QUOTE ,K))))
                              (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE (REWRITE HELP-PROVE-UNLESS)
         (GENERALIZE (((EU (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                 (LIST 'QUOTE K))
                           (FIFO-QUEUE N)
                           (LIST 'OR
                                 (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                       (LIST 'CONS '(TRUE)
                                        (LIST 'QUOTE K)))
                                 (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                       (LIST 'CONS '(FALSE)
                                        (LIST 'QUOTE K)))))
                        STATEMENT)
                       ((OLDU (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                    (LIST 'QUOTE K))
                              (FIFO-QUEUE N)
                              (LIST 'OR
                                    (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                     (LIST 'CONS '(TRUE)
                                      (LIST 'QUOTE K)))
                                    (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                     (LIST 'CONS '(FALSE)
                                      (LIST 'QUOTE K)))))
                        OLD)
                       ((NEWU (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                    (LIST 'QUOTE K))
                              (FIFO-QUEUE N)
                              (LIST 'OR
                                    (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                     (LIST 'CONS '(TRUE)
                                      (LIST 'QUOTE K)))
                                    (LIST 'EQUAL '(VALUE STATE 'INPUT)
                                     (LIST 'CONS '(FALSE)
                                      (LIST 'QUOTE K)))))
                        NEW)))
          PROVE)))

(PROVE-LEMMA OUTPUT-NEVER-SHORTENS (REWRITE)
             (IMPLIES (LESSP 1 N)
                      (UNLESS `(EQUAL (LENGTH (VALUE STATE 'OUTPUT))
                                      (QUOTE ,K))
                              `(LESSP (QUOTE ,K)
                                      (LENGTH (VALUE STATE 'OUTPUT)))
                              (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE (REWRITE HELP-PROVE-UNLESS)
         (GENERALIZE
             (((EU (LIST 'EQUAL '(LENGTH (VALUE STATE 'OUTPUT))
                         (LIST 'QUOTE K))
                   (FIFO-QUEUE N)
                   (CONS 'LESSP
                         (CONS (LIST 'QUOTE K)
                               '((LENGTH (VALUE STATE 'OUTPUT))))))
               STATEMENT)
              ((OLDU (LIST 'EQUAL '(LENGTH (VALUE STATE 'OUTPUT))
```

```
                        (LIST 'QUOTE K))
                  (FIFO-QUEUE N)
                  (CONS 'LESSP
                        (CONS (LIST 'QUOTE K)
                              '((LENGTH (VALUE STATE 'OUTPUT))))))
            OLD)
           ((NEWU (LIST 'EQUAL '(LENGTH (VALUE STATE 'OUTPUT))
                        (LIST 'QUOTE K))
                  (FIFO-QUEUE N)
                  (CONS 'LESSP
                        (CONS (LIST 'QUOTE K)
                              '((LENGTH (VALUE STATE 'OUTPUT))))))
            NEW)))
      (BASH T) PROMOTE (DIVE 2) (DIVE 1) = TOP DROP PROVE PROMOTE
      (DIVE 2) (DIVE 1) = TOP DROP PROVE PROMOTE (DIVE 2) (DIVE 1) =
      TOP DROP PROVE)))


(PROVE-LEMMA TOTAL-SUFFICIENT-FIFO-QUEUE (REWRITE)
             (IMPLIES (LESSP 1 N)
                      (TOTAL-SUFFICIENT
                       STATEMENT (FIFO-QUEUE N)
                       OLD
                       (IF (EQUAL (CAR STATEMENT) 'C-ELEMENT)
                           (IF (IFF (VALUE OLD (CADR STATEMENT))
                                    (VALUE OLD (CADDR STATEMENT)))
                               (UPDATE-ASSOC (CADDDR STATEMENT)
                                             (VALUE OLD
                                                    (CADR STATEMENT))
                                             OLD)
                             OLD)
                         (IF (EQUAL (CAR STATEMENT) 'NOR-GATE)
                             (UPDATE-ASSOC
                              (CADDDR STATEMENT)
                              (NOT (OR (VALUE OLD (CADR STATEMENT))
                                       (VALUE OLD (CADDR STATEMENT))))
                              OLD)
                           (IF (EQUAL (CAR STATEMENT) 'IN-NODE)
                               (IF (AND (EQUAL (VALUE
                                                 OLD
                                                 (TEMP
                                                  (CADR STATEMENT)))
                                                (EMPTY-NODE
                                                 OLD
                                                 (CADR STATEMENT)))
                                        (NOT (EMPTY-NODE
                                               OLD
                                               (CADR STATEMENT))))
                                   (UPDATE-ASSOC
                                    (CT (CADR STATEMENT)) F
                                    (UPDATE-ASSOC
                                     (CF (CADR STATEMENT)) F
                                     OLD))
                                 OLD)
                             (IF (EQUAL (CAR STATEMENT) 'OUT-NODE)
                                 (UPDATE-ASSOC
                                  (CT 0)
                                  (VALUE OLD (CT 1))
                                  (UPDATE-ASSOC
                                   (CF 0)
                                   (VALUE OLD (CF 1))
                                   (IF (AND (EMPTY-NODE OLD 0)
                                            (NOT (EMPTY-NODE OLD 1)))
                                       (UPDATE-ASSOC
                                        'OUTPUT
                                        (CONS (TRUE-NODE OLD 1)
                                              (VALUE OLD 'OUTPUT))
                                        OLD)
                                     OLD)))
```

```
                               OLD)))))))

    (PROVE-LEMMA TOTAL-FIFO-QUEUE (REWRITE)
                 (IMPLIES (LESSP 1 N)
                          (TOTAL (FIFO-QUEUE N)))
                 ((DISABLE TOTAL-SUFFICIENT
                           TOTAL-SUFFICIENT-FIFO-QUEUE)
                  (USE (TOTAL-SUFFICIENT-FIFO-QUEUE
                        (OLD (OLDT (FIFO-QUEUE N)))
                        (STATEMENT (ET (FIFO-QUEUE N)))))))

    (PROVE-LEMMA OUTPUT-GROWS-IMMEDIATELY (REWRITE)
                 (IMPLIES (AND (INITIAL-CONDITION
                                '(PROPER-NODES STATE (QUOTE ,N))
                                (FIFO-QUEUE N))
                               (LESSP 1 N))
                          (LEADS-TO '(AND (NOT (EMPTY-NODE STATE 1))
                                          (AND (EMPTY-NODE STATE 0)
                                               (EQUAL (LENGTH (VALUE STATE
                                                                     'OUTPUT))
                                                      (QUOTE ,K))))
                                    '(LESSP (QUOTE ,K)
                                            (LENGTH (VALUE STATE 'OUTPUT)))
                                    (FIFO-QUEUE N)))
        ((INSTRUCTIONS PROMOTE
             (REWRITE UNCONDITIONAL-FAIRNESS-GENERAL
                      (($P-1 (LIST 'AND
                                   (LIST 'PROPER-NODES 'STATE
                                         (LIST 'QUOTE N))
                                   (LIST 'AND '(NOT (EMPTY-NODE STATE 1))
                                         (LIST 'AND '(EMPTY-NODE STATE 0)
                                          (LIST 'EQUAL
                                           '(LENGTH (VALUE STATE 'OUTPUT))
                                           (LIST 'QUOTE K))))))
                       ($Q-1 (LIST 'LESSP (LIST 'QUOTE K)
                                   '(LENGTH (VALUE STATE 'OUTPUT))))
                       ($P-2 (LIST 'AND '(NOT (EMPTY-NODE STATE 1))
                                   (LIST 'AND '(EMPTY-NODE STATE 0)
                                         (LIST 'EQUAL
                                          '(LENGTH (VALUE STATE 'OUTPUT))
                                          (LIST 'QUOTE K)))))
                       ($Q-2 (LIST 'LESSP (LIST 'QUOTE K)
                                   '(LENGTH (VALUE STATE 'OUTPUT))))))
             (REWRITE HELP-PROVE-UNLESS)
             (GENERALIZE
                 (((EU (LIST 'AND
                             (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                             (LIST 'AND '(NOT (EMPTY-NODE STATE 1))
                                   (LIST 'AND '(EMPTY-NODE STATE 0)
                                         (LIST 'EQUAL
                                          '(LENGTH (VALUE STATE 'OUTPUT))
                                          (LIST 'QUOTE K)))))
                       (FIFO-QUEUE N)
                       (CONS 'LESSP
                             (CONS (LIST 'QUOTE K)
                                   '((LENGTH (VALUE STATE 'OUTPUT))))))
                   STATEMENT)
                  ((OLDU (LIST 'AND
                               (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                               (LIST 'AND '(NOT (EMPTY-NODE STATE 1))
                                     (LIST 'AND '(EMPTY-NODE STATE 0)
                                           (LIST 'EQUAL
                                            '(LENGTH (VALUE STATE 'OUTPUT))
                                            (LIST 'QUOTE K)))))
                         (FIFO-QUEUE N)
                         (CONS 'LESSP
                               (CONS (LIST 'QUOTE K)
                                     '((LENGTH (VALUE STATE 'OUTPUT))))))
                   OLD)
```

```
             ((NEWU (LIST 'AND
                          (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                          (LIST 'AND '(NOT (EMPTY-NODE STATE 1))
                                (LIST 'AND '(EMPTY-NODE STATE 0)
                                      (LIST 'EQUAL
                                       '(LENGTH (VALUE STATE 'OUTPUT))
                                       (LIST 'QUOTE K)))))
                    (FIFO-QUEUE N)
                    (CONS 'LESSP
                          (CONS (LIST 'QUOTE K)
                                '((LENGTH (VALUE STATE 'OUTPUT))))))
              NEW)))
        X-DUMB PROMOTE (CLAIM (PROPER-NODE OLD 1) 0) (DROP 1 6)
        (REWRITE EVAL-OR) (DIVE 1) (REWRITE EVAL-AND) (DIVE 1)
        (= * T 0) TOP (CLAIM (NOT (EQUAL STATEMENT '(OUT-NODE))) 0)
        (CLAIM (NOT (EQUAL N 1))
               ((DISABLE MEMBER-FIFO-QUEUE PROPER-NODE)))
        (CLAIM (NOT (EQUAL N 0))
               ((DISABLE MEMBER-FIFO-QUEUE PROPER-NODE)))
        (CLAIM (IMPLIES (MEMBER STATEMENT (INTERNAL-NODES (SUB1 N)))
                        (EQUAL (CDADDDR STATEMENT) 1))
               0)
        (PROVE (DISABLE PROPER-NODES-IMPLIES PROPER-NODES)) (DROP 2)
        (PROVE (DISABLE PROPER-NODES-IMPLIES PROPER-NODES)) (DROP 2)
        (BASH (DISABLE PROPER-NODES PROPER-NODES-IMPLIES)) PROMOTE
        (DIVE 2) (DIVE 1) = TOP DROP PROVE PROMOTE (DIVE 2) (DIVE 1) =
        TOP DROP PROVE (DROP 5) (BASH (DISABLE MEMBER-FIFO-QUEUE N))
        (CONTRADICT 7) (DROP 1 6 7) (BASH (DISABLE PROPER-NODE))
        (REWRITE HELP-PROVE-ENSURES (($STATEMENT '(OUT-NODE))))
        (DROP 1)
        (GENERALIZE
            (((OLDE '(OUT-NODE)
                    (LIST 'AND '(NOT (EMPTY-NODE STATE 1))
                          (LIST 'AND '(EMPTY-NODE STATE 0)
                                (LIST 'EQUAL
                                      '(LENGTH (VALUE STATE 'OUTPUT))
                                      (LIST 'QUOTE K))))
                    (CONS 'LESSP
                          (CONS (LIST 'QUOTE K)
                                '((LENGTH (VALUE STATE 'OUTPUT))))))
              OLD)
             ((NEWE '(OUT-NODE)
                    (LIST 'AND '(NOT (EMPTY-NODE STATE 1))
                          (LIST 'AND '(EMPTY-NODE STATE 0)
                                (LIST 'EQUAL
                                      '(LENGTH (VALUE STATE 'OUTPUT))
                                      (LIST 'QUOTE K))))
                    (CONS 'LESSP
                          (CONS (LIST 'QUOTE K)
                                '((LENGTH (VALUE STATE 'OUTPUT))))))
              NEW)))
        (BASH (DISABLE MEMBER-INTERNAL-NODES)) PROMOTE (DIVE 2)
        (DIVE 1) = TOP DROP PROVE PROMOTE (DIVE 2) (DIVE 1) = TOP DROP
        PROVE PROMOTE (DIVE 2) (DIVE 1) = TOP DROP PROVE
        (BASH (DISABLE EVAL MEMBER-FIFO-QUEUE N))
        (BASH (DISABLE EVAL MEMBER-FIFO-QUEUE N)) S
        (BASH (DISABLE EVAL MEMBER-FIFO-QUEUE N))
        (REWRITE TOTAL-FIFO-QUEUE))))

(PROVE-LEMMA TRUE-EMPTY-TEMP-MOVES (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                             `(PROPER-NODES STATE (QUOTE ,N))
                             (FIFO-QUEUE N))
                           (LESSP 1 N)
                           (NOT (ZEROP I))
                           (LESSP I N))
                      (LEADS-TO `(AND (TRUE-NODE STATE (QUOTE ,(ADD1 I)))
                                      (VALUE STATE (TEMP (QUOTE ,I))))
                                `(NOT (EMPTY-NODE STATE (QUOTE ,I)))
```

```
                              (FIFO-QUEUE N)))
((INSTRUCTIONS PROMOTE
     (REWRITE UNCONDITIONAL-FAIRNESS-GENERAL
              (($P-1 (LIST 'AND
                           (LIST 'PROPER-NODES 'STATE
                                 (LIST 'QUOTE N))
                           (LIST 'AND
                                 (LIST 'TRUE-NODE 'STATE
                                  (LIST 'QUOTE (ADD1 I)))
                                 (LIST 'VALUE 'STATE
                                  (LIST 'TEMP (LIST 'QUOTE I))))))
               ($Q-1 (LIST 'NOT
                           (LIST 'EMPTY-NODE 'STATE
                                 (LIST 'QUOTE I))))
               ($P-2 (LIST 'AND
                           (LIST 'TRUE-NODE 'STATE
                                 (LIST 'QUOTE (ADD1 I)))
                           (LIST 'VALUE 'STATE
                                 (LIST 'TEMP (LIST 'QUOTE I)))))
               ($Q-2 (LIST 'NOT
                           (LIST 'EMPTY-NODE 'STATE
                                 (LIST 'QUOTE I))))))
     (DROP 1) (REWRITE HELP-PROVE-UNLESS)
     (GENERALIZE
         (((EU (LIST 'AND
                     (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                     (LIST 'AND
                           (LIST 'TRUE-NODE 'STATE
                                 (LIST 'QUOTE (ADD1 I)))
                           (LIST 'VALUE 'STATE
                                 (LIST 'TEMP (LIST 'QUOTE I)))))
               (FIFO-QUEUE N)
               (LIST 'NOT
                     (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I))))
           STATEMENT)
          ((OLDU (LIST 'AND
                       (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                       (LIST 'AND
                             (LIST 'TRUE-NODE 'STATE
                                   (LIST 'QUOTE (ADD1 I)))
                             (LIST 'VALUE 'STATE
                                   (LIST 'TEMP (LIST 'QUOTE I)))))
                 (FIFO-QUEUE N)
                 (LIST 'NOT
                       (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I))))
           OLD)
          ((NEWU (LIST 'AND
                       (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                       (LIST 'AND
                             (LIST 'TRUE-NODE 'STATE
                                   (LIST 'QUOTE (ADD1 I)))
                             (LIST 'VALUE 'STATE
                                   (LIST 'TEMP (LIST 'QUOTE I)))))
                 (FIFO-QUEUE N)
                 (LIST 'NOT
                       (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I))))
           NEW)))
     X-DUMB PROMOTE (CLAIM (PROPER-NODE OLD (ADD1 I)) 0)
     (CLAIM (PROPER-NODE OLD I) 0) (REWRITE EVAL-OR) (DIVE 1)
     (REWRITE EVAL-AND) (DIVE 1) (= * T 0) TOP
     (CLAIM (EQUAL (ADD1 I) N) 0)
     (CLAIM (NOT (EQUAL I N))
            ((DISABLE MEMBER-FIFO-QUEUE N PROPER-NODES
                      PROPER-NODES-IMPLIES)))
     (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4)
     (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
     (CLAIM (NOT (EQUAL (ADD1 I) (CDADDDR STATEMENT)))
            ((DISABLE PROPER-NODE EVAL)))
     (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
```

```
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0)
         (CLAIM (NOT (EQUAL I N))
                ((DISABLE EVAL MEMBER-FIFO-QUEUE PROPER-NODE)))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (NOT (EQUAL I N))
                ((DISABLE PROPER-NODE MEMBER-FIFO-QUEUE)))
         (CLAIM (EQUAL (ADD1 I) (CDADDDR STATEMENT)) 0)
         (CLAIM (PROPER-NODE OLD (ADD1 (ADD1 I)))
                ((DISABLE PROPER-NODE N MEMBER-FIFO-QUEUE)))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES PROPER-NODES
                         N MEMBER-FIFO-QUEUE))
         (PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
         (PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
         (REWRITE HELP-PROVE-ENSURES
                  (($STATEMENT
                       (LIST 'C-ELEMENT (CT (ADD1 I)) (TEMP I) (CT I)))))
         PROVE (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)) S
         (BASH (DISABLE EVAL)) (REWRITE TOTAL-FIFO-QUEUE))))

(PROVE-LEMMA FALSE-EMPTY-TEMP-MOVES (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                             '(PROPER-NODES STATE (QUOTE ,N))
                             (FIFO-QUEUE N))
                            (LESSP 1 N)
                            (NOT (ZEROP I))
                            (LESSP I N))
                       (LEADS-TO '(AND (FALSE-NODE STATE (QUOTE ,(ADD1 I)))
                                       (VALUE STATE (TEMP (QUOTE ,I))))
                                 '(NOT (EMPTY-NODE STATE (QUOTE ,I)))
                                 (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE UNCONDITIONAL-FAIRNESS-GENERAL
                  (($P-1 (LIST 'AND
                                (LIST 'PROPER-NODES 'STATE
                                      (LIST 'QUOTE N))
                                (LIST 'AND
                                      (LIST 'FALSE-NODE 'STATE
                                       (LIST 'QUOTE (ADD1 I)))
                                      (LIST 'VALUE 'STATE
                                       (LIST 'TEMP (LIST 'QUOTE I))))))
                   ($Q-1 (LIST 'NOT
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE I))))
                   ($P-2 (LIST 'AND
                                (LIST 'FALSE-NODE 'STATE
                                      (LIST 'QUOTE (ADD1 I)))
                                (LIST 'VALUE 'STATE
                                      (LIST 'TEMP (LIST 'QUOTE I)))))
                   ($Q-2 (LIST 'NOT
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE I))))))
         (DROP 1) (REWRITE HELP-PROVE-UNLESS)
         (GENERALIZE
             (((EU (LIST 'AND
                         (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                         (LIST 'AND
                               (LIST 'FALSE-NODE 'STATE
                                     (LIST 'QUOTE (ADD1 I)))
                               (LIST 'VALUE 'STATE
                                     (LIST 'TEMP (LIST 'QUOTE I)))))
                   (FIFO-QUEUE N)
                   (LIST 'NOT
                         (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I))))
```

```
                  STATEMENT)
                 ((OLDU (LIST 'AND
                              (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                              (LIST 'AND
                                    (LIST 'FALSE-NODE 'STATE
                                          (LIST 'QUOTE (ADD1 I)))
                                    (LIST 'VALUE 'STATE
                                          (LIST 'TEMP (LIST 'QUOTE I)))))
                        (FIFO-QUEUE N)
                        (LIST 'NOT
                              (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I))))
                  OLD)
                 ((NEWU (LIST 'AND
                              (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                              (LIST 'AND
                                    (LIST 'FALSE-NODE 'STATE
                                          (LIST 'QUOTE (ADD1 I)))
                                    (LIST 'VALUE 'STATE
                                          (LIST 'TEMP (LIST 'QUOTE I)))))
                        (FIFO-QUEUE N)
                        (LIST 'NOT
                              (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I))))
                  NEW)))
            X-DUMB PROMOTE (CLAIM (PROPER-NODE OLD (ADD1 I)) 0)
            (CLAIM (PROPER-NODE OLD I) 0) (REWRITE EVAL-OR) (DIVE 1)
            (REWRITE EVAL-AND) (DIVE 1) (= * T 0) TOP
            (CLAIM (EQUAL (ADD1 I) N) 0)
            (CLAIM (NOT (EQUAL I N))
                   ((DISABLE MEMBER-FIFO-QUEUE N PROPER-NODES
                             PROPER-NODES-IMPLIES)))
            (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4)
            (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
            (CLAIM (NOT (EQUAL (ADD1 I) (CDADDDR STATEMENT)))
                   ((DISABLE PROPER-NODE EVAL)))
            (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
            (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
            (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
            (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0)
            (CLAIM (NOT (EQUAL I N))
                   ((DISABLE EVAL MEMBER-FIFO-QUEUE PROPER-NODE)))
            (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
            (CLAIM (NOT (EQUAL I N))
                   ((DISABLE PROPER-NODE MEMBER-FIFO-QUEUE)))
            (CLAIM (EQUAL (ADD1 I) (CDADDDR STATEMENT)) 0)
            (CLAIM (PROPER-NODE OLD (ADD1 (ADD1 I)))
                   ((DISABLE PROPER-NODE N MEMBER-FIFO-QUEUE)))
            (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
            (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
            (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
            (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
            (PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES PROPER-NODES
                            N MEMBER-FIFO-QUEUE))
            (PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
            (PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
            (REWRITE HELP-PROVE-ENSURES
                     (($STATEMENT
                          (LIST 'C-ELEMENT (CF (ADD1 I)) (TEMP I) (CF I)))))
            PROVE (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)) S
            (BASH (DISABLE EVAL)) (REWRITE TOTAL-FIFO-QUEUE))))

(PROVE-LEMMA NOT-EMPTY-EMPTY-TEMP-MOVES (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                             `(PROPER-NODES STATE (QUOTE ,N))
                             (FIFO-QUEUE N))
                           (LESSP 1 N)
                           (NOT (ZEROP I))
                           (LESSP I N))
                      (LEADS-TO `(AND (NOT (EMPTY-NODE STATE
                                                       (QUOTE ,(ADD1 I))))
```

```
                              (VALUE STATE (TEMP (QUOTE ,I))))
                        ‘(NOT (EMPTY-NODE STATE (QUOTE ,I)))
                        (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST ’OR
                             (LIST ’AND
                                   (LIST ’TRUE-NODE ’STATE
                                    (LIST ’QUOTE (ADD1 I)))
                                   (LIST ’VALUE ’STATE
                                    (LIST ’TEMP (LIST ’QUOTE I))))
                             (LIST ’AND
                                   (LIST ’FALSE-NODE ’STATE
                                    (LIST ’QUOTE (ADD1 I)))
                                   (LIST ’VALUE ’STATE
                                    (LIST ’TEMP (LIST ’QUOTE I))))))))
         (CLAIM (EVAL (LIST ’PROPER-NODE ’STATE (LIST ’QUOTE (ADD1 I)))
                      (S (FIFO-QUEUE N)
                         (ILEADS (LIST ’AND
                                       (LIST ’NOT
                                        (LIST ’EMPTY-NODE ’STATE
                                         (LIST ’QUOTE (ADD1 I))))
                                       (LIST ’VALUE ’STATE
                                        (LIST ’TEMP (LIST ’QUOTE I))))
                                 (FIFO-QUEUE N)
                                 (LIST ’NOT
                                       (LIST ’EMPTY-NODE ’STATE
                                        (LIST ’QUOTE I))))))
                ((DISABLE EVAL)))
         (DROP 1) PROVE (REWRITE DISJOIN-LEFT)
         (REWRITE TRUE-EMPTY-TEMP-MOVES)
         (REWRITE FALSE-EMPTY-TEMP-MOVES))))

(PROVE-LEMMA EMPTY-EMPTY-SETS-TEMP (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                            ‘(PROPER-NODES STATE (QUOTE ,N))
                            (FIFO-QUEUE N))
                           (LESSP 1 N)
                           (NUMBERP I)
                           (LESSP I N))
                      (LEADS-TO ‘(AND (EMPTY-NODE STATE
                                                  (QUOTE ,(ADD1 I)))
                                      (EMPTY-NODE STATE (QUOTE ,I)))
                                ‘(VALUE STATE (TEMP (QUOTE ,(ADD1 I))))
                                (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE UNCONDITIONAL-FAIRNESS-GENERAL
                  (($P-1 (LIST ’AND
                               (LIST ’PROPER-NODES ’STATE
                                     (LIST ’QUOTE N))
                               (LIST ’AND
                                     (LIST ’EMPTY-NODE ’STATE
                                      (LIST ’QUOTE (ADD1 I)))
                                     (LIST ’EMPTY-NODE ’STATE
                                      (LIST ’QUOTE I)))))
                   ($Q-1 (LIST ’VALUE ’STATE
                               (LIST ’TEMP (LIST ’QUOTE (ADD1 I)))))
                   ($P-2 (LIST ’AND
                               (LIST ’EMPTY-NODE ’STATE
                                     (LIST ’QUOTE (ADD1 I)))
                               (LIST ’EMPTY-NODE ’STATE
                                     (LIST ’QUOTE I))))
                   ($Q-2 (LIST ’VALUE ’STATE
                               (LIST ’TEMP (LIST ’QUOTE (ADD1 I)))))))
         (REWRITE HELP-PROVE-UNLESS)
         (GENERALIZE
             (((EU (LIST ’AND
                         (LIST ’PROPER-NODES ’STATE (LIST ’QUOTE N))
                         (LIST ’AND
```

```
                              (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE (ADD1 I)))
                              (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE I))))
                  (FIFO-QUEUE N)
                  (LIST 'VALUE 'STATE
                        (LIST 'TEMP (LIST 'QUOTE (ADD1 I)))))
              STATEMENT)
             ((OLDU (LIST 'AND
                          (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                          (LIST 'AND
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE (ADD1 I)))
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE I))))
                    (FIFO-QUEUE N)
                    (LIST 'VALUE 'STATE
                          (LIST 'TEMP (LIST 'QUOTE (ADD1 I)))))
              OLD)
             ((NEWU (LIST 'AND
                          (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                          (LIST 'AND
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE (ADD1 I)))
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE I))))
                    (FIFO-QUEUE N)
                    (LIST 'VALUE 'STATE
                          (LIST 'TEMP (LIST 'QUOTE (ADD1 I)))))
              NEW)))
        X-DUMB PROMOTE (DROP 1) (CLAIM (PROPER-NODE OLD (ADD1 I)) 0)
        (REWRITE EVAL-OR) (DIVE 1) (REWRITE EVAL-AND) (DIVE 1)
        (= * T 0) TOP (CLAIM (EQUAL (ADD1 I) N) 0)
        (DISABLE MEMBER-FIFO-QUEUE N PROPER-NODES
                 PROPER-NODES-IMPLIES)
        (CLAIM (NOT (EQUAL I N))
               ((DISABLE PROPER-NODES-IMPLIES PROPER-NODES N
                         MEMBER-FIFO-QUEUE)))
        (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4)
        (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
        (CLAIM (NOT (EQUAL (ADD1 I) (CDADDDR STATEMENT)))
               ((DISABLE PROPER-NODE EVAL)))
        (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
        (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
        (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
        (CLAIM (NOT (EQUAL I N))
               ((DISABLE EVAL MEMBER-FIFO-QUEUE PROPER-NODE)))
        (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0)
        (CLAIM (EQUAL I 0) 0)
        (PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES))
        (PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES))
        (CLAIM (EQUAL (ADD1 I) (CDADDDR STATEMENT)) 0)
        (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
        (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
        (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
        (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
        (PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES PROPER-NODES
                        N MEMBER-FIFO-QUEUE))
        (PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
        (REWRITE HELP-PROVE-ENSURES
                 (($STATEMENT
                      (LIST 'NOR-GATE (CT I) (CF I) (TEMP (ADD1 I))))))
        PROVE (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)) S
        (BASH (DISABLE EVAL)) (REWRITE TOTAL-FIFO-QUEUE))))

(PROVE-LEMMA FULL-FULL-UNSETS-TEMP (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION
                 `(PROPER-NODES STATE (QUOTE ,N))
                 (FIFO-QUEUE N))
```

```
            (LESSP 1 N)
            (NUMBERP I)
            (LESSP I N))
       (LEADS-TO '(AND (NOT (EMPTY-NODE STATE
                                  (QUOTE ,(ADD1 I))))
                     (NOT (EMPTY-NODE STATE (QUOTE ,I))))
                '(NOT (VALUE STATE (TEMP (QUOTE ,(ADD1 I)))))
                (FIFO-QUEUE N)))
((INSTRUCTIONS PROMOTE
     (REWRITE UNCONDITIONAL-FAIRNESS-GENERAL
              (($P-1 (LIST 'AND
                           (LIST 'PROPER-NODES 'STATE
                                 (LIST 'QUOTE N))
                           (LIST 'AND (LIST 'NOT
                                 (LIST 'EMPTY-NODE 'STATE
                                  (LIST 'QUOTE (ADD1 I))))
                                 (LIST 'NOT (LIST 'EMPTY-NODE 'STATE
                                  (LIST 'QUOTE I))))))
               ($Q-1 (LIST 'NOT (LIST 'VALUE 'STATE
                           (LIST 'TEMP (LIST 'QUOTE (ADD1 I))))))
               ($P-2 (LIST 'AND (LIST 'NOT
                           (LIST 'EMPTY-NODE 'STATE
                                 (LIST 'QUOTE (ADD1 I))))
                           (LIST 'NOT (LIST 'EMPTY-NODE 'STATE
                                 (LIST 'QUOTE I)))))
               ($Q-2 (LIST 'NOT (LIST 'VALUE 'STATE
                           (LIST 'TEMP (LIST 'QUOTE (ADD1 I))))))))
     (REWRITE HELP-PROVE-UNLESS)
     (GENERALIZE
         (((EU (LIST 'AND
                     (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                     (LIST 'AND (LIST 'NOT
                           (LIST 'EMPTY-NODE 'STATE
                                 (LIST 'QUOTE (ADD1 I))))
                           (LIST 'NOT (LIST 'EMPTY-NODE 'STATE
                                 (LIST 'QUOTE I)))))
               (FIFO-QUEUE N)
               (LIST 'NOT (LIST 'VALUE 'STATE
                                (LIST 'TEMP (LIST 'QUOTE (ADD1 I))))))
           STATEMENT)
          ((OLDU (LIST 'AND
                       (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                       (LIST 'AND (LIST 'NOT
                             (LIST 'EMPTY-NODE 'STATE
                                   (LIST 'QUOTE (ADD1 I))))
                             (LIST 'NOT (LIST 'EMPTY-NODE 'STATE
                                   (LIST 'QUOTE I)))))
                 (FIFO-QUEUE N)
                 (LIST 'NOT (LIST 'VALUE 'STATE
                       (LIST 'TEMP (LIST 'QUOTE (ADD1 I))))))
           OLD)
          ((NEWU (LIST 'AND
                       (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                       (LIST 'AND (LIST 'NOT
                             (LIST 'EMPTY-NODE 'STATE
                                   (LIST 'QUOTE (ADD1 I))))
                             (LIST 'NOT (LIST 'EMPTY-NODE 'STATE
                                   (LIST 'QUOTE I)))))
                 (FIFO-QUEUE N)
                 (LIST 'NOT (LIST 'VALUE 'STATE
                       (LIST 'TEMP (LIST 'QUOTE (ADD1 I))))))
           NEW)))
     X-DUMB PROMOTE (DROP 1) (CLAIM (PROPER-NODE OLD (ADD1 I)) 0)
     (REWRITE EVAL-OR) (DIVE 1) (REWRITE EVAL-AND) (DIVE 1)
     (= * T 0) TOP (CLAIM (EQUAL (ADD1 I) N) 0)
     (DISABLE MEMBER-FIFO-QUEUE N PROPER-NODES
              PROPER-NODES-IMPLIES)
     (CLAIM (NOT (EQUAL I N))
            ((DISABLE PROPER-NODES-IMPLIES PROPER-NODES N
```

```
                       MEMBER-FIFO-QUEUE)))
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (NOT (EQUAL (ADD1 I) (CDADDDR STATEMENT)))
                ((DISABLE PROPER-NODE EVAL)))
         (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (NOT (EQUAL I N))
                ((DISABLE EVAL MEMBER-FIFO-QUEUE PROPER-NODE)))
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0)
         (CLAIM (EQUAL I 0) 0)
         (PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (EQUAL (ADD1 I) (CDADDDR STATEMENT)) 0)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES PROPER-NODES
                         N MEMBER-FIFO-QUEUE))
         (PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
         (REWRITE HELP-PROVE-ENSURES
                  (($STATEMENT
                       (LIST 'NOR-GATE (CT I) (CF I) (TEMP (ADD1 I))))))
         PROVE (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)) S
         (BASH (DISABLE EVAL)) (REWRITE TOTAL-FIFO-QUEUE))))

(PROVE-LEMMA EMPTY-TRUE-NOT-TEMP-MOVES (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION
                 '(PROPER-NODES STATE (QUOTE ,N))
                 (FIFO-QUEUE N))
                (LESSP 1 N)
                (NOT (ZEROP I))
                (LESSP I N))
           (LEADS-TO '(AND (EMPTY-NODE STATE
                                       (QUOTE ,(ADD1 I)))
                           (AND (TRUE-NODE STATE (QUOTE ,I))
                                (NOT (VALUE STATE
                                            (TEMP (QUOTE ,I))))))
                     '(EMPTY-NODE STATE (QUOTE ,I))
                     (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE UNCONDITIONAL-FAIRNESS-GENERAL
                  (($P-1 (LIST 'AND
                               (LIST 'PROPER-NODES 'STATE
                                     (LIST 'QUOTE N))
                               (LIST 'AND
                                     (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE (ADD1 I)))
                                     (LIST 'AND
                                      (LIST 'TRUE-NODE 'STATE
                                       (LIST 'QUOTE I))
                                      (LIST 'NOT
                                       (LIST 'VALUE 'STATE
                                        (LIST 'TEMP (LIST 'QUOTE I))))))))
                   ($Q-1 (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))
                   ($P-2 (LIST 'AND
                               (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE (ADD1 I)))
                               (LIST 'AND
                                     (LIST 'TRUE-NODE 'STATE
                                      (LIST 'QUOTE I))
                                     (LIST 'NOT
                                      (LIST 'VALUE 'STATE
                                       (LIST 'TEMP (LIST 'QUOTE I)))))))
                   ($Q-2 (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))))
         (DROP 1) (REWRITE HELP-PROVE-UNLESS)
         (GENERALIZE
```

```
    (((EU (LIST 'AND
                (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                (LIST 'AND
                      (LIST 'EMPTY-NODE 'STATE
                            (LIST 'QUOTE (ADD1 I)))
                      (LIST 'AND
                            (LIST 'TRUE-NODE 'STATE
                             (LIST 'QUOTE I))
                            (LIST 'NOT
                             (LIST 'VALUE 'STATE
                              (LIST 'TEMP (LIST 'QUOTE I)))))))
          (FIFO-QUEUE N)
          (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))
      STATEMENT)
     ((OLDU (LIST 'AND
                (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                (LIST 'AND
                      (LIST 'EMPTY-NODE 'STATE
                            (LIST 'QUOTE (ADD1 I)))
                      (LIST 'AND
                            (LIST 'TRUE-NODE 'STATE
                             (LIST 'QUOTE I))
                            (LIST 'NOT
                             (LIST 'VALUE 'STATE
                              (LIST 'TEMP (LIST 'QUOTE I)))))))
          (FIFO-QUEUE N)
          (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))
      OLD)
     ((NEWU (LIST 'AND
                (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                (LIST 'AND
                      (LIST 'EMPTY-NODE 'STATE
                            (LIST 'QUOTE (ADD1 I)))
                      (LIST 'AND
                            (LIST 'TRUE-NODE 'STATE
                             (LIST 'QUOTE I))
                            (LIST 'NOT
                             (LIST 'VALUE 'STATE
                              (LIST 'TEMP (LIST 'QUOTE I)))))))
          (FIFO-QUEUE N)
          (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))
      NEW)))
X-DUMB PROMOTE (CLAIM (PROPER-NODE OLD (ADD1 I)) 0)
(CLAIM (PROPER-NODE OLD I) 0) (REWRITE EVAL-OR) (DIVE 1)
(REWRITE EVAL-AND) (DIVE 1) (= * T 0) TOP
(CLAIM (EQUAL (ADD1 I) N) 0)
(CLAIM (NOT (EQUAL I N))
       ((DISABLE MEMBER-FIFO-QUEUE N PROPER-NODES
                 PROPER-NODES-IMPLIES)))
(CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4)
(PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
(CLAIM (NOT (EQUAL (ADD1 I) (CDADDDR STATEMENT)))
       ((DISABLE PROPER-NODE EVAL)))
(CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
(PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
(PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
(CLAIM (NOT (EQUAL I N))
       ((DISABLE EVAL MEMBER-FIFO-QUEUE PROPER-NODE)))
(CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0)
(PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
(CLAIM (EQUAL (ADD1 I) (CDADDDR STATEMENT)) 0)
(PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
(CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
(PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
(PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
(PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES PROPER-NODES
                N MEMBER-FIFO-QUEUE))
(PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
(PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
```

```
        (REWRITE HELP-PROVE-ENSURES
                 (($STATEMENT
                      (LIST 'C-ELEMENT (CT (ADD1 I)) (TEMP I) (CT I)))))
        PROVE (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)) S
        (BASH (DISABLE EVAL)) (REWRITE TOTAL-FIFO-QUEUE))))

(PROVE-LEMMA EMPTY-FALSE-NOT-TEMP-MOVES (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION
                 '(PROPER-NODES STATE (QUOTE ,N))
                 (FIFO-QUEUE N))
                (LESSP 1 N)
                (NOT (ZEROP I))
                (LESSP I N))
          (LEADS-TO '(AND (EMPTY-NODE STATE
                                      (QUOTE ,(ADD1 I)))
                          (AND (FALSE-NODE STATE (QUOTE ,I))
                               (NOT (VALUE STATE
                                           (TEMP (QUOTE ,I))))))
                    '(EMPTY-NODE STATE (QUOTE ,I))
                    (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
        (REWRITE UNCONDITIONAL-FAIRNESS-GENERAL
                 (($P-1 (LIST 'AND
                              (LIST 'PROPER-NODES 'STATE
                                    (LIST 'QUOTE N))
                              (LIST 'AND
                                    (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE (ADD1 I)))
                                    (LIST 'AND
                                     (LIST 'FALSE-NODE 'STATE
                                      (LIST 'QUOTE I))
                                     (LIST 'NOT
                                      (LIST 'VALUE 'STATE
                                       (LIST 'TEMP (LIST 'QUOTE I))))))))
                  ($Q-1 (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))
                  ($P-2 (LIST 'AND
                              (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE (ADD1 I)))
                              (LIST 'AND
                                    (LIST 'FALSE-NODE 'STATE
                                     (LIST 'QUOTE I))
                                    (LIST 'NOT
                                     (LIST 'VALUE 'STATE
                                      (LIST 'TEMP (LIST 'QUOTE I)))))))
                  ($Q-2 (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))))
        (DROP 1) (REWRITE HELP-PROVE-UNLESS)
        (GENERALIZE
            (((EU (LIST 'AND
                        (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                        (LIST 'AND
                              (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE (ADD1 I)))
                              (LIST 'AND
                                    (LIST 'FALSE-NODE 'STATE
                                     (LIST 'QUOTE I))
                                    (LIST 'NOT
                                     (LIST 'VALUE 'STATE
                                      (LIST 'TEMP (LIST 'QUOTE I)))))))
                  (FIFO-QUEUE N)
                  (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))
              STATEMENT)
             ((OLDU (LIST 'AND
                          (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                          (LIST 'AND
                                (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE (ADD1 I)))
                                (LIST 'AND
                                      (LIST 'FALSE-NODE 'STATE
                                       (LIST 'QUOTE I))
```

```
                                    (LIST 'NOT
                                     (LIST 'VALUE 'STATE
                                      (LIST 'TEMP (LIST 'QUOTE I)))))))
                      (FIFO-QUEUE N)
                      (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))
                OLD)
               ((NEWU (LIST 'AND
                            (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                            (LIST 'AND
                                  (LIST 'EMPTY-NODE 'STATE
                                        (LIST 'QUOTE (ADD1 I)))
                                  (LIST 'AND
                                        (LIST 'FALSE-NODE 'STATE
                                         (LIST 'QUOTE I))
                                        (LIST 'NOT
                                         (LIST 'VALUE 'STATE
                                          (LIST 'TEMP (LIST 'QUOTE I)))))))
                      (FIFO-QUEUE N)
                      (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE I)))
                NEW)))
         X-DUMB PROMOTE (CLAIM (PROPER-NODE OLD (ADD1 I)) 0)
         (CLAIM (PROPER-NODE OLD I) 0) (REWRITE EVAL-OR) (DIVE 1)
         (REWRITE EVAL-AND) (DIVE 1) (= * T 0) TOP
         (CLAIM (EQUAL (ADD1 I) N) 0)
         (CLAIM (NOT (EQUAL I N))
                ((DISABLE MEMBER-FIFO-QUEUE N PROPER-NODES
                          PROPER-NODES-IMPLIES)))
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 4)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (NOT (EQUAL (ADD1 I) (CDADDDR STATEMENT)))
                ((DISABLE PROPER-NODE EVAL)))
         (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (NOT (EQUAL I N))
                ((DISABLE EVAL MEMBER-FIFO-QUEUE PROPER-NODE)))
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (EQUAL (ADD1 I) (CDADDDR STATEMENT)) 0)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (CLAIM (EQUAL I (CDADDDR STATEMENT)) 0)
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES))
         (PROVE (DISABLE PROPER-NODE PROPER-NODES-IMPLIES PROPER-NODES
                         N MEMBER-FIFO-QUEUE))
         (PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
         (PROVE (DISABLE MEMBER-FIFO-QUEUE PROPER-NODE PROPER-NODES N))
         (REWRITE HELP-PROVE-ENSURES
                  (($STATEMENT
                       (LIST 'C-ELEMENT (CF (ADD1 I)) (TEMP I) (CF I)))))
         PROVE (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)) S
         (BASH (DISABLE EVAL)) (REWRITE TOTAL-FIFO-QUEUE))))

(PROVE-LEMMA EMPTY-NOT-TEMP-MOVES (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                             `(PROPER-NODES STATE (QUOTE ,N))
                             (FIFO-QUEUE N))
                            (LESSP 1 N)
                            (NOT (ZEROP I))
                            (LESSP I N))
                       (LEADS-TO `(AND (EMPTY-NODE STATE
                                                  (QUOTE ,(ADD1 I)))
                                        (NOT (VALUE STATE
                                                  (TEMP (QUOTE ,I)))))
                                  `(EMPTY-NODE STATE (QUOTE ,I))
                                  (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'OR
```

```
                        (LIST 'AND
                              (LIST 'EMPTY-NODE 'STATE
                               (LIST 'QUOTE (ADD1 I)))
                              (LIST 'AND
                               (LIST 'TRUE-NODE 'STATE
                                (LIST 'QUOTE I))
                               (LIST 'NOT
                                (LIST 'VALUE 'STATE
                                 (LIST 'TEMP (LIST 'QUOTE I))))))
                        (LIST 'OR
                              (LIST 'AND
                               (LIST 'EMPTY-NODE 'STATE
                                (LIST 'QUOTE (ADD1 I)))
                               (LIST 'AND
                                (LIST 'FALSE-NODE 'STATE
                                 (LIST 'QUOTE I))
                                (LIST 'NOT
                                 (LIST 'VALUE 'STATE
                                  (LIST 'TEMP (LIST 'QUOTE I))))))
                              (LIST 'EMPTY-NODE 'STATE
                               (LIST 'QUOTE I)))))))
    (CLAIM (EVAL (LIST 'PROPER-NODE 'STATE (LIST 'QUOTE I))
                 (S (FIFO-QUEUE N)
                    (ILEADS (LIST 'AND
                                  (LIST 'EMPTY-NODE 'STATE
                                   (LIST 'QUOTE (ADD1 I)))
                                  (LIST 'NOT
                                   (LIST 'VALUE 'STATE
                                    (LIST 'TEMP (LIST 'QUOTE I)))))
                            (FIFO-QUEUE N)
                            (LIST 'EMPTY-NODE 'STATE
                                  (LIST 'QUOTE I)))))
           ((DISABLE EVAL)))
    (DROP 1) PROVE (REWRITE DISJOIN-LEFT)
    (REWRITE EMPTY-TRUE-NOT-TEMP-MOVES) (REWRITE DISJOIN-LEFT)
    (REWRITE EMPTY-FALSE-NOT-TEMP-MOVES) (REWRITE Q-LEADS-TO-Q))))

(PROVE-LEMMA EMPTY-1-LEADS-TO-EMPTY-0 (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                            '(PROPER-NODES STATE (QUOTE ,N))
                            (FIFO-QUEUE N))
                           (LESSP 1 N))
                      (LEADS-TO '(EMPTY-NODE STATE (QUOTE ,1))
                                '(EMPTY-NODE STATE (QUOTE ,0))
                                (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
      (REWRITE UNCONDITIONAL-FAIRNESS-GENERAL
               (($P-1 (LIST 'AND
                            (LIST 'PROPER-NODES 'STATE
                                  (LIST 'QUOTE N))
                            (LIST 'EMPTY-NODE 'STATE
                                  (LIST 'QUOTE 1))))
                ($Q-1 (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 0)))
                ($P-2 (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 1)))
                ($Q-2 (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 0)))))
      (REWRITE HELP-PROVE-UNLESS)
      (GENERALIZE
          (((EU (LIST 'AND
                      (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                      (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 1)))
                (FIFO-QUEUE N)
                (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 0)))
            STATEMENT)
           ((OLDU (LIST 'AND
                        (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                        (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 1)))
                  (FIFO-QUEUE N)
                  (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 0)))
            OLD)
```

```
              ((NEWU (LIST 'AND
                           (LIST 'PROPER-NODES 'STATE (LIST 'QUOTE N))
                           (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 1)))
                     (FIFO-QUEUE N)
                     (LIST 'EMPTY-NODE 'STATE (LIST 'QUOTE 0)))
               NEW)))
         (DROP 1) X-DUMB PROMOTE (REWRITE EVAL-OR) (DIVE 1)
         (REWRITE EVAL-AND) (DIVE 1)
         (= * T ((DISABLE N MEMBER-FIFO-QUEUE))) TOP
         (CLAIM (PROPER-NODE OLD 2)
                ((DISABLE PROPER-NODE MEMBER-FIFO-QUEUE N)))
         (CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N)) 0) (DROP 2)
         (CLAIM (NOT (MEMBER N '(1 0)))) (HIDE-HYPS 5)
         (PROVE (DISABLE PROPER-NODES))
         (CLAIM (EQUAL (CDADDDR STATEMENT) 1) 0)
         (CLAIM (PROPER-NODE OLD 1) 0) CHANGE-GOAL (CONTRADICT 9)
         (DROP 2 3 5 6 7 8 9) (REWRITE PROPER-NODES-IMPLIES) PROVE
         PROVE
         (PROVE (DISABLE PROPER-NODES PROPER-NODES-IMPLIES
                         EXTERNAL-NODES))
         (DROP 6) (PROVE (DISABLE PROPER-NODES EXTERNAL-NODES))
         (REWRITE HELP-PROVE-ENSURES (($STATEMENT '(OUT-NODE))))
         (DROP 1) PROVE (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)) S
         (BASH (DISABLE EVAL)) (REWRITE TOTAL-FIFO-QUEUE))))

(PROVE-LEMMA FULL-EMPTY-EMPTY-MOVES-FORWARD (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION
                 '(PROPER-NODES STATE (QUOTE ,N))
                 (FIFO-QUEUE N))
                (LESSP 1 N)
                (NOT (LESSP N (ADD1 (ADD1 I))))
                (NUMBERP I))
           (LEADS-TO
            '(AND (NOT (EMPTY-NODE
                        STATE (QUOTE ,(ADD1 (ADD1 I)))))
                  (AND (EMPTY-NODE STATE (QUOTE ,(ADD1 I)))
                       (AND (EMPTY-NODE STATE (QUOTE ,I))
                            (NOT (LISTP (QUEUE-VALUES
                                         STATE
                                         (QUOTE ,(ADD1 I))))))))
            '(AND (NOT (EMPTY-NODE STATE (QUOTE ,(ADD1 I))))
                  (EMPTY-NODE STATE (QUOTE ,I)))
            (FIFO-QUEUE N)))
 ((INSTRUCTIONS PROMOTE
  (REWRITE PSP-GENERAL
   (($P
     (LIST 'AND
         (LIST 'NOT
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE (ADD1 (ADD1 I)))))
         (LIST 'AND
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE (ADD1 I)))
               (LIST 'AND
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE I))
                     (LIST 'NOT
                           (LIST 'LISTP
                                 (LIST 'QUEUE-VALUES
                                       'STATE
                                       (LIST 'QUOTE (ADD1 I)))))))))
    ($Q (LIST 'OR
              (LIST 'NOT
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE (ADD1 I))))
```

```
            (LIST 'AND
                  (LIST 'NOT
                        (LIST 'EMPTY-NODE
                              'STATE
                              (LIST 'QUOTE (ADD1 I))))
                  (LIST 'AND
                        (LIST 'EMPTY-NODE
                              'STATE
                              (LIST 'QUOTE I))
                        (LIST 'NOT
                              (LIST 'LISTP
                                    (LIST 'QUEUE-VALUES
                                          'STATE
                                          (LIST 'QUOTE I))))))))
  ($R
   (LIST 'AND
       (LIST 'PROPER-NODES
             'STATE
             (LIST 'QUOTE N))
       (LIST 'AND
             (LIST 'NOT
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 (ADD1 I)))))
             (LIST 'AND
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 I)))
                   (LIST 'NOT
                         (LIST 'LISTP
                               (LIST 'QUEUE-VALUES
                                     'STATE
                                     (LIST 'QUOTE (ADD1 I)))))))))
  ($B
   (LIST 'AND
     (LIST 'NOT
           (LIST 'EMPTY-NODE
                 'STATE
                 (LIST 'QUOTE (ADD1 I))))
     (LIST 'AND
           (LIST 'EMPTY-NODE
                 'STATE
                 (LIST 'QUOTE (SUB1 (ADD1 I))))
           (LIST 'NOT
                 (LIST 'LISTP
                       (LIST 'QUEUE-VALUES
                             'STATE
                             (LIST 'QUOTE (SUB1 (ADD1 I))))))))))
(REWRITE CANCELLATION-LEADS-TO-GENERAL
 (($P-1
   (LIST 'AND
    (LIST 'AND
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE (ADD1 I)))
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE I)))
    (LIST 'AND
      (LIST 'PROPER-NODES
            'STATE
            (LIST 'QUOTE N))
      (LIST 'AND
            (LIST 'NOT
                  (LIST 'EMPTY-NODE
                        'STATE
                        (LIST 'QUOTE (ADD1 (ADD1 I)))))
            (LIST 'AND
                  (LIST 'EMPTY-NODE
```

```
                         'STATE
                         (LIST 'QUOTE (ADD1 I)))
                   (LIST 'NOT
                         (LIST 'LISTP
                               (LIST 'QUEUE-VALUES
                                     'STATE
                                     (LIST 'QUOTE (ADD1 I))))))))))
  ($D
   (LIST 'OR
    (LIST 'AND
     (LIST 'VALUE
           'STATE
           (LIST 'TEMP (LIST 'QUOTE (ADD1 I))))
     (LIST 'AND
       (LIST 'PROPER-NODES
             'STATE
             (LIST 'QUOTE N))
       (LIST 'AND
             (LIST 'NOT
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 (ADD1 I)))))
             (LIST 'AND
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 I)))
                   (LIST 'NOT
                         (LIST 'LISTP
                               (LIST 'QUEUE-VALUES
                                     'STATE
                                     (LIST 'QUOTE (ADD1 I)))))))))
    (LIST 'AND
      (LIST 'NOT
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (ADD1 I))))
      (LIST 'AND
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (SUB1 (ADD1 I))))
            (LIST 'NOT
                  (LIST 'LISTP
                        (LIST 'QUEUE-VALUES
                              'STATE
                              (LIST 'QUOTE (SUB1 (ADD1 I))))))))))
  ($B (LIST 'AND
            (LIST 'NOT
                  (LIST 'EMPTY-NODE
                        'STATE
                        (LIST 'QUOTE (ADD1 (ADD1 I)))))
            (LIST 'VALUE
                  'STATE
                  (LIST 'TEMP
                        (LIST 'QUOTE (ADD1 I))))))
  ($R-1 (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I))))))
(REWRITE PSP)
(REWRITE EMPTY-EMPTY-SETS-TEMP)
PROVE
(REWRITE FULL-EMPTY-REST-UNLESS-MOVES-FORWARD)
PROVE PROVE
(REWRITE NOT-EMPTY-EMPTY-TEMP-MOVES)
PROVE PROVE
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (JLEADS
```

```
(ILEADS
 (LIST 'AND
   (LIST 'NOT
         (LIST 'EMPTY-NODE
               'STATE
               (LIST 'QUOTE (ADD1 (ADD1 I)))))
   (LIST 'AND
         (LIST 'EMPTY-NODE
               'STATE
               (LIST 'QUOTE (ADD1 I)))
         (LIST 'AND
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE I))
               (LIST 'NOT
                     (LIST 'LISTP
                           (LIST 'QUEUE-VALUES
                                 'STATE
                                 (LIST 'QUOTE (ADD1 I))))))))
 (FIFO-QUEUE N)
 (LIST 'OR
       (LIST 'NOT
             (LIST 'EMPTY-NODE
                   'STATE
                   (LIST 'QUOTE (ADD1 I))))
       (LIST 'AND
             (LIST 'NOT
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 I))))
             (LIST 'AND
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE I))
                   (LIST 'NOT
                         (LIST 'LISTP
                               (LIST 'QUEUE-VALUES
                                     'STATE
                                     (LIST 'QUOTE I))))))))
(FIFO-QUEUE N)
(LIST 'OR
 (LIST 'AND
  (LIST 'VALUE
        'STATE
        (LIST 'TEMP (LIST 'QUOTE (ADD1 I))))
  (LIST 'AND
   (LIST 'PROPER-NODES
         'STATE
         (LIST 'QUOTE N))
   (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'NOT
                    (LIST 'LISTP
                          (LIST 'QUEUE-VALUES
                                'STATE
                                (LIST 'QUOTE (ADD1 I)))))))))
 (LIST 'AND
  (LIST 'NOT
        (LIST 'EMPTY-NODE
              'STATE
              (LIST 'QUOTE (ADD1 I))))
  (LIST 'AND
```

```
           (LIST 'EMPTY-NODE
                 'STATE
                 (LIST 'QUOTE (SUB1 (ADD1 I))))
           (LIST 'NOT
                 (LIST 'LISTP
                       (LIST 'QUEUE-VALUES
                             'STATE
                             (LIST 'QUOTE (SUB1 (ADD1 I))))))))))
   STATE)))
(DROP 1)
(BASH (DISABLE EVAL))
(BASH (DISABLE EVAL))
(CLAIM
 (EVAL
  (LIST 'PROPER-NODES
        'STATE
        (LIST 'QUOTE N))
  (S
   (FIFO-QUEUE N)
   (ILEADS
    (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE I))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))
    (FIFO-QUEUE N)
    (LIST 'OR
          (LIST 'NOT
                (LIST 'EMPTY-NODE
                      'STATE
                      (LIST 'QUOTE (ADD1 I))))
          (LIST 'AND
                (LIST 'NOT
                      (LIST 'EMPTY-NODE
                            'STATE
                            (LIST 'QUOTE (ADD1 I))))
                (LIST 'AND
                      (LIST 'EMPTY-NODE
                            'STATE
                            (LIST 'QUOTE I))
                      (LIST 'NOT
                            (LIST 'LISTP
                                  (LIST 'QUEUE-VALUES
                                        'STATE
                                        (LIST 'QUOTE I)))))))))
 0)
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (ILEADS
     (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
```

```
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE I))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))
     (FIFO-QUEUE N)
     (LIST 'OR
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 I))))
           (LIST 'AND
                 (LIST 'NOT
                       (LIST 'EMPTY-NODE
                             'STATE
                             (LIST 'QUOTE (ADD1 I))))
                 (LIST 'AND
                       (LIST 'EMPTY-NODE
                             'STATE
                             (LIST 'QUOTE I))
                       (LIST 'NOT
                             (LIST 'LISTP
                                   (LIST 'QUEUE-VALUES
                                         'STATE
                                         (LIST 'QUOTE I)))))))))
   STATE)))
(DROP 1)
(BASH (DISABLE EVAL))
(CONTRADICT 5)
(REWRITE INVARIANT-IMPLIES)
(REWRITE PROPER-NODES-INVARIANT)
(REWRITE FULL-EMPTY-REST-UNLESS-MOVES-FORWARD)
PROVE PROVE
(CLAIM
 (EVAL
  (LIST 'PROPER-NODES
        'STATE
        (LIST 'QUOTE N))
  (S
   (FIFO-QUEUE N)
   (ILEADS
    (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE I))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))
    (FIFO-QUEUE N)
    (LIST 'AND
          (LIST 'NOT
```

```
                (LIST 'EMPTY-NODE
                      'STATE
                      (LIST 'QUOTE (ADD1 I))))
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE I))))))
 0)
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (ILEADS
     (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE I))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))
     (FIFO-QUEUE N)
     (LIST 'AND
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 I))))
           (LIST 'EMPTY-NODE
                 'STATE
                 (LIST 'QUOTE I)))))
  STATE)))
(DROP 1)
(BASH (DISABLE EVAL))
(CONTRADICT 5)
(REWRITE INVARIANT-IMPLIES)
(REWRITE PROPER-NODES-INVARIANT)
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (JLEADS
     (ILEADS
      (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE I))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))
      (FIFO-QUEUE N)
      (LIST 'AND
```

```
               (LIST 'NOT
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE (ADD1 I))))
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE I))))
        (FIFO-QUEUE N)
        (LIST 'OR
         (LIST 'AND
          (LIST 'OR
                (LIST 'NOT
                      (LIST 'EMPTY-NODE
                            'STATE
                            (LIST 'QUOTE (ADD1 I))))
                (LIST 'AND
                      (LIST 'NOT
                            (LIST 'EMPTY-NODE
                                  'STATE
                                  (LIST 'QUOTE (ADD1 I))))
                      (LIST 'AND
                            (LIST 'EMPTY-NODE
                                  'STATE
                                  (LIST 'QUOTE I))
                            (LIST 'NOT
                                  (LIST 'LISTP
                                        (LIST 'QUEUE-VALUES
                                              'STATE
                                              (LIST 'QUOTE I)))))))
          (LIST 'AND
           (LIST 'PROPER-NODES
                 'STATE
                 (LIST 'QUOTE N))
           (LIST 'AND
                (LIST 'NOT
                      (LIST 'EMPTY-NODE
                            'STATE
                            (LIST 'QUOTE (ADD1 (ADD1 I)))))
                (LIST 'AND
                      (LIST 'EMPTY-NODE
                            'STATE
                            (LIST 'QUOTE (ADD1 I)))
                      (LIST 'NOT
                            (LIST 'LISTP
                                  (LIST 'QUEUE-VALUES
                                        'STATE
                                        (LIST 'QUOTE (ADD1 I)))))))))
         (LIST 'AND
          (LIST 'NOT
                (LIST 'EMPTY-NODE
                      'STATE
                      (LIST 'QUOTE (ADD1 I))))
          (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (SUB1 (ADD1 I))))
              (LIST 'NOT
                    (LIST 'LISTP
                          (LIST 'QUEUE-VALUES
                                'STATE
                                (LIST 'QUOTE (SUB1 (ADD1 I))))))))))
      STATE)))
   (DROP 1)
   (BASH (DISABLE EVAL)))))

(PROVE-LEMMA FULL-EMPTY-FULL-MOVES-FORWARD (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION
                 `(PROPER-NODES STATE (QUOTE ,N))
```

```
                  (FIFO-QUEUE N))
                 (LESSP 1 N)
                 (NOT (LESSP N (ADD1 (ADD1 I))))
                 (NUMBERP I))
            (LEADS-TO
             '(AND (NOT (EMPTY-NODE
                         STATE (QUOTE ,(ADD1 (ADD1 I)))))
                   (AND (EMPTY-NODE STATE (QUOTE ,(ADD1 I)))
                        (AND (NOT (EMPTY-NODE STATE (QUOTE ,I)))
                             (NOT (LISTP (QUEUE-VALUES
                                          STATE
                                          (QUOTE ,(ADD1 I)))))))))
             '(AND (NOT (EMPTY-NODE STATE (QUOTE ,(ADD1 I))))
                   (EMPTY-NODE STATE (QUOTE ,I)))
             (FIFO-QUEUE N)))
((INSTRUCTIONS PROMOTE
  (REWRITE CANCELLATION-LEADS-TO-GENERAL
   (($P-1
     (LIST 'AND
      (LIST 'AND
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (ADD1 I)))
            (LIST 'AND
                  (LIST 'NOT
                        (LIST 'EMPTY-NODE
                              'STATE
                              (LIST 'QUOTE I)))
                  (LIST 'NOT
                        (LIST 'LISTP
                              (LIST 'QUEUE-VALUES
                                    'STATE
                                    (LIST 'QUOTE (ADD1 I)))))))
      (LIST 'AND
        (LIST 'PROPER-NODES
              'STATE
              (LIST 'QUOTE N))
        (LIST 'AND
              (LIST 'NOT
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE (ADD1 (ADD1 I)))))
              (LIST 'AND
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE (ADD1 I)))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))))
    ($D
     (LIST 'OR
      (LIST 'AND
       (LIST 'EMPTY-NODE
             'STATE
             (LIST 'QUOTE I))
       (LIST 'AND
         (LIST 'PROPER-NODES
               'STATE
               (LIST 'QUOTE N))
         (LIST 'AND
               (LIST 'NOT
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE (ADD1 (ADD1 I)))))
               (LIST 'AND
                     (LIST 'EMPTY-NODE
                           'STATE
```

```
                         (LIST 'QUOTE (ADD1 I)))
                   (LIST 'NOT
                         (LIST 'LISTP
                               (LIST 'QUEUE-VALUES
                                     'STATE
                                     (LIST 'QUOTE (ADD1 I)))))))))
    (LIST 'AND
      (LIST 'NOT
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (ADD1 I))))
      (LIST 'AND
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (SUB1 (ADD1 I))))
            (LIST 'NOT
                  (LIST 'LISTP
                        (LIST 'QUEUE-VALUES
                              'STATE
                              (LIST 'QUOTE (SUB1 (ADD1 I))))))))))
  ($B
   (LIST 'AND
       (LIST 'NOT
             (LIST 'EMPTY-NODE
                   'STATE
                   (LIST 'QUOTE (ADD1 (ADD1 I)))))
       (LIST 'AND
             (LIST 'EMPTY-NODE
                   'STATE
                   (LIST 'QUOTE (ADD1 I)))
             (LIST 'AND
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE I))
                   (LIST 'NOT
                         (LIST 'LISTP
                               (LIST 'QUEUE-VALUES
                                     'STATE
                                     (LIST 'QUOTE (ADD1 I)))))))))
  ($R-1 (LIST 'AND
              (LIST 'NOT
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE (ADD1 I))))
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE I))))))
CHANGE-GOAL
(REWRITE FULL-EMPTY-EMPTY-MOVES-FORWARD)
CHANGE-GOAL
(CLAIM
 (EVAL
  (LIST 'PROPER-NODES
        'STATE
        (LIST 'QUOTE N))
  (S
   (FIFO-QUEUE N)
   (JLEADS
    (ILEADS
     (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
```

```
                    (LIST 'NOT
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE I)))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))
     (FIFO-QUEUE N)
     (LIST 'AND
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 I))))
           (LIST 'EMPTY-NODE
                 'STATE
                 (LIST 'QUOTE I))))
    (FIFO-QUEUE N)
    (LIST 'OR
     (LIST 'AND
      (LIST 'EMPTY-NODE
            'STATE
            (LIST 'QUOTE I))
      (LIST 'AND
       (LIST 'PROPER-NODES
             'STATE
             (LIST 'QUOTE N))
       (LIST 'AND
             (LIST 'NOT
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 (ADD1 I)))))
             (LIST 'AND
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 I)))
                   (LIST 'NOT
                         (LIST 'LISTP
                               (LIST 'QUEUE-VALUES
                                     'STATE
                                     (LIST 'QUOTE (ADD1 I)))))))))
     (LIST 'AND
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 I))))
           (LIST 'AND
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (SUB1 (ADD1 I))))
                 (LIST 'NOT
                       (LIST 'LISTP
                             (LIST 'QUEUE-VALUES
                                   'STATE
                                   (LIST 'QUOTE
                                         (SUB1 (ADD1 I)))))))))))
 0)
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (JLEADS
     (ILEADS
      (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
```

```
                (LIST 'EMPTY-NODE
                      'STATE
                      (LIST 'QUOTE (ADD1 I)))
                (LIST 'AND
                      (LIST 'NOT
                            (LIST 'EMPTY-NODE
                                  'STATE
                                  (LIST 'QUOTE I)))
                      (LIST 'NOT
                            (LIST 'LISTP
                                  (LIST 'QUEUE-VALUES
                                        'STATE
                                        (LIST 'QUOTE (ADD1 I))))))))
        (FIFO-QUEUE N)
        (LIST 'AND
              (LIST 'NOT
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE (ADD1 I))))
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE I))))
       (FIFO-QUEUE N)
       (LIST 'OR
        (LIST 'AND
         (LIST 'EMPTY-NODE
               'STATE
               (LIST 'QUOTE I))
         (LIST 'AND
          (LIST 'PROPER-NODES
                'STATE
                (LIST 'QUOTE N))
          (LIST 'AND
               (LIST 'NOT
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE (ADD1 (ADD1 I)))))
               (LIST 'AND
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE (ADD1 I)))
                     (LIST 'NOT
                           (LIST 'LISTP
                                 (LIST 'QUEUE-VALUES
                                       'STATE
                                       (LIST 'QUOTE (ADD1 I)))))))))
        (LIST 'AND
         (LIST 'NOT
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE (ADD1 I))))
         (LIST 'AND
             (LIST 'EMPTY-NODE
                   'STATE
                   (LIST 'QUOTE (SUB1 (ADD1 I))))
             (LIST 'NOT
                   (LIST 'LISTP
                         (LIST 'QUEUE-VALUES
                               'STATE
                               (LIST 'QUOTE (SUB1 (ADD1 I))))))))))
   STATE)))
(BASH (DISABLE EVAL))
(CONTRADICT 5)
(REWRITE INVARIANT-IMPLIES)
(REWRITE PROPER-NODES-INVARIANT)
CHANGE-GOAL
(CLAIM
 (EVAL
  (LIST 'PROPER-NODES
```

```
      'STATE
      (LIST 'QUOTE N))
(S
 (FIFO-QUEUE N)
 (JLEADS
  (JLEADS
   (ILEADS
    (LIST 'AND
      (LIST 'NOT
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (ADD1 (ADD1 I)))))
      (LIST 'AND
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (ADD1 I)))
            (LIST 'AND
                  (LIST 'NOT
                        (LIST 'EMPTY-NODE
                              'STATE
                              (LIST 'QUOTE I)))
                  (LIST 'NOT
                        (LIST 'LISTP
                              (LIST 'QUEUE-VALUES
                                    'STATE
                                    (LIST 'QUOTE (ADD1 I))))))))
    (FIFO-QUEUE N)
    (LIST 'AND
          (LIST 'NOT
                (LIST 'EMPTY-NODE
                      'STATE
                      (LIST 'QUOTE (ADD1 I))))
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE I))))
   (FIFO-QUEUE N)
   (LIST 'OR
    (LIST 'AND
     (LIST 'EMPTY-NODE
           'STATE
           (LIST 'QUOTE I))
     (LIST 'AND
      (LIST 'PROPER-NODES
            'STATE
            (LIST 'QUOTE N))
      (LIST 'AND
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 (ADD1 I)))))
           (LIST 'AND
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 I)))
                 (LIST 'NOT
                       (LIST 'LISTP
                             (LIST 'QUEUE-VALUES
                                   'STATE
                                   (LIST 'QUOTE (ADD1 I)))))))))
    (LIST 'AND
     (LIST 'NOT
           (LIST 'EMPTY-NODE
                 'STATE
                 (LIST 'QUOTE (ADD1 I))))
     (LIST 'AND
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE (SUB1 (ADD1 I))))
          (LIST 'NOT
```

```
                  (LIST 'LISTP
                        (LIST 'QUEUE-VALUES
                              'STATE
                              (LIST 'QUOTE (SUB1 (ADD1 I))))))))))
    (FIFO-QUEUE N)
    (LIST 'AND
          (LIST 'NOT
                (LIST 'EMPTY-NODE
                      'STATE
                      (LIST 'QUOTE (ADD1 I))))
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE I))))))
 0)
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (JLEADS
     (JLEADS
      (ILEADS
       (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
                    (LIST 'NOT
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE I)))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I)))))))))
       (FIFO-QUEUE N)
       (LIST 'AND
             (LIST 'NOT
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 I))))
             (LIST 'EMPTY-NODE
                   'STATE
                   (LIST 'QUOTE I))))
      (FIFO-QUEUE N)
      (LIST 'OR
       (LIST 'AND
        (LIST 'EMPTY-NODE
              'STATE
              (LIST 'QUOTE I))
        (LIST 'AND
         (LIST 'PROPER-NODES
               'STATE
               (LIST 'QUOTE N))
         (LIST 'AND
             (LIST 'NOT
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 (ADD1 I)))))
             (LIST 'AND
                   (LIST 'EMPTY-NODE
                         'STATE
                         (LIST 'QUOTE (ADD1 I)))
                   (LIST 'NOT
                         (LIST 'LISTP
```

```
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I)))))))))
       (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I))))
        (LIST 'AND
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (SUB1 (ADD1 I))))
            (LIST 'NOT
                  (LIST 'LISTP
                        (LIST 'QUEUE-VALUES
                              'STATE
                              (LIST 'QUOTE (SUB1 (ADD1 I))))))))))
     (FIFO-QUEUE N)
     (LIST 'AND
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 I))))
           (LIST 'EMPTY-NODE
                 'STATE
                 (LIST 'QUOTE I)))))
   STATE)))
(BASH (DISABLE EVAL))
(CONTRADICT 5)
(REWRITE INVARIANT-IMPLIES)
(REWRITE PROPER-NODES-INVARIANT)
CHANGE-GOAL
(CLAIM
 (EVAL
  (LIST 'PROPER-NODES
        'STATE
        (LIST 'QUOTE N))
  (S
   (FIFO-QUEUE N)
   (ILEADS
    (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
                    (LIST 'NOT
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE I)))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))
    (FIFO-QUEUE N)
    (LIST 'AND
          (LIST 'NOT
                (LIST 'EMPTY-NODE
                      'STATE
                      (LIST 'QUOTE (ADD1 I))))
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE I))))))
 0)
```

```
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (ILEADS
     (LIST 'AND
        (LIST 'NOT
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 (ADD1 I)))))
        (LIST 'AND
              (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE (ADD1 I)))
              (LIST 'AND
                    (LIST 'NOT
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE I)))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I))))))))
     (FIFO-QUEUE N)
     (LIST 'AND
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 I))))
           (LIST 'EMPTY-NODE
                 'STATE
                 (LIST 'QUOTE I)))))
   STATE)))
(BASH (DISABLE EVAL))
(CONTRADICT 5)
(REWRITE INVARIANT-IMPLIES)
(REWRITE PROPER-NODES-INVARIANT)
(REWRITE PSP)
CHANGE-GOAL
(REWRITE FULL-EMPTY-REST-UNLESS-MOVES-FORWARD)
PROVE PROVE
(CLAIM (EQUAL I 0) 0)
(REWRITE LEADS-TO-STRENGTHEN-LEFT
         (($P (LIST 'EMPTY-NODE
                    'STATE
                    (LIST 'QUOTE 1)))))
(DROP 1 2 3 4)
(BASH (DISABLE EVAL))
BASH
(REWRITE CANCELLATION-LEADS-TO-GENERAL
 (($P-1 (LIST 'AND
              (LIST 'AND
                    (LIST 'NOT
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE (ADD1 (SUB1 I)))))
                    (LIST 'NOT
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE (SUB1 I)))))
              (LIST 'AND
                    (LIST 'PROPER-NODES
                          'STATE
                          (LIST 'QUOTE N))
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE (ADD1 I))))))
  ($D
    (LIST 'OR
```

```
          (LIST 'AND
                (LIST 'NOT
                      (LIST 'VALUE
                            'STATE
                            (LIST 'TEMP
                                  (LIST 'QUOTE (ADD1 (SUB1 I))))))
                (LIST 'AND
                      (LIST 'PROPER-NODES
                            'STATE
                            (LIST 'QUOTE N))
                      (LIST 'EMPTY-NODE
                            'STATE
                            (LIST 'QUOTE (ADD1 I)))))
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE I))))
  ($B (LIST 'AND
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE (ADD1 I)))
            (LIST 'NOT
                  (LIST 'VALUE
                        'STATE
                        (LIST 'TEMP (LIST 'QUOTE I))))))
  ($R-1 (LIST 'EMPTY-NODE
              'STATE
              (LIST 'QUOTE I)))))
(REWRITE PSP)
(REWRITE FULL-FULL-UNSETS-TEMP)
PROVE CHANGE-GOAL
(REWRITE EMPTY-NOT-TEMP-MOVES)
PROVE CHANGE-GOAL
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (JLEADS
     (ILEADS
         (LIST 'AND
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE (ADD1 I)))
               (LIST 'AND
                     (LIST 'NOT
                           (LIST 'EMPTY-NODE
                                 'STATE
                                 (LIST 'QUOTE I)))
                     (LIST 'NOT
                           (LIST 'LISTP
                                 (LIST 'QUEUE-VALUES
                                       'STATE
                                       (LIST 'QUOTE (ADD1 I)))))))
         (FIFO-QUEUE N)
         (LIST 'EMPTY-NODE
               'STATE
               (LIST 'QUOTE I)))
     (FIFO-QUEUE N)
     (LIST 'OR
          (LIST 'AND
                (LIST 'NOT
                      (LIST 'VALUE
                            'STATE
                            (LIST 'TEMP
                                  (LIST 'QUOTE (ADD1 (SUB1 I))))))
                (LIST 'AND
                      (LIST 'PROPER-NODES
                            'STATE
                            (LIST 'QUOTE N))
                      (LIST 'EMPTY-NODE
                            'STATE
```

```
                              (LIST 'QUOTE (ADD1 I)))))
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE I)))))
   STATE)))
(BASH (DISABLE EVAL))
CHANGE-GOAL
(CLAIM
 (EVAL
  (LIST 'PROPER-NODES
        'STATE
        (LIST 'QUOTE N))
  (S
   (FIFO-QUEUE N)
   (JLEADS
    (JLEADS
     (ILEADS
         (LIST 'AND
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE (ADD1 I)))
               (LIST 'AND
                     (LIST 'NOT
                           (LIST 'EMPTY-NODE
                                 'STATE
                                 (LIST 'QUOTE I)))
                     (LIST 'NOT
                           (LIST 'LISTP
                                 (LIST 'QUEUE-VALUES
                                       'STATE
                                       (LIST 'QUOTE (ADD1 I)))))))
         (FIFO-QUEUE N)
         (LIST 'EMPTY-NODE
               'STATE
               (LIST 'QUOTE I)))
     (FIFO-QUEUE N)
     (LIST 'OR
          (LIST 'AND
                (LIST 'NOT
                      (LIST 'VALUE
                            'STATE
                            (LIST 'TEMP
                                  (LIST 'QUOTE (ADD1 (SUB1 I))))))
                (LIST 'AND
                      (LIST 'PROPER-NODES
                            'STATE
                            (LIST 'QUOTE N))
                      (LIST 'EMPTY-NODE
                            'STATE
                            (LIST 'QUOTE (ADD1 I)))))
          (LIST 'EMPTY-NODE
                'STATE
                (LIST 'QUOTE I))))
    (FIFO-QUEUE N)
    (LIST 'EMPTY-NODE
          'STATE
          (LIST 'QUOTE I)))))
 0)
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (JLEADS
     (JLEADS
      (ILEADS
         (LIST 'AND
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE (ADD1 I)))
               (LIST 'AND
```

```
                    (LIST 'NOT
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE I)))
                    (LIST 'NOT
                          (LIST 'LISTP
                                (LIST 'QUEUE-VALUES
                                      'STATE
                                      (LIST 'QUOTE (ADD1 I)))))))
        (FIFO-QUEUE N)
        (LIST 'EMPTY-NODE
              'STATE
              (LIST 'QUOTE I)))
     (FIFO-QUEUE N)
     (LIST 'OR
         (LIST 'AND
               (LIST 'NOT
                     (LIST 'VALUE
                           'STATE
                           (LIST 'TEMP
                                 (LIST 'QUOTE (ADD1 (SUB1 I))))))
               (LIST 'AND
                     (LIST 'PROPER-NODES
                           'STATE
                           (LIST 'QUOTE N))
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE (ADD1 I)))))
         (LIST 'EMPTY-NODE
               'STATE
               (LIST 'QUOTE I))))
    (FIFO-QUEUE N)
    (LIST 'EMPTY-NODE
          'STATE
          (LIST 'QUOTE I))))
  STATE)))
(BASH (DISABLE EVAL))
(CONTRADICT 6)
(REWRITE INVARIANT-IMPLIES)
(REWRITE PROPER-NODES-INVARIANT)
CHANGE-GOAL
(CLAIM
 (EVAL
  (LIST 'PROPER-NODES
        'STATE
        (LIST 'QUOTE N))
  (S
   (FIFO-QUEUE N)
   (ILEADS
         (LIST 'AND
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE (ADD1 I)))
               (LIST 'AND
                     (LIST 'NOT
                           (LIST 'EMPTY-NODE
                                 'STATE
                                 (LIST 'QUOTE I)))
                     (LIST 'NOT
                           (LIST 'LISTP
                                 (LIST 'QUEUE-VALUES
                                       'STATE
                                       (LIST 'QUOTE (ADD1 I)))))))
         (FIFO-QUEUE N)
         (LIST 'EMPTY-NODE
               'STATE
               (LIST 'QUOTE I)))))
 0)
(GENERALIZE
```

```
 (((S
    (FIFO-QUEUE N)
    (ILEADS
         (LIST 'AND
               (LIST 'EMPTY-NODE
                     'STATE
                     (LIST 'QUOTE (ADD1 I)))
               (LIST 'AND
                     (LIST 'NOT
                           (LIST 'EMPTY-NODE
                                 'STATE
                                 (LIST 'QUOTE I)))
                     (LIST 'NOT
                           (LIST 'LISTP
                                 (LIST 'QUEUE-VALUES
                                       'STATE
                                       (LIST 'QUOTE (ADD1 I)))))))
         (FIFO-QUEUE N)
         (LIST 'EMPTY-NODE
               'STATE
               (LIST 'QUOTE I))))
   STATE)))
(BASH (DISABLE EMPTY-NODE))
(CONTRADICT 6)
(REWRITE INVARIANT-IMPLIES)
(REWRITE PROPER-NODES-INVARIANT)
(REWRITE HELP-PROVE-UNLESS)
(GENERALIZE (((EU (LIST 'AND
                        (LIST 'PROPER-NODES
                              'STATE
                              (LIST 'QUOTE N))
                        (LIST 'EMPTY-NODE
                              'STATE
                              (LIST 'QUOTE (ADD1 I))))
                  (FIFO-QUEUE N)
                  (LIST 'EMPTY-NODE
                        'STATE
                        (LIST 'QUOTE I)))
              STATEMENT)
             ((OLDU (LIST 'AND
                          (LIST 'PROPER-NODES
                                'STATE
                                (LIST 'QUOTE N))
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE (ADD1 I))))
                    (FIFO-QUEUE N)
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE I)))
              OLD)
             ((NEWU (LIST 'AND
                          (LIST 'PROPER-NODES
                                'STATE
                                (LIST 'QUOTE N))
                          (LIST 'EMPTY-NODE
                                'STATE
                                (LIST 'QUOTE (ADD1 I))))
                    (FIFO-QUEUE N)
                    (LIST 'EMPTY-NODE
                          'STATE
                          (LIST 'QUOTE I)))
              NEW)))
(DROP 1)
X-DUMB PROMOTE
(BASH (DISABLE MEMBER-FIFO-QUEUE N EMPTY-NODE))
PROMOTE
(CLAIM (MEMBER STATEMENT (EXTERNAL-NODES N))
       0)
```

```
  (DROP 8)
  (DROP 9)
  (CLAIM (NOT (EQUAL I N)))
  (CLAIM (NOT (EQUAL (ADD1 I) N)))
  (CLAIM (NOT (EQUAL (ADD1 I) 0)))
  (BASH T)
  (CLAIM (EQUAL (CDADDDR STATEMENT) I)
         0)
  (DROP 10)
  (DEMOTE 8)
  (DIVE 1)
  (REWRITE MEMBER-FIFO-QUEUE)
  (DIVE 3)
  (= F)
  TOP
  (DROP 12)
  (BASH T)
  (CLAIM (EQUAL (CDADDDR STATEMENT) (ADD1 I))
         0)
  (CLAIM (PROPER-NODE OLD (ADD1 I))
         ((DISABLE PROPER-NODE EMPTY-NODE N MEMBER-FIFO-QUEUE
                   EXTERNAL-NODES PROPER-NODES)))
  (DEMOTE 8)
  (DIVE 1)
  (REWRITE MEMBER-FIFO-QUEUE)
  (DIVE 3)
  (= F)
  TOP
  (DROP 9 13)
  (BASH T)
  (DEMOTE 8)
  (DIVE 1)
  (REWRITE MEMBER-FIFO-QUEUE)
  (DIVE 3)
  (= F)
  TOP
  (DROP 9 13)
  (BASH T))))


(PROVE-LEMMA FULL-EMPTY-MOVES-FORWARD (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                             '(PROPER-NODES STATE (QUOTE ,N))
                             (FIFO-QUEUE N))
                            (LESSP 1 N)
                            (NOT (LESSP N (ADD1 (ADD1 I))))
                            (NUMBERP I))
                       (LEADS-TO
                        '(AND (NOT (EMPTY-NODE
                                     STATE (QUOTE ,(ADD1 (ADD1 I)))))
                              (AND (EMPTY-NODE STATE (QUOTE ,(ADD1 I)))
                                   (NOT (LISTP (QUEUE-VALUES
                                                 STATE
                                                 (QUOTE ,(ADD1 I)))))))
                        '(AND (NOT (EMPTY-NODE STATE (QUOTE ,(ADD1 I))))
                              (EMPTY-NODE STATE (QUOTE ,I)))
                        (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE LEADS-TO-STRENGTHEN-LEFT
                  (($P (LIST 'OR
                             (LIST 'AND
                                   (LIST 'NOT
                                    (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE (ADD1 (ADD1 I)))))
                                   (LIST 'AND
                                    (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE (ADD1 I)))
                                    (LIST 'AND
                                     (LIST 'NOT
```

```
                                      (LIST 'EMPTY-NODE 'STATE
                                       (LIST 'QUOTE I)))
                                     (LIST 'NOT
                                      (LIST 'LISTP
                                       (LIST 'QUEUE-VALUES 'STATE
                                        (LIST 'QUOTE (ADD1 I))))))))
                             (LIST 'AND
                                   (LIST 'NOT
                                    (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE (ADD1 (ADD1 I)))))
                                   (LIST 'AND
                                    (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE (ADD1 I)))
                                    (LIST 'AND
                                     (LIST 'EMPTY-NODE 'STATE
                                      (LIST 'QUOTE I))
                                     (LIST 'NOT
                                      (LIST 'LISTP
                                       (LIST 'QUEUE-VALUES 'STATE
                                        (LIST 'QUOTE (ADD1 I))))))))))))
         (BASH (DISABLE EVAL)) (REWRITE DISJOIN-LEFT)
         (REWRITE FULL-EMPTY-FULL-MOVES-FORWARD)
         (REWRITE FULL-EMPTY-EMPTY-MOVES-FORWARD))))

(PROVE-LEMMA FULL-REST-EMPTY-MOVES-FORWARD (REWRITE)
             (IMPLIES (AND (INITIAL-CONDITION
                            '(PROPER-NODES STATE (QUOTE ,N))
                            (FIFO-QUEUE N))
                           (LESSP 1 N)
                           (LESSP I N)
                           (NOT (ZEROP I)))
                      (LEADS-TO
                       '(AND (NOT (EMPTY-NODE
                                   STATE (QUOTE ,(ADD1 I))))
                             (AND (EMPTY-NODE STATE (QUOTE ,I))
                                  (NOT (LISTP (QUEUE-VALUES
                                               STATE
                                               (QUOTE ,I))))))
                       '(AND (NOT (EMPTY-NODE STATE (QUOTE ,I)))
                             (AND (EMPTY-NODE STATE (QUOTE ,(SUB1 I)))
                                  (NOT (LISTP (QUEUE-VALUES
                                               STATE
                                               (QUOTE ,(SUB1 I)))))))
                       (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE PSP-GENERAL
                  (($P (LIST 'AND
                             (LIST 'NOT
                                   (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE (ADD1 I))))
                             (LIST 'AND
                                   (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE I))
                                   (LIST 'NOT
                                    (LIST 'LISTP
                                     (LIST 'QUEUE-VALUES 'STATE
                                      (LIST 'QUOTE I)))))))
                   ($Q (LIST 'AND
                             (LIST 'NOT
                                   (LIST 'EMPTY-NODE 'STATE
                                    (LIST 'QUOTE I)))
                             (LIST 'EMPTY-NODE 'STATE
                                   (LIST 'QUOTE (SUB1 I)))))
                   ($R (LIST 'AND
                             (LIST 'PROPER-NODES 'STATE
                                   (LIST 'QUOTE N))
                             (LIST 'AND
                                   (LIST 'NOT
                                    (LIST 'EMPTY-NODE 'STATE
```

```
                                     (LIST 'QUOTE (ADD1 I))))
                                   (LIST 'AND
                                    (LIST 'EMPTY-NODE 'STATE
                                     (LIST 'QUOTE I))
                                    (LIST 'NOT
                                     (LIST 'LISTP
                                      (LIST 'QUEUE-VALUES 'STATE
                                       (LIST 'QUOTE I))))))))
                 ($B (LIST 'AND
                           (LIST 'NOT
                                 (LIST 'EMPTY-NODE 'STATE
                                  (LIST 'QUOTE I)))
                           (LIST 'AND
                                 (LIST 'EMPTY-NODE 'STATE
                                  (LIST 'QUOTE (SUB1 I)))
                                 (LIST 'NOT
                                  (LIST 'LISTP
                                   (LIST 'QUEUE-VALUES 'STATE
                                    (LIST 'QUOTE (SUB1 I))))))))))
       (USE-LEMMA FULL-EMPTY-MOVES-FORWARD ((I (SUB1 I))))
       (BASH (DISABLE EVAL))
       (REWRITE FULL-EMPTY-REST-UNLESS-MOVES-FORWARD)
       (BASH (DISABLE EVAL)) (BASH (DISABLE EVAL)))))

(PROVE-LEMMA FULL-REST-EMPTY-OUTPUT-GROWS (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION
                 '(PROPER-NODES STATE (QUOTE ,N))
                 (FIFO-QUEUE N))
                (LESSP 1 N)
                (LESSP I N)
                (NUMBERP I))
           (LEADS-TO
            '(AND (NOT (EMPTY-NODE
                        STATE (QUOTE ,(ADD1 I))))
                  (AND (EMPTY-NODE STATE (QUOTE ,I))
                       (AND (NOT (LISTP (QUEUE-VALUES
                                         STATE
                                         (QUOTE ,I))))
                            (EQUAL (LENGTH (VALUE STATE
                                                  (QUOTE OUTPUT)))
                                   (QUOTE ,K)))))
            '(LESSP (QUOTE ,K)
                    (LENGTH (VALUE STATE (QUOTE OUTPUT))))
            (FIFO-QUEUE N)))
((INSTRUCTIONS
  (INDUCT (PLUS I J))
  PROMOTE PROMOTE
  (REWRITE LEADS-TO-STRENGTHEN-LEFT
           (($P (LIST 'AND
                      '(NOT (EMPTY-NODE STATE 1))
                      (LIST 'AND
                            '(EMPTY-NODE STATE 0)
                            (LIST 'EQUAL
                                  '(LENGTH (VALUE STATE 'OUTPUT))
                                  (LIST 'QUOTE K)))))))
  CHANGE-GOAL
  (REWRITE OUTPUT-GROWS-IMMEDIATELY)
  (GENERALIZE
   (((S
     (FIFO-QUEUE N)
     (ILEADS
            (LIST 'AND
                  (LIST 'NOT
                        (LIST 'EMPTY-NODE
                              'STATE
                              (LIST 'QUOTE (ADD1 I))))
                  (LIST 'AND
                        (LIST 'EMPTY-NODE
                              'STATE
```

```
                              (LIST 'QUOTE I))
                        (LIST 'AND
                              (LIST 'NOT
                                    (LIST 'LISTP
                                          (LIST 'QUEUE-VALUES
                                                'STATE
                                                (LIST 'QUOTE I))))
                              (LIST 'EQUAL
                                    '(LENGTH (VALUE STATE 'OUTPUT))
                                    (LIST 'QUOTE K)))))
            (FIFO-QUEUE N)
            (CONS 'LESSP
                  (CONS (LIST 'QUOTE K)
                        '((LENGTH (VALUE STATE 'OUTPUT)))))))
    STATE)))
 BASH PROMOTE PROMOTE
 (REWRITE LEADS-TO-TRANSITIVE
  (($Q
    (LIST 'OR
      (LIST 'AND
            (LIST 'NOT
                  (LIST 'EMPTY-NODE
                        'STATE
                        (LIST 'QUOTE I)))
            (LIST 'AND
                  (LIST 'EMPTY-NODE
                        'STATE
                        (LIST 'QUOTE (SUB1 I)))
                  (LIST 'AND
                        (LIST 'NOT
                              (LIST 'LISTP
                                    (LIST 'QUEUE-VALUES
                                          'STATE
                                          (LIST 'QUOTE (SUB1 I)))))
                        (LIST 'EQUAL
                              '(LENGTH (VALUE STATE 'OUTPUT))
                              (LIST 'QUOTE K)))))
      (LIST 'LESSP
            (LIST 'QUOTE K)
            '(LENGTH (VALUE STATE 'OUTPUT)))))))
 CHANGE-GOAL
 (REWRITE DISJOIN-LEFT)
 (BASH (DISABLE EVAL))
 (REWRITE Q-LEADS-TO-Q)
 (REWRITE PSP-GENERAL
    (($P (LIST 'AND
               (LIST 'NOT
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE (ADD1 I))))
               (LIST 'AND
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE I))
                     (LIST 'NOT
                           (LIST 'LISTP
                                 (LIST 'QUEUE-VALUES
                                       'STATE
                                       (LIST 'QUOTE I)))))))
     ($R (LIST 'EQUAL
               '(LENGTH (VALUE STATE 'OUTPUT))
               (LIST 'QUOTE K)))
     ($Q (LIST 'AND
               (LIST 'NOT
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE I)))
               (LIST 'AND
                     (LIST 'EMPTY-NODE
```

```
                                'STATE
                                (LIST 'QUOTE (SUB1 I)))
                          (LIST 'NOT
                                (LIST 'LISTP
                                      (LIST 'QUEUE-VALUES
                                            'STATE
                                            (LIST 'QUOTE (SUB1 I))))))))
    ($B (CONS 'LESSP
              (CONS (LIST 'QUOTE K)
                    '((LENGTH (VALUE STATE 'OUTPUT)))))))
(REWRITE FULL-REST-EMPTY-MOVES-FORWARD)
(REWRITE OUTPUT-NEVER-SHORTENS)
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (ILEADS
     (LIST 'AND
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (ADD1 I))))
           (LIST 'AND
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE I))
                 (LIST 'AND
                       (LIST 'NOT
                             (LIST 'LISTP
                                   (LIST 'QUEUE-VALUES
                                         'STATE
                                         (LIST 'QUOTE I))))
                       (LIST 'EQUAL
                             '(LENGTH (VALUE STATE 'OUTPUT))
                             (LIST 'QUOTE K)))))
     (FIFO-QUEUE N)
     (LIST 'OR
      (LIST 'AND
           (LIST 'NOT
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE I)))
           (LIST 'AND
                 (LIST 'EMPTY-NODE
                       'STATE
                       (LIST 'QUOTE (SUB1 I)))
                 (LIST 'AND
                       (LIST 'NOT
                             (LIST 'LISTP
                                   (LIST 'QUEUE-VALUES
                                         'STATE
                                         (LIST 'QUOTE (SUB1 I)))))
                       (LIST 'EQUAL
                             '(LENGTH (VALUE STATE 'OUTPUT))
                             (LIST 'QUOTE K)))))
      (CONS 'LESSP
            (CONS (LIST 'QUOTE K)
                  '((LENGTH (VALUE STATE 'OUTPUT)))))))
   STATE)))
(BASH (DISABLE EVAL))
(GENERALIZE
 (((S
    (FIFO-QUEUE N)
    (JLEADS
     (ILEADS
      (LIST 'AND
            (LIST 'NOT
                  (LIST 'EMPTY-NODE
                        'STATE
                        (LIST 'QUOTE (ADD1 I))))
```

```
                (LIST 'AND
                      (LIST 'EMPTY-NODE
                            'STATE
                            (LIST 'QUOTE I))
                      (LIST 'AND
                            (LIST 'NOT
                                  (LIST 'LISTP
                                        (LIST 'QUEUE-VALUES
                                              'STATE
                                              (LIST 'QUOTE I))))
                            (LIST 'EQUAL
                                  '(LENGTH (VALUE STATE 'OUTPUT))
                                  (LIST 'QUOTE K)))))
          (FIFO-QUEUE N)
          (LIST 'OR
           (LIST 'AND
               (LIST 'NOT
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE I)))
               (LIST 'AND
                     (LIST 'EMPTY-NODE
                           'STATE
                           (LIST 'QUOTE (SUB1 I)))
                     (LIST 'AND
                           (LIST 'NOT
                                 (LIST 'LISTP
                                       (LIST 'QUEUE-VALUES
                                             'STATE
                                             (LIST 'QUOTE (SUB1 I)))))
                           (LIST 'EQUAL
                                 '(LENGTH (VALUE STATE 'OUTPUT))
                                 (LIST 'QUOTE K)))))
           (CONS 'LESSP
                 (CONS (LIST 'QUOTE K)
                       '((LENGTH (VALUE STATE 'OUTPUT)))))))
         (FIFO-QUEUE N)
         (LIST 'OR
          (LIST 'AND
             (LIST 'AND
                   (LIST 'NOT
                         (LIST 'EMPTY-NODE
                               'STATE
                               (LIST 'QUOTE I)))
                   (LIST 'AND
                         (LIST 'EMPTY-NODE
                               'STATE
                               (LIST 'QUOTE (SUB1 I)))
                         (LIST 'NOT
                               (LIST 'LISTP
                                     (LIST 'QUEUE-VALUES
                                           'STATE
                                           (LIST 'QUOTE (SUB1 I)))))))
             (LIST 'EQUAL
                   '(LENGTH (VALUE STATE 'OUTPUT))
                   (LIST 'QUOTE K)))
          (CONS 'LESSP
                (CONS (LIST 'QUOTE K)
                      '((LENGTH (VALUE STATE 'OUTPUT))))))))
      STATE)))
   (BASH (DISABLE EVAL)))))


(DEFN FIRST-EMPTY-SUBQUEUE (STATE N)
  (IF (ZEROP N)
      0
    (IF (LISTP (QUEUE-VALUES STATE N))
        (FIRST-EMPTY-SUBQUEUE STATE (SUB1 N))
      N)))
```

```
(PROVE-LEMMA FIRST-EMPTY-SUBQUEUE-IS-EMPTY (REWRITE)
             (NOT (LISTP (QUEUE-VALUES
                          STATE
                          (FIRST-EMPTY-SUBQUEUE STATE N)))))

(PROVE-LEMMA QUEUE-NOT-EMPTY-IMPLIES (REWRITE)
  (IMPLIES (LISTP (QUEUE-VALUES STATE N))
           (AND (NOT (EMPTY-NODE STATE
                                 (ADD1 (FIRST-EMPTY-SUBQUEUE
                                        STATE N))))
                (EMPTY-NODE STATE (FIRST-EMPTY-SUBQUEUE STATE N))))
  ((DISABLE EMPTY-NODE)))

(PROVE-LEMMA NOT-LESSP-FIRST-EMPTY-SUBQUEUE (REWRITE)
             (NOT (LESSP N (FIRST-EMPTY-SUBQUEUE STATE N))))

(PROVE-LEMMA LESSP-FIRST-EMPTY-SUBQUEUE (REWRITE)
             (IMPLIES (LISTP (QUEUE-VALUES STATE N))
                      (LESSP (FIRST-EMPTY-SUBQUEUE STATE N)
                             N))
             ((EXPAND (QUEUE-VALUES STATE N)
                      (FIRST-EMPTY-SUBQUEUE STATE N))))

(PROVE-LEMMA OUTPUT-GROWS (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION '(PROPER-NODES STATE (QUOTE ,N))
                                   (FIFO-QUEUE N))
                (LESSP 1 N))
           (LEADS-TO '(AND (LISTP (QUEUE-VALUES STATE N))
                           (EQUAL (LENGTH (VALUE STATE
                                                 (QUOTE OUTPUT)))
                                  (QUOTE ,K)))
                     '(LESSP (QUOTE ,K)
                             (LENGTH (VALUE STATE (QUOTE OUTPUT))))
                     (FIFO-QUEUE N)))
((INSTRUCTIONS PROMOTE
  (CLAIM
   (LISTP
    (QUEUE-VALUES
        (S (FIFO-QUEUE N)
           (ILEADS (LIST 'AND
                         '(LISTP (QUEUE-VALUES STATE N))
                         (LIST 'EQUAL
                               '(LENGTH (VALUE STATE 'OUTPUT))
                               (LIST 'QUOTE K)))
                   (FIFO-QUEUE N)
                   (CONS 'LESSP
                         (CONS (LIST 'QUOTE K)
                               '((LENGTH (VALUE STATE 'OUTPUT)))))))
        N))
   0)
  (REWRITE LEADS-TO-STRENGTHEN-LEFT
   (($P
     (LIST 'AND
      (LIST 'NOT
            (LIST 'EMPTY-NODE
                  'STATE
                  (LIST 'QUOTE
                        (ADD1 (FIRST-EMPTY-SUBQUEUE STATE N)))))
      (LIST 'AND
       (LIST 'EMPTY-NODE
             'STATE
             (LIST 'QUOTE
                   (FIRST-EMPTY-SUBQUEUE STATE N)))
       (LIST 'AND
           (LIST 'NOT
                 (LIST 'LISTP
                       (LIST 'QUEUE-VALUES
                             'STATE
                             (LIST 'QUOTE
```

```
                                    (FIRST-EMPTY-SUBQUEUE STATE N)))))
          (LIST 'EQUAL
                '(LENGTH (VALUE STATE 'OUTPUT))
                (LIST 'QUOTE K)))))))
  (PUT
   (STATE
     (S (FIFO-QUEUE N)
        (ILEADS (LIST 'AND
                      '(LISTP (QUEUE-VALUES STATE N))
                      (LIST 'EQUAL
                            '(LENGTH (VALUE STATE 'OUTPUT))
                            (LIST 'QUOTE K)))
                (FIFO-QUEUE N)
                (CONS 'LESSP
                      (CONS (LIST 'QUOTE K)
                            '((LENGTH (VALUE STATE 'OUTPUT))))))))
  BASH
  (REWRITE FULL-REST-EMPTY-OUTPUT-GROWS)
  PROVE
  (REWRITE LEADS-TO-STRENGTHEN-LEFT
           (($P '(FALSE))))
  BASH
  (REWRITE PROVE-LEADS-TO)
  PROVE)))

(PROVE-LEMMA IN-NODE-LEAVES-REST-OF-QUEUE-UNCHANGED (REWRITE)
             (IMPLIES (AND (N OLD NEW (LIST 'IN-NODE N))
                           (LESSP 1 N)
                           (LESSP I N)
                           (NUMBERP I))
                      (AND (EQUAL (VALUE NEW 'OUTPUT)
                                  (VALUE OLD 'OUTPUT))
                           (EQUAL (QUEUE-VALUES NEW I)
                                  (QUEUE-VALUES OLD I))))
    ((INSTRUCTIONS SPLIT BASH (INDUCT (PLUS I J)) (BASH (DISABLE N))
         PROMOTE PROMOTE
         (CLAIM (AND (NOT (EQUAL I N)) (NOT (EQUAL (SUB1 I) N)))
                ((DISABLE N)))
         (BASH T))))

(PROVE-LEMMA IN-NODE-PRESERVES-VALUES (REWRITE)
             (IMPLIES (AND (N OLD NEW (LIST 'IN-NODE N))
                           (PROPER-NODE OLD N)
                           (EQUAL (VALUE OLD 'INPUT)
                                  (APPEND (QUEUE-VALUES OLD N)
                                          (VALUE OLD 'OUTPUT)))
                           (LESSP 1 N))
                      (EQUAL (VALUE NEW 'INPUT)
                             (APPEND (QUEUE-VALUES NEW N)
                                     (VALUE NEW 'OUTPUT))))
    ((INSTRUCTIONS (DIVE 1) (DIVE 2) (DIVE 2) (DIVE 1) (DIVE 2)
         (DIVE 1) X TOP PROMOTE (DIVE 2) (DIVE 1) X-DUMB (DIVE 3)
         (DIVE 2) (DIVE 2)
         (REWRITE IN-NODE-LEAVES-REST-OF-QUEUE-UNCHANGED
                  (($OLD OLD) ($N N)))
         UP UP (DIVE 3)
         (REWRITE IN-NODE-LEAVES-REST-OF-QUEUE-UNCHANGED
                  (($OLD OLD) ($N N)))
         TOP (DIVE 2) (DIVE 2)
         (REWRITE IN-NODE-LEAVES-REST-OF-QUEUE-UNCHANGED
                  (($OLD OLD) ($N N) ($I (SUB1 N))))
         TOP (BASH T) PROVE PROVE PROVE)))

(PROVE-LEMMA OUT-NODE-PRESERVES-QUEUE-VALUES (REWRITE)
             (IMPLIES (AND (N OLD NEW '(OUT-NODE))
                           (LESSP 1 N)
                           (NOT (LESSP N I))
                           (NOT (ZEROP I)))
                      (AND (EQUAL (VALUE NEW 'INPUT)
```

```
                              (VALUE OLD 'INPUT))
                       (EQUAL (APPEND (QUEUE-VALUES NEW I)
                                      (VALUE NEW 'OUTPUT))
                              (APPEND (QUEUE-VALUES OLD I)
                                      (VALUE OLD 'OUTPUT)))))
    ((INSTRUCTIONS SPLIT BASH (INDUCT (PLUS I J)) (BASH (DISABLE N))
         (CLAIM (EQUAL I 1) 0)
         (BASH T (EXPAND (QUEUE-VALUES OLD 1) (QUEUE-VALUES NEW 1)))
         (BASH T))))


(PROVE-LEMMA EXTERNAL-NOR-GATES-PRESERVES-VALUES (REWRITE)
             (IMPLIES (AND (N OLD NEW
                              (LIST 'NOR-GATE (CT (SUB1 N))
                                    (CF (SUB1 N)) (TEMP N)))
                           (LESSP 1 N)
                           (NOT (LESSP N I))
                           (NUMBERP I))
                      (AND (EQUAL (VALUE NEW 'INPUT)
                                  (VALUE OLD 'INPUT))
                           (EQUAL (VALUE NEW 'OUTPUT)
                                  (VALUE OLD 'OUTPUT))
                           (EQUAL (QUEUE-VALUES NEW I)
                                  (QUEUE-VALUES OLD I))))
    ((INSTRUCTIONS SPLIT BASH BASH (INDUCT (PLUS I J)) BASH BASH)))

(PROVE-LEMMA INTERNAL-NODES-PRESERVES-REST-OF-QUEUE (REWRITE)
             (IMPLIES (AND (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (FIFO-NODE I))
                           (NUMBERP J)
                           (LESSP J I))
                      (EQUAL (QUEUE-VALUES NEW J)
                             (QUEUE-VALUES OLD J)))
    ((INSTRUCTIONS (INDUCT (PLUS J K)) BASH PROMOTE PROMOTE
         (CLAIM (NOT (EQUAL J I))) (CLAIM (NOT (EQUAL (SUB1 J) I)))
         (BASH T))))

(PROVE-LEMMA INTERNAL-NODES-PRESERVE-VALUES (REWRITE)
             (IMPLIES (AND (N OLD NEW STATEMENT)
                           (MEMBER STATEMENT (FIFO-NODE I))
                           (PROPER-NODE OLD (ADD1 I))
                           (PROPER-NODE OLD I)
                           (NOT (ZEROP I))
                           (LESSP I J))
                      (AND (EQUAL (VALUE NEW 'INPUT)
                                  (VALUE OLD 'INPUT))
                           (EQUAL (VALUE NEW 'OUTPUT)
                                  (VALUE OLD 'OUTPUT))
                           (EQUAL (QUEUE-VALUES NEW J)
                                  (QUEUE-VALUES OLD J))))
 ((INSTRUCTIONS
   SPLIT
   (DROP 3 4 6)
   (BASH T)
   (DROP 3 4 6)
   (BASH T)
   (INDUCT (PLUS J K))
   (BASH T (DISABLE PROPER-NODE N))
   PROMOTE PROMOTE
   (DEMOTE 2)
   (DIVE 1)
   (DIVE 1)
   (S NIL)
   S TOP
   (CLAIM (EQUAL (ADD1 I) J) 0)
   PROMOTE
   (DROP 9 7 1)
   (CLAIM (EQUAL (QUEUE-VALUES NEW (SUB1 (SUB1 J)))
                 (QUEUE-VALUES OLD (SUB1 (SUB1 J))))
```

```
          ((DISABLE FIFO-NODE PROPER-NODE N)))
   (BASH T
         (DISABLE INTERNAL-NODES-PRESERVES-REST-OF-QUEUE))
   (CLAIM (NOT (EQUAL I J)))
   (DROP 4 5)
   (BASH T))))


(PROVE-LEMMA VALUES-PRESERVED (REWRITE)
             (IMPLIES (AND (N OLD NEW STATEMENT)
                           (PROPER-NODES OLD N)
                           (LESSP 1 N)
                           (EQUAL (VALUE OLD 'INPUT)
                                  (APPEND (QUEUE-VALUES OLD N)
                                          (VALUE OLD 'OUTPUT)))
                           (MEMBER STATEMENT (FIFO-QUEUE N)))
                      (EQUAL (VALUE NEW 'INPUT)
                             (APPEND (QUEUE-VALUES NEW N)
                                     (VALUE NEW 'OUTPUT))))
    ((INSTRUCTIONS
         (BASH (DISABLE N VALUE FIFO-NODE CT CF TEMP PROPER-NODE))
         (USE-LEMMA EXTERNAL-NOR-GATES-PRESERVES-VALUES ((I N)))
         PROMOTE (DEMOTE 1) (DIVE 1) (DIVE 1) (S NIL) (= * T 0) TOP
         SPLIT (DIVE 1) = = UP (DIVE 2) (DIVE 1) = UP (DIVE 2) = TOP S
         (DROP 1 2 4) PROVE
         (USE-LEMMA OUT-NODE-PRESERVES-QUEUE-VALUES ((I N))) PROMOTE
         (DEMOTE 1) (DIVE 1) (DIVE 1) (S NIL) (= * T 0) TOP SPLIT
         (DIVE 1) = = TOP (DIVE 2) = TOP S (DROP 1 2 4) PROVE PROMOTE
         (USE-LEMMA IN-NODE-PRESERVES-VALUES) (DEMOTE 5) (DIVE 1)
         (DIVE 1) (S NIL) (= * T IFF) UP (DIVE 2) (DIVE 2) (DIVE 2)
         (REWRITE IN-NODE-LEAVES-REST-OF-QUEUE-UNCHANGED (($I 0))) TOP
         S (DROP 1 2 4) PROVE S-PROP (REWRITE PROPER-NODES-IMPLIES)
         DROP PROVE (DROP 1 2 4) PROVE (DROP 1 2 4) PROVE PROMOTE
         (CLAIM (LESSP (CDADDDR STATEMENT) N) 0)
         (CLAIM (NOT (ZEROP (CDADDDR STATEMENT))) 0) (DIVE 1)
         (CLAIM (PROPER-NODE OLD (CDADDDR STATEMENT)) 0)
         (CLAIM (PROPER-NODE OLD (ADD1 (CDADDDR STATEMENT))) 0)
         (REWRITE INTERNAL-NODES-PRESERVE-VALUES
                  (($I (CDADDDR STATEMENT)) ($J N) ($OLD OLD)))
         UP (DIVE 2) (DIVE 2)
         (REWRITE INTERNAL-NODES-PRESERVE-VALUES
                  (($I (CDADDDR STATEMENT)) ($J N) ($OLD OLD)))
         TOP (DIVE 1) = TOP S TOP (CONTRADICT 13)
         (REWRITE PROPER-NODES-IMPLIES)
         (DROP 1 2 3 4 5 6 7 8 9 11 12 13) PROVE DROP PROVE TOP
         (CONTRADICT 12) (REWRITE PROPER-NODES-IMPLIES)
         (DROP 1 2 3 4 5 6 7 8 9 11 12) PROVE (CONTRADICT 11)
         (DROP 1 2 3 4 5 6 9 10 11) PROVE (CONTRADICT 6)
         (DROP 1 2 3 4 5 6 7 8) PROVE)))


(PROVE-LEMMA VALUES-INVARIANT (REWRITE)
 (IMPLIES (LESSP 1 N)
          (UNLESS `(AND (PROPER-NODES STATE (QUOTE ,N))
                        (EQUAL (VALUE STATE (QUOTE INPUT))
                               (APPEND (QUEUE-VALUES STATE
                                                     (QUOTE ,N))
                                       (VALUE STATE
                                              (QUOTE OUTPUT)))))
                   '(FALSE)
                   (FIFO-QUEUE N)))
             ((INSTRUCTIONS PROMOTE (REWRITE HELP-PROVE-UNLESS)
                            (PROVE (DISABLE MEMBER-FIFO-QUEUE VALUE N)))))

(PROVE-LEMMA QUEUE-VALUES-INVARIANT (REWRITE)
  (IMPLIES (AND (INITIAL-CONDITION
                 `(AND (PROPER-NODES STATE (QUOTE ,N))
                       (EQUAL (VALUE STATE (QUOTE INPUT))
                              (APPEND (QUEUE-VALUES STATE
```

```
                                                        (QUOTE ,N))
                                          (VALUE STATE
                                                 (QUOTE OUTPUT)))))
                  (FIFO-QUEUE N))
                 (LESSP 1 N))
            (INVARIANT ‘(EQUAL (VALUE STATE (QUOTE INPUT))
                               (APPEND (QUEUE-VALUES STATE
                                                        (QUOTE ,N))
                                          (VALUE STATE
                                                 (QUOTE OUTPUT))))
                        (FIFO-QUEUE N)))
    ((INSTRUCTIONS PROMOTE
         (REWRITE INVARIANT-CONSEQUENCE
                  (($P (LIST ’AND
                             (LIST ’PROPER-NODES ’STATE
                                   (LIST ’QUOTE N))
                             (LIST ’EQUAL ’(VALUE STATE ’INPUT)
                                   (LIST ’APPEND
                                    (LIST ’QUEUE-VALUES ’STATE
                                     (LIST ’QUOTE N))
                                    ’(VALUE STATE ’OUTPUT)))))))
         (REWRITE UNLESS-PROVES-INVARIANT
                  (($IC (LIST ’AND
                              (LIST ’PROPER-NODES ’STATE
                                    (LIST ’QUOTE N))
                              (LIST ’EQUAL ’(VALUE STATE ’INPUT)
                                    (LIST ’APPEND
                                     (LIST ’QUEUE-VALUES ’STATE
                                      (LIST ’QUOTE N))
                                     ’(VALUE STATE ’OUTPUT)))))))
         (REWRITE VALUES-INVARIANT) S DROP (BASH (DISABLE EVAL)))))


))
```

References

[1] Martin Abadi and Leslie Lamport.
The Existence of Refinement Mappings.
Technical Report Research Report 29, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301,
August, 1988.

[2] Bowen Alpern and Fred B. Schneider.
Defining Liveness.
*Information Processing Letters*21:181-185, 1985.

[3] Bowen Alpern, Alan J. Demers, and Fred B. Schneider.
Safety Without Stuttering.
*Information Processing Letters*23:177-180, 1986.

[4] Flemming Anderson.
A Verified Theorem Prover for UNITY in Higher Order Logic.
PhD thesis, Technical University of Denmark, .
to appear.

[5] K.R. Apt, N. Francez, and S. Katz.
Appraising Fairness in Distributed Languages.
*Distributed Computing*2:226-241, August, 1988.

[6] M. Ben-Ari.
Algorithms for On-the-Fly Garbage Collection.
*ACM Transactions of Programming Languages and Systems*6:281-296, 1984.

[7] William R. Bevier.
A Library for Hardware Verification.
Technical Report , Computational Logic, Inc., Austin, Texas 78703,
1988.
CLI Internal Note 57.

[8] William R. Bevier, Warren A. Hunt, Jr., and William D. Young.
Toward Verified Execution Environments.
Technical Report 5, Computational Logic, Inc.,
1987.
Also appears in 'Proceedings of the 1987 IEEE Symposium on Security and Privacy'.

[9] R. S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[10] R. S. Boyer and J S. Moore.
Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.
In R. S. Boyer and J S. Moore (editors), *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[11] R. S. Boyer and J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[12] R. S. Boyer and J S. Moore.
The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover.
Technical Report ICSCA-CMP-52, Institute for Computer Science, University of Texas at Austin,
January, 1988.
To appear in the *Journal of Automated Reasoning*, 1988. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

[13] R.S. Boyer, D. Goldschlag, M. Kaufmann, J Strother Moore.
Functional Instantiation in First Order Logic.
In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Pages 7-26. Academic Press, 1991.

[14] Michael C. Browne, Edmund M. Clarke, and David L. Dill.
Automatic Ciruit Verification Using Temporal Logic: Two New Examples.
In George J. Milne and P. A. Subrahmanyam (editors), *Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 113-124. North Holland, 1986.

[15] Jerry R. Burch.
Combining CTL, Trace Theory, and Timing Models.
In J. Sifakis (editors), *Automatic Verification Methods for Finite State Systems*, pages 334-348. Springer-Verlag, 1990.

[16] Albert Camilleri.
Reasoning in CSP via the HOL Theorem Prover.
*IEEE Transactions on Software Engineering*SE-16, September, 1990.

[17] K. Mani Chandy and Jayadev Misra.
Parallel Program Design: A Foundation.
Addison Wesley, Massachusetts, 1988.

[18] E.M. Clarke and O. Grumberg.
Research on Automatic Verification of Finite State Systems.
Technical Report CS-87-105, CMU,
January, 1987.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla.
Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic.
*ACM Transactions on Programming Languages and Systems*8(2):244-263, April, 1986.

[20] Avra Cohn.
The Notion of Proof in Hardware Verification.
*Journal of Automated Reasoning*5(2):127-139, June, 1989.

[21] Jimi Crawford and David Goldschlag.
The Mechanical Verification of Distributed Systems.
Technical Report , Computational Logic, Inc. Austin Texas 78703,
July, 1987.
Technical Report 7.

[22] David L. Dill.
Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.
The MIT Press, Cambridge, Massachusetts, 1988.

[23] Nissim Francez.
Fairness.
Springer-Verlag, New York, 1986.

[24] S.J. Garland, J.V. Guttag, J.J. Horning.
Debugging Larch Shared Language Specifications.
*IEEE Transactions on Software Engineering*SE-16(9), September, 1990.

[25] Steven M. German and Yu Wang.
Formal Verification of Parameterized Hardware Designs.
Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers:549-552, 1985.

[26] Rob Gerth and Amir Pnueli.
Rooting UNITY.
In *Fifth International Workshop on Software Specification and Design*, Pages 11-19. ACM Sigsoft Engineering Notes, 1989.

[27] David Goldschlag.
A Mechanically Verified Proof System for Concurrent Programs.
Technical Report , Computational Logic, Inc. Austin Texas 78703,
January, 1989.
Technical Report 32.

[28] David M. Goldschlag.
Proving Proof Rules: A Proof System for Concurrent Programs.
Compass '90, June, 1990.

[29] David M. Goldschlag.
Mechanizing Unity.
In M. Broy and C. B. Jones (editors), *Programming Concepts and Methods*. North Holland, Amsterdam, 1990.

[30] David M. Goldschlag.
Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover.
*IEEE Transactions on Software Engineering*SE-16(9), September, 1990.

[31] David Goldschlag.
A Mechanical Formalization of Several Fairness Notions.
In S. Prehn and W.J. Toetenel (editors), *VDM '91: Formal Software Development Methods*. Springer-Verlag Lecture Notes in Computer Science 551, 1991.

[32] David M. Goldschlag.
Verifying Safety and Liveness Properties of a Delay Insensitive Fifo Circuit on the Boyer-Moore Prover.
1991 International Workshop on Formal Methods in VLSI Design, January, 1991.

[33] David M. Goldschlag.
Mechanically Verifying Safety and Liveness Properties of Delay Insensitive Circuits.
Computer Aided Verification 1991, July, 1991.

[34] D. I. Good, R. M. Cohen, and J. Keeton-Williams.
Principles of Proving Concurrent Programs in Gypsy.
In *Proceedings of 6th Symposium of Principles of Programming Languages*, ACM, January, 1979.

[35] Donald I. Good.
The Proof of a Distributed System in Gypsy.
In *Formal Specification - Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar - M. J. Elphick (Ed.)*, September, 1982.
Also Technical Report #30, Institute for Computing Science, The University of Texas at Austin.

[36] M. Gordon.
HOL: A Proof Generating System for Higher-Order Logic.
Technical Report 103, University of Cambridge, Computer Laboratory,
1987.

[37] David Gries.
An Exercise in Proving Parallel Programs Correct.
*CACM*20(12):921-930, 1977.

[38] John Guttag, James Horning, and Jeannette Wing.
The Larch Family of Specification Languages.
IEEE Transactions on Software Engineering:24-36, September, 1985.

[39] Eric C.R. Hehner.
Predicative Programming.
*CACM*27(2):134-151, 1984.

[40] C.A.R. Hoare.
An Axiomatic Basis for Computer Programming.
*CACM*12(10):576-580, 1969.

[41] C.A.R. Hoare.
An Axiomatic Basis for Computer Programming.
*CACM*12:271-281, 1969.

[42] C.A.R. Hoare.
Towards a Theory of Parallel Programming.
In Hoare and Perott (editors), *Operating System Techniques*. Academic Press, New York, 1972.

[43] C.A.R. Hoare.
Communicating Sequential Processes.
*CACM*21(8):666-677, 1978.

[44] C.A.R. Hoare.
Communicating Sequential Processes.
Prentice Hall International, Englewood Cliffs, NJ, 1985.

[45] Warren A. Hunt, Jr.
Microprocessor Design Verification.
*Journal of Automated Reasoning*5(4):429-460, December, 1989.

[46] INMOS Limited.
Occam Programming Manual.
Prentice-Hall International, Englewood Cliffs, NJ, 1984.

[47] INMOS Limited.
Transputer Reference Manual.
Prentice Hall International, New York, 1988.

[48] He Jifeng, M.B. Josephs, and C.A.R. Hoare.
A Theory of Synchrony and Asynchrony.
In M. Broy and C. B. Jones (editors), *Programming Concepts and Methods*, pages 459-478. North Holland, Amsterdam, 1990.

[49] Cliff B. Jones.
Software Development: A Rigorous Approach.
Prentice Hall, 1980.

[50] Charanjit S. Jutla, Edgar Knapp, and Josyula R. Rao.
Extensional Semantics of Parallel Programs.
Technical Report , Department of Computer Sciences, The University of Texas at Austin, November, 1988.

[51] M. Kaufmann.
A Formal Semantics and Proof of Soundness for the Logic of the NQTHM Version of the Boyer-Moore Theorem Prover.
Technical Report , Institute for Computing Science, University of Texas at Austin, Austin, TX 78712,
1986.
ICSCA Internal Note 229.

[52] M. Kaufmann.
A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover.
Technical Report ICSCA-CMP-60, Institute for Computing Science, University of Texas at Austin, Austin, TX 78712,
1987.
Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

[53] Matt Kaufmann.
DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers.
Technical Report 43, Computational Logic, Inc.,
May, 1989.
Draft.

[54] Edgar Knapp.
Soundness and Relative Completeness of Unity Logic.
Technical Report Department of Computer Science, The University of Texas at Austin, October, 1990.

[55] Leslie Lamport.
The 'Hoare Logic' of Concurrent Programs.
*Acta Informatica*14(1980):21-37, 1980.

[56] Leslie Lamport.
Personal Communication.

[57] Leslie Lamport.
A Simple Approach to Specifying Concurrent Systems.
*Communications of the ACM*32:32-45, 1989.

[58] Leslie Lamport.
The Temporal Logic of Actions.
Technical Report 79, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, 1991.

[59] Leslie Lamport and Fred B. Schneider.
The 'Hoare Logic' of CSP, and All That.
*ACM Transactions of Programming Languages and Systems*6(2):281-296, 1984.

[60] Z. Manna and A. Pnueli.
Verification of Concurrent Programs: The Temporal Framework.
In R. S. Boyer and J S. Moore (editors), *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[61] Zohar Manna and Amir Pnueli.
Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs.
*Science of Computer Programming*4:257-289, 1984.

[62] Zohar Manna and Amir Pnueli.
An Exercise in the Verification of Multi-Process Programs.
In W.H.J. Feijen, A.J.M. van gasteren, D. Gries, and J. Misra (editors), *Beautry is Our Business*, pages 289-301. Springer Verlag, 1990.

[63] Alain J. Martin.
Compiling Communicating Processes into Delay-Insensitive VLSI Circuits.
*Distributed Computing*1:226-234, 1986.

[64] Alain J. Martin.
Self-Timed FIFO: An Exercise in Compiling Programs into VLSI Circuits.
In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 133-153. North-Holland, Amsterdam, 1987.

[65] R. E. Miller.
Switching Theory.
Wiley, 1965.

[66] Robin Milner.
A Calculus of Communicating Systems.
Springler-Verlag, Berlin, 1980.

[67] Jayadev Misra.
Auxiliary Variables.
Technical Report Notes on UNITY: 15-90, Department of Computer Sciences, The University of Texas at Austin,
July, 1990.

[68] Jayadev Misra.
Soundess of the Substitution Axiom.
Technical Report Notes on UNITY: 14-90, Department of Computer Sciences, The University of Texas at Austin,
March, 1990.

[69] Jayadev Misra.
Preserving Progress Under Program Composition.
Technical Report Notes on UNITY: 17-90, Department of Computer Sciences, The University of Texas at Austin,
July, 1990.

[70] J Strother Moore.
A Mechanically Verified Language Implemenation.
*Journal of Automated Reasoning*5(4):493-518, December, 1989.

[71] J Strother Moore.
System Verification.
*Journal of Automated Reasoning*5(4):409-410, December, 1989.

[72] Misao Nagayama and Carolyn Talcott.
An NQTHM Mechanization of an Exercise in the Verification of Multi-Process Programs.
Technical Report STAN-CS-91-1370, Stanford University,
1991.

[73] S. Owicki.
Axiomatic Proof Techniques for Parallel Programs.
PhD thesis, Cornell University, 1975.

[74] Susan Owicki and David Gries.
Verifying Parallel Programs: An Aximatic Approach.
*CACM*19(5):279-285, 1976.

[75] S. Owicki and L. Lamport.
Proving Liveness Properties of Concurrent Programs.
*ACM TOPLAS 4*3:455-495, 1982.

[76] J. Pachl.
A Simple Proof of a Completeness Result for leads-to in the UNITY Logic.
Technical Report RZ 2060 (#72085), IBM Research Division,
November, 1990.

[77] David M. Russinoff.
Verifying Concurrent Programs with the Boyer-Moore Prover.
Technical Report STP/ACT-218-90, MCC, Austin, Texas,
1990.

[78] David M. Russinoff.
A Mechanically Verified Incremental Garbage Collector.
Technical Report STP/ACT--91, MCC, Austin, Texas,
1991.

[79] B. A. Sanders.
Stepwise Refinement of Mixed Specifications of Concurrent Programs.
In M. Broy and C. B. Jones (editors), *Programming Concepts and Methods*. North Holland, Amsterdam, 1990.

[80] N. Shankar.
Proof Checking Metamathematics: Volumes I and II.
Technical Report 9, Computational Logic, Inc.,
April, 1987.

[81] N. Shankar.
A Mechanical Proof of the Church-Rosser Theorem.
*Journal of the ACM*35:475-522, 1988.

[82] Ambuj Singh.
Leads-To and Program Union.
Technical Report Notes on UNITY: 06-89, Department of Computer Sciences, The University of Texas at Austin,
June, 1989.

[83] J. Staunstrup and M.R. Greenstreet.
Designing Delay Insensitive Circuits using ‘‘Synchronized Transitions’’.
In Dr. Luc Claesen (editors), *Proceedings of the IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 741-758. Elsevier Science Publishers B.V., Amsterdam, 1989.

[84] Jorgen Staunstrup, Stephen J. Garland, and John V. Guttag.
Localized Verification of Circuit Descriptions.
In J. Sifakis (editors), *Automatic Verification Methods for Finite State Systems*, pages 348-364. Springer-Verlag, 1990.

[85] G. L. Steele, Jr.
Common Lisp The Language.
Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.

[86] J. von Wright.
Mechanising the Temporal Logic of Actions in HOL.
In *1991 International Workshop on the HOL Theorem Proving System and its Applications*, 1991.

[87] William D. Young.
A Mechanically Verified Code Generator.
*Journal of Automated Reasoning*5(4):493-518, December, 1989.

Index

This index lists the primary reference to each of the definitions and theorems introduced in the main text of this thesis. Definition names are in **SMALL CAPITALS**. Theorem names are in **Boldface**.

Index

Table of Contents

List of Figures

List of Tables