The Mechanical Verification of Distributed Systems

J. M. Crawford
D. M. Goldschlag

Technical Report #7 July 1987

Computational Logic, Inc.
1717 West Sixth Street
Suite 290
Austin, Texas 78703

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

Acknowledgements

At the University of Texas at Austin, Crawford's work was supported in part by Grant ARO DAAG 29-84-k-0060. Goldschlag's work was supported in part at the University of Texas by NSF Grant DCR81-22039 and at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the authors 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.

Chapter 1
INTRODUCTION

This paper describes a theory for the mechanical verification of distributed systems which has been implemented on the Boyer-Moore theorem prover [1] [2]. The Boyer-Moore Logic, the language of the theorem prover, is a quantifier free version of predicate calculus, with recursion. Our theory has been used to mechanically prove safety and liveness properties for a solution to the N-processor mutual exclusion problem. It defines a novel characterization of fairness and demonstrates the use of an operational semantics to derive proof rules.

1.1 The Transition System Model

Our model of distributed systems is a *transition system* model. A transition system consists of a set of system states, and a set of mappings from system states to system states called *processes*. A process is selected and applied to some initial state, producing a new state. This is repeated, generating a sequence of system states, called a *system computation*, and a sequence of process names, called a *trace*. If every process name appears infinitely often in the trace, the trace is said to be *(weakly) fair*.

The behavior of the transition system model can be abstractly characterized by meta-theorems called *proof rules*. These proof rules are used to prove safety and liveness properties. Safety properties state that a bad property is never introduced into the system; liveness properties state that a good property will eventually hold. In order to prove a property about a transition system, one must prove that the property holds for all fair traces. The fairness of traces is critically important for proving liveness properties.

Implicit in the transition system model is the assumption that the execution of a distributed system can be modeled by interleaving atomic transitions. An atomic transition executed concurrently with other events produces the same result it would have produced if executed in isolation. Furthermore, the transition system model assumes that processes are scheduled fairly. Thus, the domain of the transition system model is those distributed systems which satisfy the assumptions of atomicity and fairness.

1.2 The Boyer-Moore Logic and its Theorem Prover

The Boyer-More Logic is a quantifier free version of predicate calculus with recursion. All formulas are implicitly universally quantified. Recursive functions are allowed, provided that one can prove that they terminate. The syntax of the logic is similar to *Pure Lisp*. Since we use the Boyer-Moore Logic in this paper, we now present informal definitions for several common Boyer-Moore functions.

- The term *(Add1 X)*'s value is the number one greater than *X*.
- *List* makes a list out of its arguments; e.g., the term *(List X Y)* is a list containing two elements, *X* and *Y*.
- *Car* extracts the first element of a list; hence *(Car (List X Y)* is X.
- *Cdr* returns everything in the list but the first element; hence *(Cdr (List X Y))* is *(List Y)*.
- *Cons* inserts its first argument into a list which is its second argument.
- Multiple *Car*'s and *Cdr*'s may be combined into a single function by putting the appropriate sequence of *a*'s and *d*'s between $C \cdots R$ (e.g., *(Cadr X) = (Car (Cdr X))*.
- *(If X Y Z)* is a test. If *X* is false, then the *If* returns *Z*; otherwise, the value is *Y*.
- *(Numberp X)* returns true if *X* is a number; otherwise, it returns false.
- *(Zerop X)* returns true if *X* is *0* or is not a number; otherwise, it returns false.
- *(Fix X)* returns *X* if *X* is a number; otherwise, it returns *0*.
- *(Lessp X Y)* returns true if *X* is numerically less than *Y*; otherwise, it returns false.
- Constants preceded by a single apostrophe: (e.g., the constant P is 'P).

Other functions will be defined when they are used in the text. The Boyer-Moore theorem prover allows one to define functions in the Boyer-Moore Logic, and to prove theorems about those functions. The theorem prover will not validate theorems which it cannot prove from previous theorems and definitions. Thus, proofs on a mechanical theorem prover are more reliable than hand proofs, and guarantee a standard level of formality.

1.3 Our Theory

We describe fair traces by a function which takes a process name and a position in the trace and returns the next position in the trace where that process name occurs. Such functions, which we call *Next-Clock* functions, correspond to fair traces. In our theory we consider an arbitrary *Next-Clock* function; therefore all results are valid for all fair traces.

We define a function that, for the arbitrary *Next-Clock* function and any set of processes, simulates the system computation. This function and the processes are defined in the Boyer-Moore Logic. This function is an operational semantics for distributed systems. From this operational semantics, we derive several general theorems which hold for any set of processes and facilitate the proofs of safety and liveness properties. We call these general theorems *proof rules* because they are similar to the proof rules taken as axioms in other theories [3] [4]. In the case study, we define a set of processes that solves the mutual exclusion problem, and mechanically verify that solution using the proof rules.

Chapter 2
THE MODEL

2.1 Next-Clock

We now define the set of Next-Clock functions, and examine their relationship to fair sequences.

2.1.1 Notation

First some notation describing sequences:

Definition 1: An *(infinite) total sequence* with *alphabet A* is a mapping from the non-negative integers to the set A.

Definition 2: An *(infinite) partial sequence* with *alphabet A* is a partial mapping from the non-negative integers to the set A.

Definition 3: For a sequence *T* let

- T_i denote the i'th element of the sequence (the first element is T_0).
- *T:i* denote the first *i* elements of *T*.
- <*e*> denote the singleton sequence with element *e*.
- ; denote sequence concatenation.

Definition 4: A sequence *T* is *fair to R* iff $(\forall\, p{:}\ p \in R : (\forall\, i{:}\ i \geq 0{:}\ (\exists\, j{:}\ j \geq i{:}\ T_j = p)))$

If *T* is fair to *R* then each element of *R* appears infinitely often in *T*.

2.1.2 The Next-Clock Function

We will now formally introduce the *Next-Clock* functions and prove their close relationship to fair sequences. The relationship holds for sequences with any alphabet but it may be helpful to think of sequences of processes (traces of distributed systems).

Let *U* denote a countably infinite set and *I* denote the integers. A *Next-Clock* function is a mapping:

$$\textbf{Next-Clock} : \mathbf{U} \times \mathbf{I} \rightarrow \mathbf{I} \qquad \textbf{(1)}$$

such that for all $p, q \in U$ and $i \in I$:

```
Next-Clock(p, i) ≥ i                                              (2)
Next-Clock(p, i) = Next-Clock(q, i) → p=q                         (3)
Next-Clock(p,i) > i → Next-Clock(p,i+1) = Next-Clock(p,i)         (4)
```

Next-Clock(p, i) may be viewed as the next time at or after time *i* at which process *p* is scheduled. (2) states that *Next-Clock(p, i)* is not before *i*. (3) implies that only one process is scheduled at a given time. (4) requires that processes are not rescheduled (e.g., if *Next-Clock(p, 1) = 3* then *Next-Clock(p, 2) = 3*).

Next-Clock functions are idempotent on their second argument.

```
Next-Clock(p, i) = Next-Clock(p, Next-Clock(p, i))                (5)
```

The proof of this, from equations (2)-(4), is by upward induction on *i*.

2.1.3 Next Clock Functions and Fair Sequences

A *Next-Clock* function defines a sequence.

Definition 5: The sequence *T corresponding* to a *Next-Clock* function *NC* is:

T_i **=** p **iff** $NC(p,i)=i$**.**

We know from equation (3) that there is at most one *p* such that $NC(p,i) = i$ so *T* is well defined. However, since there may be an i such that ($\forall$ *p* :: $NC(p, i) \neq i$), *T* is a partial sequence. Partial sequences can be made total by reindexing. Reindexing is accomplished by restriction:

Definition 6: The restriction of sequence *V* to alphabet *A* is *V/A*:

```
i := 0;
j := 0;
while TRUE do
    if (V_i ∈ A) then (V/A)_j := V_i;
                        j := j + 1;
    i := i + 1;
```

The reindexing of *T* is *T\U*. We can also restrict a sequence to a total sequence with a smaller alphabet. If $A \subseteq U$ then *T/A* is a total trace with alphabet *A*.

Now consider the set of all *Next-Clock* functions and the set of all fair sequences. They are related by the following theorem:

Theorem 7: NEXT-CLOCK THEOREM: A sequence *T* with alphabet $A \subseteq U$ is fair to *A* iff it is the restriction to *A* of a sequence corresponding to a *Next-Clock* function.

This theorem is actually quite intuitive. Fairness in a process sequence corresponds to the idea that every process in the system is still "alive." Thus, every process eventually appears again in the sequence. A Next-Clock function is intuitively a skolemization of exactly when this next appearance will be.

This theorem is important because it shows that our definition of *Next-Clock* is appropriate for characterizing fair traces. Also, the proof shows that *Next-Clock* functions exist; hence, the definition of *Next-Clock* is consistent.

The proof of this theorem is quite intuitive. In a sequence fair to alphabet *A* every element appears infinitely often; Next-Clock is a skolemization of the next appearance.

In the proof that follows, alphabet *A* is implicitly quantified over all subsets of *U*. A *Next-Clock* function, *NC*, is implicitly quantified over all *Next-Clock* functions

We will first show that any restriction to *A* of a sequence corresponding to a *Next-Clock* function is fair to *A*; and then show that any sequence fair to its alphabet *A* is the restriction to *A* of a sequence corresponding to some *Next-Clock* function.

Theorem 8: The sequence *T* corresponding to a *Next-Clock* function *NC* is fair to *U*.

Proof: **Observe that for any NC**
the corresponding T is fair to U
= { def of fairness (4)}
$(\forall\ p\!: p \in U\ (\forall\ i\!: i \neq 0 : (\exists\ j\!: j \geq i\!: T_j = p)))$
Observe for all p, i
let j = NC(p,i) (j $\geq$ **i by (2))**
$\rightarrow$ **{(5)}**
j = NC(p, NC(p,i))
=> { def j}
j = NC(p, j)
=> {def correspondence (definition 5)}
T.j = p

Then we must show that the restriction of *T* to *A* yields a sequence fair to *A*.

Theorem 9: *T* fair to *U* $\rightarrow$ *T/A* fair to A.

Proof: Requires two sublemmas.

Lemma (*i*): Any sequence fair to U is fair to A.
Proof: immediate from definition (4).

Lemma (*ii*): For any sequence V, V fair to A $\rightarrow$
V/A fair to A.
Proof: First we construct the function elem(A,V,i)
which returns the number of elements of A occurring

in V with index less than i:

Definition 10:

$$\mathbf{elem(A,V,0) = 0}$$
$$\mathbf{elem(A,V,i+1) = 1 + elem(A, V, i)\ if\ V_i \in A}$$
$$\mathbf{\qquad\qquad elem(A, V, i)\ otherwise}$$

One can then show by induction on i that:

$$\mathbf{V_i \in A \rightarrow V_i = V/A_{elem(A, V, i)}} \qquad \mathbf{(6)}$$

Thus V/A is fair to A by definition (4).

Now, T is fair to U
$\rightarrow$ {lemma (*i*)}
T is fair to A
$\rightarrow$ {lemma (*ii*)}
T/A is fair to A.

Thus, any restriction to *A* of a sequence corresponding to a *Next-Clock* function is fair to *A*.

For the other part of the *iff*, for any sequence *V*, fair to its alphabet *A*, we construct a *Next-Clock* function *NC* which corresponds to a sequence which when restricted to *A* yields *V*.

We begin by constructing a sequence *T* from *V* which is fair to *U* but which, when restricted to *A* (the alphabet of *V*), yields *V*. Let *W* be a sequence fair to *U/A*. (*U/A* is the set difference of *U* and *A*.)

$$\mathbf{T = V\ |||\ W} \qquad \mathbf{(7)}$$

Definition 11: ||| is the sequence interleaving operator:

$$\mathbf{(S1\ |||\ S2)_i = S1_{i\ div\ 2}\ if\ i\ even}$$
$$\mathbf{\qquad\qquad S2_{i-1\ div\ 2}\ if\ i\ odd}$$

div is integer division.

It remains to show that such a *W* exists:

Theorem 12:

For any countably infinite set C there exists a sequence fair to C.

Proof: If C is countably infinite then so is C $\times$ C. Let e be an enumeration of C $\times$ C. We can view e as a sequence with alphabet C $\times$ C. Let W be the sequence of the first components of e. Since every possible pair (c1, c2) with c1 $\in$ C, c2 $\in$ C must occur in e thus every first element must appear infinitely often in e and thus in W.

T is fair to *U* by:

Theorem 13:

```
If S1 is fair to A1 and S2 is fair to A2 then (S1 ||| S2)
is fair to (A1 ∪ A2).

Proof:  by definition of fairness (definition 4) and
interleaving (definition 11).
```

We show that the restriction of *T* to *A* yields *V*.

Theorem 14: *T/A = V*

Proof: We will need one lemma:
Lemma (*iii*): elem(A, (V ||| W), i) = i div 2.
Proof: If i is even then by the definition of |||,
$(V\ |||\ W)_i = V_{i\ div\ 2}$.
Thus, $(V\ |||\ W)_i \in A$ (since A is the alphabet of V).
Similarly if i is odd, $(V\ |||\ W)_i$ ¬in" A. Thus, by
induction on i, elem(A, (V ||| W), i) = i div 2.

Observe for any i

$(T/A)_i$
= {equation (7)}
$((V\ |||\ W)/A)_i$
= {let j = 2i and by Lemma (*iii*)}
$((V\ |||\ W)/A)_{elem(A,\ (V\ |||\ W),\ j)}$
= {equation (6) since even j $\rightarrow$ $(V\ |||\ W)_j \in A$}
$(V\ |||\ W)_j$
= {definition of j}
$(V\ |||\ W)_{2i}$
= {definition 11}
V_i

Finally we construct from *T* a *Next-Clock* function *NC* which corresponds to *T*.

$$NC(p,\ i) = (Min\ j:\ j \geq i:\ T_j = p) \qquad (8)$$

Min is the minimum function.

NC is a Next-Clock function. (2) is immediate from the restriction $j \geq i$. (3) is just:

$NC(p,i) = NC(q,i)$
$\rightarrow$ {def of NC (8)}
$(MIN\ j:\ j \geq i:\ T_j = p) = (MIN\ j:\ j \geq i:\ T_j = q)$
$\rightarrow$ {def of Min}
$(\exists\ k::\ (T_k = p) \wedge (T_k = q))$
$\rightarrow$
$p = q$

Finally (4) follows from the observation that for any predicate *w*:

```
(Min j: j ≥ i: w(j)) > i) →
(Min j: j ≥ i: w(j)) = (Min j: j ≥ i+1: w(j))
```

We complete the proof by:

Theorem 15: NC corresponds to T.

Proof: Let Z be the sequence corresponding to NC by (5).
Observe for any i
Z_i
= {definition 5 with p = T_i}
T_i iff NC(T_i, i) = i
= {def of NC (equation 8)}
T_i iff (Min j: j $\geq$ i: T_j = T_i) = i
= {def of Min}
T_i

Q.E.D.

2.1.4 Interpretation of Next Clock

If we take *U* to be a set of process names then a *Next-Clock* function is a description of a fair trace. One should consider *Next-Clock* to be an arbitrary *Next-Clock* function. Hence, it describes an arbitrary fair trace. We can identify points of interest in this trace. Let **P** and **Q** be distinct process names. For instance, the first time at or after *Clock* that process **P** is scheduled is:

```
(Next-Clock P Clock)                                        (9)
```

However, the first time after *Clock* that process **P** is scheduled is:

```
(Next-Clock P (Add1 Clock))                                 (10)
```

The *Add1* is necessary to ensure that *Next-Clock* does not return *Clock*, if indeed the process is scheduled at *Clock*. Using these two techniques, the term describing the second time at or after *Clock* that process **P** is scheduled is:

```
(Next-Clock P                                               (11)
            (Add1 (Next-Clock P Clock)))
```

The *Add1* on the inner *Next-Clock* is required for reasons similar to those given in equation (10). Other, more complicated manipulations of *Next-Clock* are possible. We conclude with two more examples here. The earliest time at or after *Clock* such that either process **P** or **Q** is scheduled is:

```
(Min (Next-Clock P Clock)                                   (12)
     (Next-Clock Q Clock))
```

Min is the minimum function. No matter which *Next-Clock* value is returned, the other process is not scheduled in the interval between *Clock* and that value. The earliest time at or after *Clock* such that both **P** and **Q** have been scheduled is:

```
(Max (Next-Clock P Clock)                                    (13)
     (Next-Clock Q Clock))
```

Max is the maximum function. The proof that this equation (13) is equal to the more complicated, but perhaps more obvious, formula is left to the reader:

```
(Min (Next-Clock P                                           (14)
                 (Next-Clock Q Clock))
     (Next-Clock Q
                 (Next-Clock P Clock)))
```

Given these examples, it should be clear it is possible to determine the clock value associated with any point in the computation. In fact, it is often necessary to write functions that compute these clock values during the course of a liveness proof.

2.2 The Operational Semantics

In this section, we develop an operational semantics for distributed systems.

Recall that a system computation is a sequence of system states:

$$S = S_0,\ S_1,\ S_2,\ \ldots \tag{15}$$

Each subsequent S_{i+1} is the result of the transition effected on S_i by the process executed at i. This process may be identified using the *Next-Clock* function. For any set of processes, *Process-List*, and any value *Clock*, *Choose-Fairly* collects all the processes that *Next-Clock* maps to that same value of *Clock* (notice, however, that the resulting set has at most one element). Formally:

Definition 16: (Choose-Fairly Process-List Clock) =

```
(If (Listp Process-List)
    (If (Equal (Fix Clock) (Next-Clock (Car Process-List)
                                        Clock))
        (Cons (Car Process-List)
              (Choose-Fairly (Cdr Process-List) Clock))
        (Choose-Fairly (Cdr Process-List) Clock))
    Nil)
```

Choose-Fairly identifies the process description scheduled at a particular clock value. Given any initial state, the system state at the end of an interval is computed by applying the process chosen by *Choose-Fairly* at the beginning of that interval to the initial state, and repeating with the new state and the rest of the interval. This is done by the function *Int* (mnemonic for *Interpreter*), for a set of processes *Process-List*, an interval from *Clock* to *Max-Clock*, and an initial state *System-State*:

Definition 17: (Int Process-List Clock Max-Clock System-State) =

```
(If (Lessp Clock Max-Clock)
    (Int Process-List
         (Add1 Clock)
         Max-Clock
         (Single-Step (Choose-Fairly Process-List
                                     Clock)
                      Process-List
                      Clock
                      System-State))
    System-State)
```

Single-Step is a function which applies a single process (its first argument) to the system state and returns the new state. If no process is scheduled, then *Single-Step* returns the unchanged system state:

Definition 18: (Single-Step Process-Description Process-List Clock System-State) =

```
(If (Listp Process-Description)
    (Apply$ (Caar Process-Description)
            (List (Cadar Process-Description)
                  (Family-Size (Caar Process-Description)
                               Process-List)
                  (Fix Clock)
                  System-State))
    System-State)
```

Single-Step checks the process description. Since *Choose-Fairly* returns *Nil* (the empty list) when no process is scheduled, the false branch of the *If* simply returns the system state; this is the identity transition. When a process is scheduled, however, the value returned is the function application of that process on several arguments. For now, we simply claim that the *Apply$* does that function application. When we describe the process specification, we will explain *Single-Step* more completely.

Using *Int*, we can generate the infinite computation by varying only *Max-Clock* (and keeping the other arguments constant):

```
S = (Int Process-List 0 0 Initial-State),                        (16)
    (Int Process-List 0 1 Initial-State), ...
```

Furthermore, we can identify the final system state associated with any interval of the computation using:

```
(Int Process-List Clock Max-Clock System-State)                  (17)
```

Each argument is then instantiated appropriately. For example, if we want to identify the system state following the next run of process **P**, we instantiate *Max-Clock* with *(Add1 (Next-Clock* **P** *Clock))*.

Since *Next-Clock* is an arbitrary Next-Clock function, theorems about *Int* are valid for all Next-Clock functions. The trace generated by *Choose-Fairly* is a partial trace with alphabet *Process-List*. Since *Single-Step* treats undefined indices as the identity transition, this trace generates a system computation equivalent to the system computation that would be generated from this trace were it restricted to *Process-List* (albeit with repeated states).

This restricted trace is the restriction to *Process-List* of the trace corresponding to *Next-Clock*. Thus, by the Next-Clock Theorem (7), theorems about *Int* are valid for all and only fair traces.

2.3 The Process List

The first argument to *Int* is the process list. The process list is a set describing the processes in the system. This section defines the process list and its syntax, and identifies the requirements and constraints that influenced its design.

Definition 19: A process description is an identifier for a single process.
Definition 20: A process list is a set of process descriptions.
Definition 21: An indexed process is a single process in an array of processes.

Using indexed processes, one specification may define an arbitrarily large number of processes, each of which differ only by an index.

The process list design must meet certain criteria. Each process description in the process list must have the same format. This format must be sufficiently general to allow for indexed processes. Furthermore, we wish to describe the process list which contains an identifier for each process in the system (the linear representation) more abstractly (by a non-linear representation). We must then have functions that map from a non-linear representation to a linear one. Finally, the representations of the two levels must provide a simple methodology for lifting proofs about the behavior of single processes to proofs about the behavior of the entire system.

2.3.1 The Non-Linear Representation

We begin by describing the non-linear representation. The processes in a distributed system are sufficiently described by a set of families of processes, each family having a name and a size. The name of a family is the name of the function specifying the behavior of processes in that family. The size corresponds to the number of instances of processes in that family. A family size of *N* will correspond to indices from *0* to *N-1*. This representation is called a *system description*, and is represented by a list of lists, where each sublist has two elements: the family name and the family size. As an example, a system consisting of only one family, named **P**, with family size one, would have the following system description:

```
(List (List 'P 1))                                        (18)
```

2.3.2 The Linear Representation

The linear representation of a system description is a process list. The process list is a set of process descriptions each of which is a list of two elements: the name of the family that the process belongs to, and that process's index. For example, the process list corresponding to the system description given in equation (18) is:

```
(List (List 'P 0))                                        (19)
```

The system description specifies only one process, with family name **P** and index zero. As another example consider a system description of two families, **P** and **Q**, with family sizes *m* and *n*, respectively. The system description is:

```
(List (List 'P m)                                         (20)
      (List 'Q n))
```

The corresponding process list is:

```
(List (List 'P 0) ··· (List 'P m-1)                       (21)
      (List 'Q 0) ··· (List 'Q n-1))
```

Of course, any permutation of the previous list is suitable. This two level representation is convenient because it provides *Int* with the linear process list it expects, while giving the user the ability to describe systems in a concise manner. Furthermore, although in these examples this method is used to describe singly subscripted processes, this method is easily generalized to multi-subscripted processes.

2.3.3 The Technical Advantage

At a more technical level, however, this standard two level representation provides another advantage. We first define the function that builds a process list for a single family:

Definition 22: (Make-Family Family-Name Current-Index) =

```
(If (Zerop Current-Index)
    Nil
    (Cons (List Family-Name (Sub1 Current-Index))
          (Make-Family Family-Name
                       (Sub1 Current-Index))))
```

Make-Family, given a family name, *Family-Name* and an index, *Current-Index*, generates a list of elements of the form *(List Family-Name i)* for all *i* between *0* and *Current-Index - 1*.

Using *Make-Family*, we define the function that maps between system descriptions and process lists:

Definition 23: (Make-Process-List System-Description) =

```
(If (Listp System-Description)
    (Append (Make-Family (Caar System-Description)
                         (Cadar System-Description))
            (Make-Process-List (Cdr System-Description)))
    Nil)
```

For all family name, family size pairs, *Make-Process-List* generates the corresponding process lists and appends all of the lists together.

When process lists are formed from system descriptions using the function *Make-Process-List*, we can specify necessary and sufficient conditions for process descriptions to be elements of the process list. This is expressed in the following theorem:

Theorem 24:

```
(Implies (Setp (Family-Names System-Description))
         (Equal (Member Process-Description
                        (Make-Process-List
                          System-Description))
                (And (Numberp (Cadr Process-Description))
                     (Lessp (Cadr Process-Description)
                            (Declared-Family-Size
                              (Car Process-Description)
                              System-Description))
                     (Member (Car Process-Description)
                             (Family-Names
                               System-Description))
                     (Equal (Length Process-Description)
                            2)
                     (Equal (Cddr  Process-Description)
                            Nil))))
```

Declared-Family-Size is the function that returns the family size of its first argument as specified in the system description (its second argument). This theorem states that if a system description contains no repeated families, then every process description in the corresponding process list must have several characteristics. Furthermore, every process description with those characteristics is in the process list. These characteristics are an abstract characterization of valid process descriptions. Using these characteristics, the behavior of every instance of a family can be described by a theorem stating the behavior of a single instance with an arbitrary index (less than that family's family size). This theorem, in conjunction with the proof rules, is the interface between process level properties and system level properties.

2.4 Specification of Processes

Each family has a specification. The specification is a function in the Boyer-Moore Logic. The specification of a process is the specification of its family with the process's index and family size instantiated. Every family specification takes four arguments:

1. *Index*: The instance of the process.
2. *Family Size*: The family size.
3. *Clock*: The clock in *Int*. Since it varies from transition to transition, it may be used to model non-determinism.
4. *System State*: The state of the system.

Given these arguments, each specification specifies the resulting system state.

2.4.1 An Example Specification

As an example, consider the specification of a family of processes which copy messages from one channel to another. At each transition, a process should read a message from its left channel, and send that message on its right channel. The name of the family should be *Copy*.

Definition 25: (Copy Index Family-Size Clock System-State) =

```
(If (Channel-Emptyp (Cons 'Left Index) System-State)
    System-State
    (Send (Get-Head (Cons 'Left Index) System-State)
          (Cons 'Right Index)
          (Receive (Cons 'Left Index) System-State)))
```

Send and *Receive* are the appropriate list functions to model channels; *Get-Head* returns the first message in a channel; *Channel-Emptyp* tests whether the channel is empty. Each process has its own left and right channel: the name of the left channel is *(Cons 'Left Index)* (the *Index*'th left channel); the name of the right channel is *(Cons 'Right Index)* (the *Index*'th right channel). If the left channel is empty, *Copy* returns the original system state. If not, *Copy* returns the system state altered by sending the message from the head of the left channel upon the right channel, and shortening the left channel.

There are several important points in this example. First, this definition of *Copy* does not depend upon the value of *Family-Size*. This is because its copying is independent of the number of other indexed *Copy* processes. Second, *Copy* does not use its *Clock* argument, because it is deterministic.

Third, the Boyer-Moore Logic defines no notion of channels; we assume that channels are satisfactorily modeled by lists, where receive operations shorten the list, and send operations append messages to the end of the list. The definitions of *Channel-Emptyp*, *Get-Head*, *Send*, and *Receive* are simply definitions satisfying that model.

Finally, we make no claim that processes ought to be specified or be implemented in a functional language. The only claim we make is that it is convenient to prove properties about functional specifications. These functional specifications may be translated into more efficient ones; conversely, specifications in other notations may be translated to their functional equivalents.

2.4.2 Interaction with Single-Step

All specifications have the same argument list because *Single-Step*'s interface with a process is static. Hence, all the information that any process may need is in the argument list, even though many specifications may not need all the arguments. Let us now examine *Single-Step* again, in the context of the *Copy* specification.

```
(Single-Step Process-Description                                (22)
              Process-List Clock System-State)
=
(If (Listp Process-Description)
    (Apply$ (Caar Process-Description)
            (List (Cadar Process-Description)
                  (Family-Size (Caar Process-Description)
                               Process-List)
                  (Fix Clock)
                  System-State))
    System-State)
```

Family-Size is the analog of *Declared-Family-Size* for process lists. It returns the number of instances of a particular family (its first argument) in the process list (its second argument).

Let us suppose that *Process-Description* is a list containing the process description of the zero'th instance of the *Copy* family defined above (just like *Choose-Fairly* would return). That is, *Process-Description* is:

```
(List (List 'Copy 0))                                    (23)
```

Let us further assume that the family size of *Copy* is *N*. *Single-Step* then returns the value of the *Apply$*. Instantiating the *Apply$* yields:

```
(Apply$ 'Copy                                            (24)
        (List 0
              N
              (Fix Clock)
              System-State))
```

Informally, this *Apply$* is equivalent to a function call of *Copy* on the elements of the list that is the second argument to *Apply$*[1]. The arguments are used in order. So the *Apply$* becomes:

```
(Copy 0 N (Fix Clock) System-State)                      (25)
```

This is the expected call to *Copy*. As an aside, *Copy* may be used to specify a minimum channel delay, to prevent instantaneous channel communication.

2.4.3 Implementability

An algorithm is *implementable* if it may be realized upon a real machine. Hence, the implementability of a specification is dependent upon both the algorithm and the target architecture. Given a specification and an architecture, the notion of implementability may be split into two distinct questions:

- Are all the operations that the specification requires available on the target architecture? For example, if the specification requires channel communication, does the architecture support channels? The questions may be more involved: does the specification require shared memory? Is the memory model appropriate for the specification available on the architecture?
- Are all of the transitions required by the specification guaranteed to be atomic on the target architecture? The transition model of distributed systems requires that all transitions terminate with the expected result even if other process are running concurrently. This is a reasonable assumption because one can design specifications that do not require transitions which are not atomic on the target architecture. For example, on a message based architecture, with only local memory, it is reasonable to expect that a process may access its memory an arbitrary number of times in any transition, since no other process may access that memory. On the other hand, it may not be reasonable to expect that a process be able to send and receive an arbitrary number of messages during a single transition, since the communication devices may be updated during the transition.

Since the notion of implementability and the correctness of that implementation is quite a different matter from proving properties of specifications within the transition model of distributed systems, we do not make any

[1]The actual definition of *Apply$* is more complicated, but when its first argument is the name of a total function, this informal definition is satisfactory.

attempt to formalize architectural limitations, or to prove that a specification is realizable on a particular architecture. It is easy to devise a programming language only expressive enough to allow programs for a particular architecture. If a specification may be translated into that programming language, then the specification may be realized on that architecture.

2.5 Proof Rules

The operational semantics defined by *Int* is sufficient to prove safety and liveness properties for distributed systems. However, such proofs are facilitated by proof rules. The proof rules we develop in this section offer an abstract mechamism for verifying distributed systems. We have stated the proof rules in a version of the Boyer-Moore logic which defines second order functions [2], and have mechanically verified their derivation from the definition of *Int*.

2.5.1 Compositionality

The first proof rule states that *Int* is composable:

Theorem 26: Compositionality

```
(Implies (And (Not (Lessp Time Clock))
              (Not (Lessp Max-Clock Time)))
         (Equal (Int Process-List
                     Time
                     Max-Clock
                     (Int Process-List Clock
                          Time System-State))
                (Int Process-List Clock
                     Max-Clock System-State)))
```

This theorem states that two adjacent intervals of the computation may be composed into one, using *Int* over the larger interval.

2.5.2 Invariance

The second proof rule is used to prove invariants. Invariants are properties that are preserved by the system throughout the computation. That is, if an invariant holds on the initial state, it will hold on all subsequent states. For any property **Q** and process list *Process-List*, invariance theorems have the following form:

```
(Implies (Q System-State)                                   (26)
         (Q (Int Process-List
                 Clock
                 Max-Clock
                 System-State)))
```

Since *Clock Max-Clock* and *System-State* are universally quantified, such a theorem states that if **Q** holds on *System-State* it will hold on every future point in the computation.

The invariance proof rule should have a consequent that looks like equation (26). The only interval of the computation mentioned in the proof rule is *Clock* to *Max-Clock*. Therefore, the antecedent should state that **Q** is preserved over every transition in that inverval. Each transition is effected by *Single-Step* on the current system state when passed the process description chosen by *Choose-Fairly*. The antecedent is captured by the following definition:

Definition 27: (Invariance-Over-Single-Step Process-List Clock Max-Clock System-State) =

```
(∀ Time: Clock ≤ Time < Max-Clock:²
   (Implies (Q (Int Process-List
                    Clock
                    Time
                    System-State))
            (Q (Single-Step (Choose-Fairly Process-List
                                           Time)
                            Process-List
                            Time
                            (Int Process-List
                                 Clock
                                 Time
                                 System-State)))))
```

Since *System-State* is the state at the beginning of the interval, the current system state at any *Time* in the interval is:

```
(Int Process-List
     Clock
     Time
     System-State)
```

Definition (27) says that every transition in the interval preserves **Q**. That is, if the current system state satisfies **Q**, then the next system state also satisfies **Q**.

The invariance equation (26) and definition (27) are then combined to form the proof rule:

Theorem 28: Invariance Proof Rule

```
(Implies (Invariance-Over-Single-Step Process-List
                                      Clock
                                      Max-Clock
                                      System-State)
         (Implies (Q System-State)
                  (Q (Int Process-List
                          Clock
                          Max-Clock
                          System-State))))
```

To use this proof rule to prove invariance properties, one must prove theorems of the following form for each family *Family* with family size *Family-Size*:

---

[2]Appendix I discusses the expression of embeded quantification in the Boyer-Moore Logic.

Theorem 29:

```
(Implies (And (Lessp Index Family-Size)
              (Numberp Index)
              (Q System-State))
         (Q (Family Index
                    Family-Size
                    Clock
                    System-State)))
```

Such theorems state that every valid instance of a family preserves property **Q**. From such theorems one appeals to the invariance proof rule to prove system level invariance properties (equation (26)).

2.5.3 Progress

A progress property states that if a property **Q** holds for some state then a property **R** will hold at a later state. The basic progress proof rule uses the following scheme: assume a property **Q** is preserved by all processes except for one key process and that key process changes a system state satisfying property **Q** to a system state satisfying property **R**. Under this assumption, if the system starts in a state satisfying **Q** then it will satisfy **R** right after the key process runs.

Assume that the key process's name is **P**, the initial system state is *System-State*, and the initial clock is *Clock*. Since **Q** is preserved by all processes but **P**, and **P** is not scheduled during the interval *Clock* to *(Next-Clock* **P** *Clock)*, **Q** is stable over that interval. Furthermore, **P** is scheduled at *(Next-Clock* **P** *Clock)*, so, **R** will hold subsequently. This is stated in the following scheme for progress proofs, given a process list *Process-List* and process named **P** in that process list.

Theorem 30:

```
(Implies (Q System-State)
         (R (Int Process-List
                 Clock
                 (Add1 (Next-Clock P Clock))
                 System-State)))
```

To run *Int* through *(Next-Clock* **P** *Clock)*, it must be passed a *Max-Clock* that is one greater.

The actual progress proof rule is more general. It has the following antecedents:

- The property **Q** is stable over an interval.
- If the interval ends satisfying **Q** then the property **R** holds one step after the end of the interval.

The proof rule is:

Theorem 31: Progress Proof Rule

```
(Implies (And (Lessp Clock Max-Clock)
              (Invariance-Over-Single-Step Process-List
                                           Clock
                                           (Sub1 Max-Clock)
                                           System-State)
              (Implies (Q (Int Process-List
                                 Clock
                                 (Sub1 Max-Clock)
                                 System-State))
                       (R (Single-Step
                            (Choose-Fairly Process-List
                                           (Sub1 Max-Clock))
                            Process-List
                            (Sub1 Max-Clock)
                            (Int Process-List
                                 Clock
                                 (Sub1 Max-Clock)
                                 System-State)))))
         (Implies (Q System-State)
                  (R (Int Process-List
                          Clock
                          Max-Clock
                          System-State))))
```

To use the proof rule to prove basic progress properties, one instantiates *Max-Clock* to the *Add1* of *Next-Clock* of the key process. A more complex progress property, in which any one of a set of key processes changes a system state satisfying property **Q** to a system state satisfying property **R**, would be proved by instantiating *Max-Clock* to the minimun of the *Next-Clock*'s of the key processes.

The invariance and progress proof rules allow one to prove properties about processes and automatically lift those properties to system level properties.

Chapter 3
A CASE STUDY

In this section, we present a mechanically verified solution to the mutual exclusion problem. Since there are many well known solutions to the mutual exclusion problem, we concentrate on specifying the problem in our formalism, and on presenting the techniques that we used to mechanically verify its solution.

3.1 The Informal Specification

We define a process *Me*, which is a member of a ring of *N* similar processes connected by *N* channels. This process has three states: *Non-Critical*, *Wait*, and *Critical*. *Me* non-deterministically changes from *Non-Critical* to *Wait*. If *Me* is *Waiting* and has a token upon its left channel, it absorbs the token and becomes *Critical* for an arbitrary, but finite, number of transitions. When *Me* leaves its *Critical* state, it releases a token upon its right channel and becomes *Non-Critical*. If *Me* is *Non-Critical* and does not change to *Wait*, it copies a token from its left to its right channel, if a token is available.

The system, consisting of *N* instances of *Me*, satisfies two properties:

- Mutual Exclusion: At most one process is *Critical* at any time.
- Liveness: Any *Waiting* process eventually becomes *Critical*.

This is the system specification. (References to simply specification refer to the specification of the process *Me*.)

In this specification, we assume a message based architecture with only local memory. We further assume that it is reasonable for a process to do as much as test its left channel, read from its left channel, and send on its right channel during a single transition. If these assumptions are not appropriate for another architecture, then the longer transitions may be split into a sequence of shorter ones. (Of course, the proofs would have to be altered).

Figure 3-1: Transitions of Process *Me*

Figure 3-2: Ring of N instances of *Me*

3.2 The Formal Specification

We now present the formal specification for the process *Me*:

Definition 32: (Me Index Family-Size Clock System-State) =

```
(i)   (If (Equal (Get-State (Cons 'Me Index) System-State)
                 'Non-Critical)
(ii)     (If (Random-Boolean Clock)
              (Update-State (Cons 'Me Index)
                            'Wait
                            System-State)
(iii)       (If (Channel-Emptyp (Cons 'C Index)
                                 System-State)
                 System-State
                 (Send 'Token
                     (Cons 'C (Add1-Mod Family-Size Index))
                     (Receive (Cons 'C Index)
                                System-State))))
(iv)     (If (Equal (Get-State (Cons 'Me Index) System-State)
                    'Wait)
(v)         (If (Channel-Emptyp (Cons 'C Index) System-State)
                System-State
                (Update-State (Cons 'Me Index)
                              (Random-Number Clock)
                              (Receive (Cons 'C Index)
                                       System-State)))
(vi)        (If (Zerop (Get-State (Cons 'Me Index)
                                   System-State))
                 (Update-State (Cons 'Me Index) 'Non-Critical
                               (Send 'Token
                                     (Cons 'C (Add1-Mod
                                                Family-Size
                                                Index))
                                     System-State))
                 (Update-State (Cons 'Me Index)
                               (Sub1 (Get-State
                                        (Cons 'Me Index)
                                        System-State))
                               System-State))))
```

The argument list to *Me* is the process's *Index*, the *Family-Size* (it is *N*), the *Clock*, and the *System-State*. Each process identifies its own state within the system state by the name *(Cons 'Me Index)*, and accesses its state by *(Get-State (Cons 'Me Index) System-State)*. In a similar manner, channels are named *(Cons 'C Index)*. In this system, the left channel of the *Index*'th instance of *Me* is *(Cons 'C Index)* and its right channel is *(Cons 'C (Add1-Mod Family-Size Index))* (add one modulo the family size). *Update-State* replaces the state identified by its first argument, with its second argument.

The body of *Me* defines the process. The first *If* (*i*) tests whether the process is currently *Non-Critical*. If it is, then a random boolean value is checked (*ii*) to see whether the process should *Wait* to become *Critical*. If the random boolean value is true, the process's state becomes *Wait*. If not, the process remains *Non-Critical*, and sends a token from the left to the right channel, if (*iii*) there is a token on the left channel.

If the process is not *Non-Critical*, the *If* at (*iv*) tests whether the process is currently in its *Wait* state. If it is, then if (*v*) the left channel is empty, the process remains in the *Wait* state. If the left channel is not empty, the process receives a token (shortens the channel), and changes its state to a random number, signifying the *Critical* state. This value is referred to as the *critical counter*.

If the process is neither *Non-Critical* nor *Wait*, then it is *Critical*. The *If* at (*vi*) tests whether the process has any time left in its *Critical* state. If its critical counter is zero, then the process becomes *Non-Critical* and sends a token upon its right channel. If its critical counter is greater than zero, then the process decrements its state (and remains *Critical*).

The first obvious feature of this specification is that it does not manufacture tokens. That is, a process will never add tokens to the system. A process becomes *Critical* by absorbing a token, and becomes *Non-Critical* by releasing one. This arrangement is necessary to preserve mutual exclusion.

The second obvious features in this specification is that a token will move along the ring. No process, not even a *Critical* one, will keep a token forever. This property is necessary for liveness proofs.

The subtle point in this specification is one that will not be checked by the system specification. In this specification, the process changes from *Non-Critical* to *Wait* (*ii*) without inspecting its left channel (*iii*). If a token on the left channel were passed to the right before (and in a separate transition from) the process's change to the *Wait* state, the process might never be able to enter its *Wait* state. Both strategies satisfy the system specification; however, it is clear that a specification that can prohibit a process from entering its *Wait* state is faulty.

The functions *Random-Boolean* and *Random-Number* used in the definition of *Me* provide a notation for non-determinism necessary to complete the formalization the informal specification. Each of these functions is defined as a function of one argument with respective ranges of boolean and non-negative integers. Since we assume nothing about these function beyond these ranges, as long as each invocation of each of these functions is presented with a different argument, the value of the function at any call is unknown. This is sufficient to model non-determinism. Since the value of the *Clock* varies from transition to transition, the process uses that value as the argument to *Random-Boolean* or *Random-Number*.

We specify that the system contains *N* instances of process *Me* by defining the system description.

Definition 33: (System-Description N) =

```
(List (List 'Me N))
```

That is, the system description specifies a single family named *Me*, with *N* instances. Whenever the system description is needed in the proof, we use the term:

```
(System-Description N)                                          (27)
```

Notice that the *N* (like all variables) is universally quantified: the proofs are valid for all *N*.

Before proceeding with the proof of the invariance and liveness properties for this system, we must first prove three important properties about the system description. Three similar properties must be proved for every system description so the proof rules and other theorems may be used automatically by the prover.

The family names in the system description are unique (*Family-Names* is the function that collects all the family names in its first argument):

Theorem 34:

```
(Setp (Family-Names (System-Description N)))
```

For each family in the system description we prove a theorem of the following form (simplify the declared family size):

Theorem 35:

```
(Declared-Family-Size 'Me (System-Description N))
=
(Fix N)
```

We simplify what membership in the family names means. For systems with more than one family, the membership should rewrite to a disjunct of equalities:

Theorem 36:

```
(Equal (Member Process-Name (Family-Names
                              (System-Description N)))
       (Equal Process-Name 'Me))
```

After these three theorems are proved, *System-Description* is disabled[3] (its executable counterpart is disabled also (equation (3))).

3.3 The Specification and Proof of Invariance

The formal statement of mutual exclusion is:

Theorem 37:

```
(Implies (Mutual-Exclusionp N System-State)
         (Mutual-Exclusionp
         N (Int (Make-Process-List (System-Description N))
                 Clock
                 Max-Clock
                 System-State)))
```

The first argument to *Int*, *(Make-Process-List (System-Description N))*, is the process list containing the *N* instances of process *Me*. Given the proper definition for *Mutual-Exclusionp*, this formula specifies mutual exclusion for the system described above. It reads:

- If a system state satisfies mutual exclusion, then the system state at any subsequent point in the computation also satisfies mutual exclusion.

Since *Clock* and *Max-Clock* are universally quantified, mutual exclusion is invariant over the entire system computation (if it holds initially, it will hold forever). Hence, properties in this form are called invariance properties.

3.3.1 The Proof of Invariance

The definition of *Mutual-Exclusionp* is:

Definition 38: (Mutual-Exclusionp N System-State) =

```
(Equal (Plus (Weight-Of-Channels N System-State)
             (Weight-Of-Processes N System-State))
       1)
```

Mutual exclusion is simply the property that the weight of the system is one. The weight of the system is the sum of the weight of the processes and the weight of the channels. The weight of the processes is the number of processes that are *Critical*; the weight of the channels is the sum of the lengths of the channels.

The invariance of *Mutual-Exclusionp* (37) is proved using the invariance proof rule (8). The theorems that the

[3]Disabling a definition *discourages* the theorem prover from expanding that definition.

invariance proof rule requires should state that mutual exclusion is *stable* over the single transition of each process in the system. Since there is only one type of process (*Me*) in this system, this task is accomplished with the following theorem:

Theorem 39:

```
(Implies (And (Lessp Index N)
              (Numberp Index)
              (Mutual-Exclusionp N
                                 System-State))
         (Mutual-Exclusionp N
                            (Me Index N
                                Clock
                                System-State)))
```

This theorem states that given any *N*, and any numeric *Index* less than *N*, if mutual exclusion holds on a system state, then mutual exclusion will also hold on the system state produced by applying that instance of *Me* once. This theorem captures the stability argument for all instances of *Me*, since those instances have indices ranging from *0* to *N-1*.

This theorem is the only statement necessary to prove the system level invariance of (37).

To prove that mutual exclusion is stable over *Me* (39), we prove that the weight of the system equals the weight of the system subsequent to a transition of *Me*. We do this by breaking the proof into two parts, based on the value of *N*.

For *N* with values of zero or one, this property is true by inspection. The following theorem suffices:

Theorem 40:

```
(Implies (And (Not (Lessp 1 N))
              (Numberp Index)
              (Lessp Index N))
         (Equal (Plus (Weight-Of-Channels N
                                          (Me Index N
                                              Clock
                                              System-State))
                      (Weight-Of-Processes N
                                           (Me Index N
                                               Clock
                                               System-State)))
              (Plus (Weight-Of-Channels N System-State)
                    (Weight-Of-Processes N System-State))))
```

For values of *N* greater than one, we proceed a bit more slowly. For any transition, we split the system state into two fragments with stable weights. The first fragment, called a *triple*, is composed of a process along with its left and right channels. For all transitions of the *Index*'th instance of *Me*, the weight of its triple is stable. This is evident by case analysis on that *Me*'s possible transitions: The left channel may be shortened, but then *Me* either becomes *Critical*, or its right channel is extended. If *Me* changes state from *Critical* to *Non-Critical*, then its right channel will be extended as well.

The proof that the weight of the triple is stable is done in one step:

Theorem 41:

```
(Implies (Lessp 1 N)
         (Equal (Weight-Of-Triple Index N (Me Index
                                              N
                                              Clock
                                              System-State))
                (Weight-Of-Triple Index N System-State)))
```

The second fragment of the system is all processes and channels in the system that are not accounted for by the triple. This fragment is not altered at all by a transition of the process in the triple. This is proved using the next two non-interference theorems:

Theorem 42:

```
(Implies (And (Not (Equal I Index))
              (Not (Equal I (Add1-Mod N Index))))
         (Equal (Cdr (Assoc (Cons 'C I) (Me Index
                                            N
                                            Clock
                                            System-State)))
                (Cdr (Assoc (Cons 'C I) System-State))))
```

(*Cdr of Assoc* is equivalent to *Get-State*). That is, the value of all channels remains unchanged by a transition of the *Index*'th instance of *Me* except for that process's left and right channels. This is true because a process does not interfere with any channel outside the triple. The next theorem is the analogous property for processes:

Theorem 43:

```
(Implies (Not (Equal I Index))
         (Equal (Cdr (Assoc (Cons 'Me I) (Me Index
                                             N
                                             Clock
                                             System-State)))
                (Cdr (Assoc (Cons 'Me I) System-State))))
```

That is, the state of every other process is left unchanged by a transition of *Me*. This is true, because any instance of *Me* only modifies its own local state.

Using these two non-interference theorems (42) and (43), we prove that the weight of the channels outside the triple, and the weight of the processes outside the triple remain constant.

Given that result, we have proved that the weights of the triple, the rest of the processes, and the rest of the channels are stable. Since the weight of the system (for *N* greater than one) is the sum of those three, it is stable too. This is proved in the following theorem:

Theorem 44:

```
(Implies (And (Lessp 1 N)
              (Numberp Index)
              (Lessp Index N))
        (Equal (Plus (Weight-Of-Channels N
                                         (Me Index
                                             N
                                             Clock
                                             System-State))
                     (Weight-Of-Processes N
                                          (Me Index
                                              N
                                              Clock
                                              System-State)))
               (Plus (Weight-Of-Channels N System-State)
                     (Weight-Of-Processes N System-State))))
```

Using this result, and the one proved earlier for *N* less than two (40), we prove that mutual exclusion is stable for a single transition of *Me* (39). This stability property implies the mutual exclusion invariance property (37), by appeal to the invariance proof rule.

The mechanical proof of mutual exclusion required twenty-nine definitions, theorems, and other events.

3.3.2 Discussion of the Proof of Invariance

Proofs of invariance have two parts:

- One first proves that the invariant is stable for a single transition of every process in the system. This is usually stated as a single theorem for each family (with the index restricted to the family size). The difficulty of the process level invariance proof, is directly related to the complexity of the processes' transitions.
- One then lifts the process level invariant to the system level invariant. This is accomplished using the invariance proof rule (28), and the theorem stating that *Choose-Fairly* returns only processes that are members of the process list (20). This part of the proof is fully automated.

3.4 The Specification and Proof of Liveness

The formal statement of the liveness property is:

Theorem 45:

```
(Implies (And (Mutual-Exclusionp N System-State)
              (Numberp Index)
              (Lessp Index N)
              (Process-Wait Index System-State))
      (Process-Critical Index
```

```
(Int (Make-Process-List
       (System-Description N))
     Clock
   (Clock-Of-Token-Moving-To-Critical
         Index N Clock System-State)
     System-State)))
```

Process-Wait tests whether the *Index*'th instance of *Me* is *Waiting*; *Process-Critical* tests whether the *Index*'th instance of *Me* is *Critical*. This theorem can be read as follows:

- If mutual exclusion holds on the initial state, and an instance of *Me* is *Waiting*, then there exists a time in the future when the state of that process is *Critical*.

3.4.1 The Proof of Liveness

The proof of this liveness property requires building the function *Clock-Of-Token-Moving-To-Critical*. The general scheme is: identify transitions in the computation that must occur for the *Waiting* process to become *Critical*. This sequence of transitions is easily translated to a composition of *Next-Clock*'s; the result is the time at which the process becomes *Critical*.

The proof is divided into two parts. The first describes the movement of the token up to the left channel of the *Waiting* process; this requires three progress properties. The second describes the interval during which a *Waiting* process with a token on its left channel becomes *Critical*; this is one progress property. We will show, in detail, the simpler second, part of the proof. Then we examine one of the properties from the first part of the proof, and conclude by deriving the function *Clock-Of-Token-Moving-To-Critical*, which coordinates the progress properties to identify the time at which the *Waiting* process becomes *Critical*.

The second part of the proof is simpler. It is a simple progress property stating that a *Waiting* process with a full left channel will become *Critical* the very first time that it runs. Notice that many other processes may be executed in the meantime. The statement of this progress property is:

Theorem 46: Progress Property Four

```
(Implies (And (Lessp Index N)
              (Numberp Index))
         (Implies (Left-Channel-Full-Process-Wait
                    Index
                    System-State)
                  (Process-Critical-With-Ticks
                    (Random-Number
                      (Next-Clock (List 'Me Index)
                                  Clock))
                    Index
                    (Int (Make-Process-List
                           (System-Description N))
```

```
Clock
(Add1 (Next-Clock (List 'Me Index)
                  Clock))
System-State))))
```

Left-Channel-Full-Process-Wait tests whether the *Index*'th channel is full and the *Index*'th process is *Waiting*. *Process-Critical-With-Ticks* tests whether the *Index*'th process is *Critical* with critical counter numerically equal to its first argument. This theorem states: If the *Index*'th instance of *Me* is *Waiting* with a full left channel, after the system executes until the first time that instance of *Me* runs, that instance of *Me* will be *Critical*. Furthermore, its critical counter will have the value:

```
(Random-Number (Next-Clock (List 'Me Index) Clock))     (28)
```

There are a few important points in this theorem. First, the interval of the computation which Int must consider is from *Clock* to *(Add1 (Next-Clock (List 'Me Index) Clock))*. The *Next-Clock* signifies the next time that the *Index*'th instance of *Me* is scheduled. The *Add1* is necessary because *Int* must compute the state up to and including that final transition. However, the critical counter will have the value *Random-Number* returns when it is passed the precise time at which that instance of *Me* runs. That time is *(Next-Clock (List 'Me Index) Clock)*.

This theorem is proved using the progress proof rule (31) and two other theorems. The first, the stability theorem, states: the property that a particular instance of *Me* is *Waiting* and that its left channel is full, is preserved over all transitions except the transition of that instance of *Me*. The second theorem states that if a particular instance of *Me* is *Waiting*, its left channel is full, and that instance of *Me* runs, then it becomes *Critical* (with a specific value on its critical counter). The stability theorem is:

Theorem 47:

```
(Implies (And (Not (Equal Cindex Index))
              (Lessp Cindex N)
              (Numberp Cindex)
              (Lessp Index N)
              (Numberp Index)
              (Left-Channel-Full-Process-Wait Index
                                               System-State))
         (Left-Channel-Full-Process-Wait Index
                                         (Me Cindex
                                             N
                                             Clock
                                             System-State)))
```

This states that the property *Left-Channel-Full-Process-Wait* is stable across any transition of *Me* other than the *Index*'th instance of *Me*. The key transition is identified with the following theorem:

Theorem 48:

```
(Implies (And (Numberp Index)
              (Lessp Index N)
              (Left-Channel-Full-Process-Wait Index
                                              System-State))
         (Process-Critical-With-Ticks (Random-Number Clock)
                                      Index
                                      (Me Index
                                          N
                                          Clock
                                          System-State)))
```

This theorem states that if *Left-Channel-Full-Process-Wait* holds before that instance of *Me* runs, then *Process-Critical-With-Ticks* will hold after that instance of *Me* runs. Furthermore, since *Me* is running at time *Clock*, the critical counter will be *(Random-Number Clock).*

This proves progress property four. In a similar manner, we prove that given two distinct processes, one *Waiting* and the other either *Non-Critical* with its left channel full, *Wait* with its left channel full, or *Critical*, when the latter is finally executed it progresses appropriately, and the former process is still *Waiting*.

Assuming that the system state satisfies mutual exclusion, this scheme corresponds to progress properties one, two, and three:

One process remains *Waiting* and the other:

1. Progresses from *Non-Critical* with left channel full to right channel full or to *Wait* with left channel full.
2. Progresses from *Wait* with left channel full to *Critical*.
3. Progresses from *Critical* to *Critical* or to right channel full.

Notice that one property, that of the *Waiting* process, is stable across all of the transitions. Also, each of the progress statements describes a final state which is suitable to initiate another progress statement. For example, (1) initiates (2) if the process becomes *Wait*. However, if the process passes the token to the right channel, (1) initiates either (1) or (2) depending on the state of the adjacent process. (2) initiates (3). Finally, (3) initiates (3), if the process remains *Critical*, or either (1) or (2) depending on whether the adjacent process is *Non-Critical* or *Wait*.

Progress property three is typical and we discuss it here.

Theorem 49: Progress Property Three

```
(Implies (And (Lessp Index1 N)
              (Numberp Index1)
              (Lessp Index2 N)
              (Numberp Index2)
              (Not (Equal Index1 Index2)))
         (Implies (Critical-Wait Ticks Index1 Index2 N
                                 System-State)
                  (Critical-Or-Non-Critical-Wait
                    Ticks Index1 Index2 N
                    (Int (Make-Process-List
                           (System-Description N))
                         Clock
                         (Add1 (Next-Clock (List 'Me Index1)
                                           Clock))
                         System-State))))
```

The function *Critical-Wait* is a predicate testing whether the *Index1*'th instance of *Me* is *Critical* with critical counter set at *Ticks*; whether the *Index2*'th instance of *Me* is *Waiting*; and, whether the system state satisfies mutual exclusion. All three of these properties are necessary preconditions to this progress proof.

The final value of the system state is generated by *Int* from *Clock* up to (and including) the time when the *Index1*'th instance of, *Me* runs. That time is *(Next-Clock (List 'Me Index1) Clock)*. That new system state satisfies the predicate *Critical-Or-Non-Critical-Wait,* which tests:

- Whether the *Index1*'th instance of *Me* has critical counter *Ticks-1*, if *Ticks* is greater than zero. Otherwise, tests whether that instance of *Me*'s right channel is full.
- Whether the *Index2*'th instance of *Me* is *Waiting*.
- Whether the system state satisfies mutual exclusion.

The reason that the predicate *Critical-Or-Non-Critical-Wait* cases on the value of *Ticks* is because a *Critical* process cases on the value of its critical counter. If the value of its critical counter is greater than zero, the process remains *Critical*; if the value is zero, the process will release a token on its right channel.

This progress statement is proved using the same methodology that was used to prove progress property four (46). First, *Critical-Wait* is shown to be stable for all transitions except for the *Index1*'th instance of *Me*; then the key transition, that of the *Index1*'th instance of *Me,* is shown to change *Critical-Wait* into *Critical-Or-Non-Critical-Wait*. The progress proof rule is used to lift these two results to progress property three (49).

Progress property three is especially interesting because it can be used repeatedly by induction on *Ticks*. That is, if *Ticks* is greater than zero, then *Critical-Or-Non-Critical-Wait* implies *Critical-Wait* using *Ticks-1*. Using induction, the computations of *Int* may be composed until the process releases a token (and becomes *Non-Critical*).

We identify this interval by defining the function *Clock-Of-Critical-To-Non-Critical*:

Definition 50: (Clock-Of-Critical-To-Non-Critical Index Clock Ticks) =

```
(If (Zerop Ticks)
    (Add1 (Next-Clock (List 'Me Index) Clock))
    (Add1 (Next-Clock (List 'Me Index)
                      (Clock-Of-Critical-To-Non-Critical
                        Index Clock
                        (Sub1 Ticks)))))
```

This function computes the time when a *Critical* instance of *Me* with critical counter *Ticks* releases a token upon its right channel. The process must run *Ticks+1* times. This is *Ticks+1* composition of *Next-Clock*'s of *(List 'Me Index)*. The *Add1* is necessary to push *Next-Clock* to the next time that the instance of *Me* runs; otherwise the entire composition collapses to a single *Next-Clock*. This arrangement is fortunate in another respect: the computation of *Int* up to and including a *Max-Clock* must be run until *Max-Clock+1*; hence the result of *Clock-Of-Critical-To-Non-Critical* may be used directly in *Int*.

This is demonstrated in the following theorem:

Theorem 51:

```
(Implies (And (Numberp Index1)
              (Lessp Index1 N)
              (Numberp Index2)
              (Lessp Index2 N)
              (Not (Equal Index1 Index2))
              (Critical-Wait Ticks Index1 Index2 N
                             System-State))
         (Right-Channel-Full-Wait
           Index1 Index2 N
           (Int (Make-Process-List (System-Description N))
                Clock
                (Clock-Of-Critical-To-Non-Critical
                  Index1
                  Clock
                  Ticks)
                System-State)))
```

That is, if *Index1*'th instance of *Me* is *Critical* with critical counter *Ticks* and the *Index2*'th instance of *Me* is *Waiting*, and the system state satisfies mutual exclusion, then the new system state generated by *Int* from *Clock* to *(Clock-Of-Critical-To-Non-Critical Index1 Clock Ticks)* will have the right channel of the *Index1*'th instance of *Me* full, and the *Index2*'th instance of *Me* still *Waiting*. (51) identifies the time in the future that the *Critical* instance of *Me* releases its token. The function *Clock-Of-Critical-To-Non-Critical* replaces an existentially quantified time: (e.g., *there exists Time, Time>Clock such that Right-Channel-Full-Wait holds*).

Using *Clock-Of-Critical-To-Non-Critical*, and progress properties one and two, similar functions may be defined for the intervals of *Wait* to *Non-Critical* and *Non-Critical* to *Non-Critical*. Furthermore, theorems describing the effect of those functions (similar to (51)) are proved. These three clock functions and their corresponding theorems supersede progress properties one, two, and three. These three clock functions are then combined into a function which moves a token around the ring:

Definition 52: (Clock-Of-Token-Moving Index1 Index2 N Clock System-State) =

```
(i)  (If (And (Numberp Index1)
              (Lessp Index1 N)
              (Numberp Index2)
              (Lessp Index2 N))
(ii)     (If (Equal Index1 Index2)
             Clock
(iii)      (If (Process-Non-Critical Index1 System-State)
               (Clock-Of-Token-Moving
                 (Add1-Mod N Index1)
                 Index2
                 N
                (Clock-Of-Non-Critical-To-Non-Critical
                   Index1 Clock)
                 (Int (Make-Process-List
                        (System-Description N))
                      Clock
                      (Clock-Of-Non-Critical-To-Non-Critical
                        Index1 Clock)
                      System-State))
             (Clock-Of-Token-Moving
               (Add1-Mod N Index1)
               Index2
               N
               (Clock-Of-Wait-To-Non-Critical Index1 Clock)
               (Int (Make-Process-List
                        (System-Description N))
                      Clock
                      (Clock-Of-Wait-To-Non-Critical
                         Index1 Clock)
                      System-State))))
       Clock)
```

This function returns the time that the token, considered to be at the left channel of the *Index1*'th instance of *Me*, will be at the left channel of the *Index2*'th instance of *Me*. The system state is assumed to satisfy mutual exclusion. The *If* at (*i*) tests whether *Index1* and *Index2* are reasonable indices for this system. If they are, then (*ii*) tests whether the token is already at the left channel of *Index2*. If it is, then *Clock* is returned. If the token must move further along the ring, it is moved to the right channel and *Clock-Of-Token-Moving* is called recursively.

The token is moved to the right channel by (*iii*). If the *Index1*'th instance of *Me* is *Non-Critical*, then the clock necessary to move the token to the right channel is computed by *Clock-Of-Non-Critical-To-Non-Critical*. If that instance of *Me* is *Waiting*, then the clock is computed by *Clock-Of-Wait-To-Non-Critical*. Notice, that that instance of *Me* cannot be *Critical*, since, by assumption, the left channel is full, and mutual exclusion precludes the presence of another token in the ring.

Each recursive call of *Clock-Of-Token-Moving* increments *Index1* (modulo *N*) because the token has moved along the ring. *Clock* is changed to reflect the time that the token will be at the new location. *System-State* is changed to reflect the new system state. This is important: because many processes may execute during the interval that the token moves from the left to the right channel of the *Index1*'th instance of *Me*, the state of other processes may have changed. Therefore, *Clock-Of-Token-Moving* must keep track of the changing system state during the recursive calls.

The semantics of *Clock-Of-Token-Moving* are more succinctly described in the following theorem:

Theorem 53:

```
(Implies (And (Numberp Index1)
              (Lessp Index1 N)
              (Numberp Index2)
              (Lessp Index2 N)
              (Full-Wait Index1 Index2 N System-State))
         (Full-Wait Index2 Index2 N
                    (Int (Make-Process-List
                           (System-Description N))
                         Clock
                         (Clock-Of-Token-Moving
                           Index1
                           Index2
                           N
                           Clock
                           System-State)
                         System-State)))
```

The function *Full-Wait* tests whether its first argument is the index of a full channel; whether its second argument is the index of a *Waiting* process; and whether the system state satisfies mutual exclusion. This theorem states that if mutual exclusion holds, the *Index1*'th instance of *Me* has its left channel full, and the *Index2*'th instance of *Me* is *Waiting*, then in the new system state, obtained by *Int* from *Clock* to

Clock-Of-Token-Moving from *Index1* to *Index2*, the *Index2*'th instance of *Me* will still be *Waiting*, and its left channel will be full. Thus, *Clock-Of-Token-Moving* computes the time at which the token has moved all the way around the ring from a left channel to the left channel of the *Waiting* process.

The definition of *Clock-Of-Token-Moving* has system state as one of its arguments; *Clock-Of-Critical-To-Non-Critical* does not. This is because the latter does not need to consider the system state: the process in question must execute *Ticks+1* times, no matter what other processes do. However, *Clock-Of-Token-Moving* must dovetail the progress of the system computation with its computation of the final clock value because the computation of that clock value depends on branches that the system computation may take.

If the token is on the left channel of some *Index1*'th instance of *Me*, then using *Clock-Of-Token-Moving* it is a simple matter to run *Int* until the token is on the left channel of the *Waiting Index2*'th instance of *Me*; subsequently, that instance of *Me* is scheduled another time; at that point, that instance of *Me* will be *Critical* (by progress property four). This is the time that the function *Clock-Of-Token-Moving-To-Critical* (in the formal statement of the liveness property (45)) is supposed to compute.

However, the assumption that the token is at the left channel of some instance of *Me* need not be true: the token may have been absorbed by a *Critical* process. If the *Index1*'th instance of *Me* is *Critical*, then the time when the token is at the left channel of the *Waiting Index2*'th instance of *Me* is:

```
(Clock-Of-Token-Moving                                          (29)
  (Add1-Mod N Index1)
  Index2
  N
  (Clock-Of-Critical-To-Non-Critical
    Index1
    Clock
    (Get-State (Cons 'Me Index1)
               System-State))
  (Int (Make-Process-List (System-Description N))
       Clock
       (Clock-Of-Critical-To-Non-Critical
         Index1
         Clock
         (Get-State (Cons 'Me Index1)
                    System-State))
       System-State))
```

The token is released upon the right channel of the *Index1*'th instance of *Me*, using *Clock-Of-Critical-To-Non-Critical*, and then *Clock-Of-Token-Moving* is used to move the token until the left channel of *Index2*.

This idea is incorporated into the definition of *Clock-Of-Token-Moving-To-Wait*, which has the following arguments:

```
(Clock-Of-Token-Moving-To-Wait Index N Clock System-State)      (30)
```

Index is the index of the *Waiting* process. *Clock-Of-Token-Moving-To-Wait* searches the system for the token: if the token is in a left channel, it uses *Clock-Of-Token-Moving* directly to move the token; if, instead, a process is *Critical*, the formula in (29) is sufficient to move the token (where *Index2* is instantiated by *Index* and *Index1* is instantiated by the index of the *Critical* process).

Finally, *Clock-Of-Token-Moving-To-Critical* may be defined, using equation (30) and progress property four:

Definition 54: (Clock-Of-Token-Moving-To-Critical Index N Clock System-State) =

```
(If (And (Lessp Index N)
         (Numberp Index))
    (Add1 (Next-Clock (List 'Me Index)
                      (Clock-Of-Token-Moving-To-Wait
                        Index N Clock System-State)))
    Clock)
```

For any valid *Waiting* instance of *Me*, *Clock-Of-Token-Moving-To-Critical* returns the first time that it runs after the token is moved to its left channel. At that point, that instance of *Me* becomes *Critical*.

This definition satisfies the formal statement of liveness (45).

The mechanical proof of liveness required 144 definitions, theorems, and other events.

3.4.2 Discussion of the Proof of Liveness

Progress proofs have two parts:

- Show the initial predicate is stable except for the key transition.
- Show that the key transition changes the stable predicate to the desired one.

The progress proof rule and the theorem stating that *Choose-Fairly* returns every process but one (21), over the interval in question, is used to lift these two theorems to the progress property.

Many progress proofs may be necessary to provide the foundation for the desired liveness result. The progress statements must be sufficiently strong to imply the liveness property: for example, had these progress statements not carried along the *Waiting* status of the final process, the final statement would be able to move the token around the ring, but would not indicate that the *Waiting* process is still *Waiting*.

The liveness property is derived by the composition of various progress statements. Often, the functions that compute future clock values must keep track of the changing system state, since the system computation may branch.

3.5 Summary

The proofs of mutual exclusion and liveness presented here are similar to hand proofs that one would have done. The key differences are in the level of detail, which is necessarily greater for a machine verified proof, and in the definitions of the clock functions. Since the Boyer-Moore logic does not define quantifiers, the future system state that satisfies the liveness property must be explicitly derived rather than simply shown to exist. This system state is derived from *Int* using a computed clock that corresponds to the necessary sequence of transitions. These transitions correspond to the effective transitions that one must demonstrate when proving liveness properties in temporal logic.

Chapter 4
CONCLUSION

4.1 Comparison to Related Work

The transition system model of distributed systems has been used widely in the literature. Our theory is most closely related to [4] [5] [6] and to *Unity* [3]. The proof rules which we mechanically derive from our operational semantics are strongly influenced by those taken as axioms in *Unity*.

Our theory differs from previous work because our proofs are mechanically verified. Since the Boyer-Moore Logic does not define quantification, a proof in our theory that a property **Q** leads to a property **R** must explicitly construct, by a composition of *Next-Clock*'s, the time at which **R** holds. Previous transition system work has been based on temporal logic and allows one to conclude that **R** will hold at some unknown future time. However, this abstraction does not simplify verification, since proofs of liveness properties in temporal logic must demonstrate a sequence of effective transitions [5] much like our composition of *Next-Clock*'s.

Lengauer's work on concurrency is also mechanized on the Boyer-Moore prover. Given architectural assumptions, it develops provably correct transformations that generate the optimum concurrent execution from a sequential program [7]. It differs in approach from our work because it concentrates on proving that the transformations are correct rather than proving properties directly about distributed systems.

4.2 Future Work

Our future research will be in three areas:

- Liveness proofs in our theory explicitly construct a composition of *Next-Clock*'s that specify when the desired property occurs, rather than simply that it occurs. Thus, if one could place time bounds on real machines' scheduling strategies, one might be able to derive bounds on an algorithm's efficiency from its correctness proof.
- Distributed systems should be designed hierarchically. This strategy is aided when their (mechanical) proofs are reusable. This reusability would allow, for example, the incorporation of the proved solution to the mutual exclusion problem into a larger system.

- We are also investigating the application of our model to real time systems. Dropping equation (3) from the definition of *Next-Clock* would allow multiple processes to be scheduled simultaneously. Some assumptions must be made about the architecture when assigning semantics to simultaneous execution. The new operational semantics should be designed so it yields useful proof rules.

4.3 Summary

Our theory has demonstrated that mechanical proofs of distributed systems are possible. Furthermore, the axiomatization of *Next-Clock* is interesting in its own right as a characterization of fairness. Finally, the operational semantics defined in *Int* is conceptually simple, yet generates proof rules which may be used to prove safety and liveness properties.

4.4 Acknowledgments

We would like to thank the many people who helped us during the course of this research and during the writing of this paper. They include: Ken Calvert, Ernie Cohen, E. Allen Emerson, C.A.R. Hoare, Matt Kaufmann, Chris Lengauer, J S. Moore, Randy Pollack, Mark Staskauskas, Tom Verhoeff, Michael Vose, and Matt Wilding. We also thank the Institute for Computer Sciences and Computer Applications and its staff, at the University of Texas at Austin, for providing an excellent working environment.

Appendix A
Expressing Embedded Quantification

,

In certain cases, it is possible to express embedded quantification in the Boyer-Moore Logic. Embedded quantification can be expressed if one can construct a Boyer-Moore function which enumerates the range of the quantification. In the case of our proof rules, Matt Kaufmann found a clever way to express the embedded quantification of the invariance proof rule (28).

Our invariance proof rule contains within it the function *Invariance-Over-Single-Step* which has the general form:

$$(\forall\ \mathtt{Time:\ Clock} \leq \mathtt{Time} < \mathtt{Max\text{-}Clock:\ P(Clock,\ Time)}) \tag{31}$$

Notice that **P** is an abbreviation for the more complex implication in the definition of Invariance-Over-Single-Step which involves the additional variables *Process-List* and *System-State*. These variables are hidden since they play no role in the embedded quantification.

One could express (31) in the logic using a function which recursed over the interval *Clock* to *Max-Clock* and returned true if **P** was true at every point. However, proving facts about it would require an induction.

A more clever solution is to observe that by predicate calculus (31) is equal to:

$$\neg\ (\exists\ \mathtt{Time:\ Clock} \leq \mathtt{Time} < \mathtt{Max\text{-}Clock:}\ \neg\ \mathtt{P(Clock,\ Time)}) \tag{32}$$

One can then introduce a skolem function, *Sk*, which recurses through the interval *Clock* to *Max-Clock* and returns the *Time*, if any, at which $\neg$ *P(Clock, Time)* is true.[4]One must insure that if there is no time in the interval such that $\neg$ *P(Clock*, *Time)*, then **P** of the value returned by *Sk* is true.> Then (32) is equal to:

$$\neg\ (\neg\ \mathtt{P(Clock,\ Sk(Clock,\ Max\text{-}Clock))}) \tag{33}$$

This is (of course):

$$\mathtt{P(Clock,\ Sk(Clock,\ Max\text{-}Clock))} \tag{34}$$

4

The form (34) is convenient since if one can prove that **P** is true for any value of its second argument, then one can prove that (34) is true without looking at the internals of *Sk*. We actually express the antecedent of the invariance proof rule (8) in the form of (34) and use the proof rule by essentially showing that **P** is true for any value of its second argument (the actual invariance proof rule contains various complicating details-see (8)).

Appendix B
Utilities

This file contains lemmas and definitions that we find useful in the rest of the proof.

Several Useful Induction Schemes.

Three hypothesis: all possible permutations of at most one CDR on each of two arguments.

Definition 1: (CDR-ON-BOTH X Y) =

```
(IF (LISTP X)
    (IF (LISTP Y)
        (AND (CDR-ON-BOTH (CDR X) Y)
             (CDR-ON-BOTH X (CDR Y))
             (CDR-ON-BOTH (CDR X) (CDR Y)))
        T)
    T)

((LESSP (PLUS (COUNT X) (COUNT Y))))
```

The standard Integer induction.

Definition 2: (INTEGER-INDUCTION-SCHEME-1 N) =

```
(IF (ZEROP N)
    T
    (INTEGER-INDUCTION-SCHEME-1 (SUB1 N)))
```

Downward induction from M.

Definition 3: (INTEGER-INDUCTION-SCHEME-2 N M) =

```
(IF (LESSP N M)
    (INTEGER-INDUCTION-SCHEME-2 (ADD1 N) M)
    T)

((LESSP (DIFFERENCE M N)))
```

Three hypothesis: all possible permutations of at most one SUB1 on each of two arguments.

Definition 4: (SUB1-ON-BOTH N M) =

```
(IF (ZEROP N)
    T
    (IF (ZEROP M)
        T
        (AND (SUB1-ON-BOTH (SUB1 N) M)
             (SUB1-ON-BOTH N (SUB1 M))
```

```
                    (SUB1-ON-BOTH (SUB1 N) (SUB1 M)))))
    ((LESSP (PLUS N M)))
```

Returns the Length of the List. Length is not defined in NQTHM.

Definition 5: (LENGTH L) =

```
(IF (LISTP L)
    (ADD1 (LENGTH (CDR L)))
    0)
```

Is the final CDR of the list NIL?

Definition 6: (PROPER-LISTP L) =

```
(IF (LISTP L)
    (PROPER-LISTP (CDR L))
    (EQUAL L NIL))
```

Returns T if SET contains no duplicate elements.

Definition 7: (SETP SET) =

```
(IF (LISTP SET)
    (IF (MEMBER (CAR SET) (CDR SET))
        F
        (SETP (CDR SET)))
    T)
```

Returns T if the intersection of X and Y is empty.

Definition 8: (DISJOINTP X Y) =

```
(IF (LISTP X)
    (AND (NOT (MEMBER (CAR X) Y))
         (DISJOINTP (CDR X) Y))
    T)
```

This is a useful lemma, simplifying the term.

Theorem 9: ABOUT-LENGTH-AND-APPEND (REWRITE)

```
(EQUAL (LENGTH (APPEND X Y))
       (PLUS (LENGTH X)
             (LENGTH Y)))
```

This forces case analysis.

Theorem 10: ABOUT-MEMBER-APPEND (REWRITE)

```
(EQUAL (MEMBER X (APPEND Y Z))
       (OR (MEMBER X Y)
           (MEMBER X Z)))
```

This forces case analysis.

Theorem 11: ABOUT-DISJOINTP-APPEND (REWRITE)

```
(EQUAL (DISJOINTP X (APPEND Y Z))
       (AND (DISJOINTP X Y)
            (DISJOINTP X Z)))
```

This forces case analysis.

Theorem 12: SETP-UNION-DISJOINT-SETS (REWRITE)

```
(EQUAL (SETP (APPEND X Y))
       (AND (DISJOINTP X Y)
            (SETP X)
            (SETP Y)))
```

Disjoint is commutative. The Induction scheme is interesting.

Theorem 13: DISJOINTP-IS-COMMUTATIVE (REWRITE)

```
(EQUAL (DISJOINTP X Y)
       (DISJOINTP Y X))

((INDUCT (CDR-ON-BOTH X Y)))
```

This is a terrible rewrite rule. It is used only when the proof rules are invoked.

Theorem 14: EQUAL-LISTS-OF-LENGTH-TWO (REWRITE)

```
(IMPLIES (AND (EQUAL (LENGTH L1) 2)
              (EQUAL (LENGTH L2) 2))
         (EQUAL (EQUAL L1 L2)
                (AND (EQUAL (CAR L1) (CAR L2))
                     (EQUAL (CADR L1) (CADR L2))
                     (EQUAL (CDDR L1) (CDDR L2)))))

(DISABLE EQUAL-LISTS-OF-LENGTH-TWO)                              (1)
```

Replace the previous value associated with Target-Name with New-Value. If Target-Name is not found in Assoc-List, then add the pair at the end of the list.

Definition 15: (UPDATE-ASSOC TARGET-NAME NEW-VALUE ASSOC-LIST) =

```
(IF (LISTP ASSOC-LIST)
    (IF (EQUAL (CAAR ASSOC-LIST)
               TARGET-NAME)
        (CONS (CONS TARGET-NAME NEW-VALUE)
              (CDR ASSOC-LIST))
        (CONS (CAR ASSOC-LIST)
              (UPDATE-ASSOC TARGET-NAME
                            NEW-VALUE
                            (CDR ASSOC-LIST))))
    (LIST (CONS TARGET-NAME NEW-VALUE)))
```

This lemma simplifies multiple Assoc and Update-Assoc combinations. Its current form is due to Matt Kaufmann.

Theorem 16: SIMPLIFY-ASSOC (REWRITE)

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

Defines the Add1 Modulo function. The ring size is first because it reads better.

Definition 17: (ADD1-MOD N X) =

```
(IF (LESSP (ADD1 X) N)
    (ADD1 X)
    0)
```

Defines the Sub1 Modulo function. The ring size is first because it reads better.

Definition 18: (SUB1-MOD N X) =

```
(IF (OR (ZEROP X)
        (LESSP N X))
    (SUB1 N)
    (SUB1 X))
```

The following lemmas are only used if Add1-Mod and Sub1-Mod are disabled. They are useful in a few proofs, where the enabled Add1-Mod and Sub1-Mod cause a great deal of case analysis. These lemmas do not attempt to replace the definitions of Add1-Mod and Sub1-Mod.

Theorem 19: ADD1-MOD-SUB1-MOD-SIMP (REWRITE)

```
(EQUAL (ADD1-MOD N (SUB1-MOD N I))
       (IF (LESSP I N)
           (FIX I)
           0))
```

Theorem 20: SUB1-MOD-ADD1-MOD-SIMP (REWRITE)

```
(EQUAL (SUB1-MOD N (ADD1-MOD N I))
       (IF (LESSP I N)
           (FIX I)
           (SUB1 N)))
```

Theorem 21: ADD1-MOD-LESSP (REWRITE)

```
(EQUAL (LESSP (ADD1-MOD N I) N)
       (LESSP 0 N))
```

Theorem 22: SUB1-MOD-LESSP (REWRITE)

```
(EQUAL (LESSP (SUB1-MOD N I) N)
       (LESSP 0 N))
```

The next two lemmas are not as strong as they could be, but they are strong enough for our purposes. Stronger ones would cause the case analysis these were designed to prevent.

Theorem 23: ADD1-MOD-EQUAL (REWRITE)

```
(IMPLIES (LESSP 1 N)
         (NOT (EQUAL (ADD1-MOD N I) I)))
```

Theorem 24: SUB1-MOD-EQUAL (REWRITE)

```
(IMPLIES (LESSP 1 N)
         (NOT (EQUAL (SUB1-MOD N I) I)))
```

Appendix C
The Scheduler

This files defines the Fair Scheduler. It is composed of two functions: Next-Clock and Choose-Fairly.

Next-Clock functions correspond to Fair Traces. We define the Next-Clock functions now, and the correspondece, subsequently.

Declare Next-Clock to be a total function of two arguments.

Definition 1:

```
(DCL NEXT-CLOCK (PROCESS-DESCRIPTION CLOCK))
```

Theorem 2:

```
(ADD-AXIOM NEXT-CLOCK-IS-SUBRP (REWRITE)
           (SUBRP 'NEXT-CLOCK))
```

These two lines ought to be consequences of the previous Add-Axiom event.

```
(PUT1 'NEXT-CLOCK *1*T 'SUBRP)                                    (1)
(PUT1 'NEXT-CLOCK '((NEXT-CLOCK . T)) 'TOTALP-LST)
```

The following five axioms define the set of functions that Next-Clock represents.

Next-Clock's first argument is any element of the Boyer-Moore universe its second is a number (or coerced to a number).

Axiom 1: Time makes no sense if it is not ordered. We choose the natural numbers.

Theorem 3:

```
(ADD-AXIOM NUMBERP-NEXT-CLOCK (REWRITE)
           (NUMBERP (NEXT-CLOCK PROCESS-DESCRIPTION
                                CLOCK)))
```

Axiom 2: For convenience, we fix the second argument.

Theorem 4:

```
(ADD-AXIOM NEXT-CLOCK-FIXES-CLOCK (REWRITE)
           (IMPLIES (NOT (NUMBERP CLOCK))
                    (EQUAL (NEXT-CLOCK PROCESS-DESCRIPTION
```

```
                                         CLOCK)
                            (NEXT-CLOCK PROCESS-DESCRIPTION
                                        0))))
```

Axiom 3: Next-Clock always returns a clock value that is at least the current clock value.

Theorem 5:

```
(ADD-AXIOM NEXT-CLOCK-AT-LEAST-CLOCK (REWRITE)
           (NOT (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION
                                   CLOCK)
                       CLOCK)))
```

Axiom 4: Next-Clock never maps two processes to the same clock value.

Theorem 6:

```
(ADD-AXIOM NEXT-CLOCK-IS-ONE-TO-ONE-BASIC ()
           (IMPLIES (EQUAL (NEXT-CLOCK PROCESS-DESCRIPTION-1
                                       CLOCK)
                           (NEXT-CLOCK PROCESS-DESCRIPTION-2
                                       CLOCK))
                    (EQUAL PROCESS-DESCRIPTION-1
                           PROCESS-DESCRIPTION-2)))
```

Axiom 5: The clock argument may be incremented without changing Next-Clock’s value, provided it is not incremented past (Next-Clock Process-Description Clock).

Theorem 7:

```
(ADD-AXIOM NEXT-CLOCK-IS-HONEST-BASIC ()
           (IMPLIES (LESSP CLOCK
                           (NEXT-CLOCK PROCESS-DESCRIPTION
                                       CLOCK))
                    (EQUAL (NEXT-CLOCK PROCESS-DESCRIPTION
                                       CLOCK)
                           (NEXT-CLOCK PROCESS-DESCRIPTION
                                       (ADD1 CLOCK)))))
```

Derived Properties of Next-Clock.

Indution for the interval [Clock, (Next-Clock Process-Description Clock)].

Definition 8: (NEXT-CLOCK-IS-HONEST-INDUCTION-HINT PROCESS-DESCRIPTION CLOCK TIME) =

```
(IF (OR (NOT (LESSP CLOCK TIME))
        (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION
                           CLOCK)
               TIME))
```

```
0
(NEXT-CLOCK-IS-HONEST-INDUCTION-HINT
  PROCESS-DESCRIPTION CLOCK (SUB1 TIME)))
```

This is an extension of Axiom 5. Not only can Clock be incremented within the interval, any Time in the interval can be chosen.

Theorem 9: NEXT-CLOCK-IS-HONEST ()

```
(IMPLIES (AND (NOT (LESSP TIME CLOCK))
              (NOT (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION
                                      CLOCK)
                          TIME)))
         (EQUAL (EQUAL (NEXT-CLOCK PROCESS-DESCRIPTION
                                   CLOCK)
                       (NEXT-CLOCK PROCESS-DESCRIPTION
                                   TIME))
                T))

((USE (NEXT-CLOCK-IS-HONEST-BASIC (CLOCK (SUB1 TIME))))
 (INDUCT (NEXT-CLOCK-IS-HONEST-INDUCTION-HINT
           PROCESS-DESCRIPTION CLOCK TIME)))
```

Next-Clock is Idempotent in its second argument.

Theorem 10: NEXT-CLOCK-IS-IDEMPOTENT (REWRITE)

```
(EQUAL (NEXT-CLOCK PROCESS-DESCRIPTION
                   (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK))
       (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK))

((USE (NEXT-CLOCK-IS-HONEST
        (CLOCK CLOCK)
        (TIME (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK)))))
```

This is an ugly statement that given two distinct process descriptions, and any two clock values, Next-Clock will never map the processes to the same time.

Theorem 11: NEXT-CLOCK-IS-ONE-TO-ONE (REWRITE)

```
(EQUAL (EQUAL (NEXT-CLOCK PROCESS-DESCRIPTION-1
                          CLOCK-1)
              (NEXT-CLOCK PROCESS-DESCRIPTION-2
                          CLOCK-2))
       (AND (EQUAL PROCESS-DESCRIPTION-1
                   PROCESS-DESCRIPTION-2)
            (NOT (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION-1
                                    CLOCK-1)
                        CLOCK-2))
            (NOT (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION-2
                                    CLOCK-2)
                        CLOCK-1))))
```

```
((USE (NEXT-CLOCK-IS-HONEST
        (PROCESS-DESCRIPTION PROCESS-DESCRIPTION-1)
        (CLOCK CLOCK-1)
        (TIME CLOCK-2))
      (NEXT-CLOCK-IS-HONEST
        (PROCESS-DESCRIPTION PROCESS-DESCRIPTION-2)
        (CLOCK CLOCK-2) (TIME CLOCK-1))
      (NEXT-CLOCK-IS-ONE-TO-ONE-BASIC (CLOCK CLOCK-1))
      (NEXT-CLOCK-IS-ONE-TO-ONE-BASIC (CLOCK CLOCK-2))))
```

The next two lemmas are necessary to prove the third.

Theorem 12: NEXT-CLOCK-LESSPS-1 ()

```
(IMPLIES (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK-1)
                CLOCK-2)
         (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION
                            CLOCK-1)
                (NEXT-CLOCK PROCESS-DESCRIPTION
                            CLOCK-2)))
```

Theorem 13: NEXT-CLOCK-LESSPS-2 ()

```
(IMPLIES (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK-1)
                (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK-2))
         (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK-1)
                CLOCK-2))

((DISABLE NEXT-CLOCK-IS-ONE-TO-ONE)
 (USE (NEXT-CLOCK-IS-HONEST (CLOCK CLOCK-1)
                            (TIME CLOCK-2))
      (NEXT-CLOCK-IS-HONEST (CLOCK CLOCK-2)
                            (TIME CLOCK-1))))
```

Identify distinct intervals of [Clock, (Next-Clock Process-Description Clock)].

Theorem 14: NEXT-CLOCK-LESSPS (REWRITE)

```
(EQUAL (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK-1)
              (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK-2))
       (LESSP (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK-1)
              CLOCK-2))

((USE (NEXT-CLOCK-LESSPS-1)
      (NEXT-CLOCK-LESSPS-2)))
```

Choose-Fairly defines the correspondence between Next-Clock and a fair trace. Given a set of processes and a clock, it returns the one that is scheduled at that clock.

Choose-Fairly returns a list of the processes (contained in Process-List) that are scheduled at Clock. From the definition of Next-Clock, we know that this list contains at most one distinct process. However, that is a property of Next-Clock, and should not be enforced by Choose-Fairly. For convenience (just like in Next-Clock), Choose-Fairly fixes the Clock argument.

Definition 15: (CHOOSE-FAIRLY PROCESS-LIST CLOCK) =

```
(IF (LISTP PROCESS-LIST)
    (IF (EQUAL (FIX CLOCK) (NEXT-CLOCK (CAR PROCESS-LIST)
                                       CLOCK))
        (CONS (CAR PROCESS-LIST)
              (CHOOSE-FAIRLY (CDR PROCESS-LIST) CLOCK))
        (CHOOSE-FAIRLY (CDR PROCESS-LIST) CLOCK))
    NIL)
```

Theorem 16: CHOOSE-FAIRLY-FIXES-CLOCK (REWRITE)

```
(IMPLIES (NOT (NUMBERP CLOCK))
         (EQUAL (CHOOSE-FAIRLY PROCESS-LIST CLOCK)
                (CHOOSE-FAIRLY PROCESS-LIST 0)))
```

Choose-Fairly will not invent a process.

Theorem 17: CHOOSE-FAIRLY-RETURNS-VALID-PROCESS-DESCRIPTION (REWRITE)

```
(OR (EQUAL (CHOOSE-FAIRLY PROCESS-LIST CLOCK)
           NIL)
    (MEMBER (CAR (CHOOSE-FAIRLY PROCESS-LIST CLOCK))
            PROCESS-LIST))
```

This is the important property of Choose-Fairly. Choose-Fairly correctly identifies the process scheduled at Clock. We are only concerned about the case when Process-List is a set.

Theorem 18: CHOOSE-FAIRLY-RETURNS-CORRECT-PROCESS-DESCRIPTION (REWRITE)

```
(IMPLIES (SETP PROCESS-LIST)
         (EQUAL (CHOOSE-FAIRLY
                  PROCESS-LIST
                  (NEXT-CLOCK PROCESS-DESCRIPTION CLOCK))
                (IF (MEMBER PROCESS-DESCRIPTION
                            PROCESS-LIST)
                    (LIST PROCESS-DESCRIPTION)
                    NIL)))
```

In the interval [Clock, (Next-Clock Process-Description Clock)), Next-Clock never schedules Process-Description, so Choose-Fairly will never find it.

Theorem 19: CHOOSE-FAIRLY-NEVER-RETURNS-PROCESS-IN-INTERVAL (REWRITE)

```
(IMPLIES (AND (NOT (LESSP TIME CLOCK))
              (LESSP TIME (NEXT-CLOCK PROCESS-DESCRIPTION
                                      CLOCK)))
         (EQUAL (EQUAL (CHOOSE-FAIRLY PROCESS-LIST TIME)
```

```
                              (LIST PROCESS-DESCRIPTION))
                       F))

       ((DISABLE NEXT-CLOCK-IS-ONE-TO-ONE)
        (USE (NEXT-CLOCK-IS-HONEST)))
```

The next two lemmas are used in conjuction with the proof rules. The are used to instruct the prover which processes in the process list may be scheduled during an interval of a particular form. The naming scheme for these lemmas is: Choose-Fairly-Returns-Member-0 says that every process (all but 0) may be scheduled in the interval. This is useful for open intervals. Choose-Fairly-Returns-Member-1 says that every process but one (subtract one) is scheduled in the interval. This is useful when the right end of the interval is open and is of the form (Next-Clock Process-Description Clock). Process-Description is not scheduled in that open interval.

More complicated interval may be discussed. Certain ones may prevent two processes from being scheduled. Depending on the progress proof, one may need to prove additional lemmas here. We have found these to be sufficient in our example.

This lemma is used in conjunction with the Invariance Proof rule to prove Invariance properties. It says that for an arbitrary Clock, Choose-Fairly schedules nothing, or schedules a member of the Process-List. This is suitable for Invariance Properties, since any process may be scheduled.

Theorem 20: CHOOSE-FAIRLY-RETURNS-MEMBER-0 ()

```
(IMPLIES (SETP PROCESS-LIST)
         (OR (EQUAL (CHOOSE-FAIRLY PROCESS-LIST CLOCK)
                    NIL)
             (MEMBER (CAR (CHOOSE-FAIRLY PROCESS-LIST
                                         CLOCK))
                     PROCESS-LIST)))
```

This lemma is used in conjunction with the Progress Proof rule to prove progress properties. It says that over the interval [Clock, (Next-Clock Process-Description Clock)) every process but Process-Description may be scheduled. This is appropriate for the Stable interval in a progress proof, since the Key process is not scheduled during that interval.

Theorem 21: CHOOSE-FAIRLY-RETURNS-MEMBER-1 ()

```
(IMPLIES (AND (SETP PROCESS-LIST)
              (NOT (LESSP TIME CLOCK))
              (LESSP TIME (NEXT-CLOCK PROCESS-DESCRIPTION
                                      CLOCK)))
         (OR (EQUAL (CHOOSE-FAIRLY PROCESS-LIST TIME)
```

```
         NIL)
  (AND (MEMBER (CAR (CHOOSE-FAIRLY PROCESS-LIST
                                   TIME))
               PROCESS-LIST)
       (NOT (EQUAL (CAR (CHOOSE-FAIRLY
                             PROCESS-LIST TIME))
                   PROCESS-DESCRIPTION)))))
```

Appendix D
The Process List

This file defines our scheme for defining systems. This included the System Description and the Process List.

The family size of a family in the process list is the number of occurences of instances of that family in the process list.

Definition 1: (FAMILY-SIZE FAMILY-NAME PROCESS-LIST) =

```
(IF (LISTP PROCESS-LIST)
    (IF (EQUAL FAMILY-NAME
               (CAAR PROCESS-LIST))
        (ADD1 (FAMILY-SIZE FAMILY-NAME
                           (CDR PROCESS-LIST)))
        (FAMILY-SIZE FAMILY-NAME (CDR PROCESS-LIST)))
    0)
```

Returns a family (a list of instances of that family). The top-most call is of the form: (Make-Family family-name family-size)

Definition 2: (MAKE-FAMILY FAMILY-NAME CURRENT-INDEX) =

```
(IF (ZEROP CURRENT-INDEX)
    NIL
    (CONS (LIST FAMILY-NAME (SUB1 CURRENT-INDEX))
          (MAKE-FAMILY FAMILY-NAME
                       (SUB1 CURRENT-INDEX))))
```

Builds the Process List corresponding to the System Description.

Definition 3: (MAKE-PROCESS-LIST SYSTEM-DESCRIPTION) =

```
(IF (LISTP SYSTEM-DESCRIPTION)
    (APPEND (MAKE-FAMILY (CAAR SYSTEM-DESCRIPTION)
                         (CADAR SYSTEM-DESCRIPTION))
            (MAKE-PROCESS-LIST (CDR SYSTEM-DESCRIPTION)))
    NIL)
```

An abbreviation for determining the size of a family from the system description. Each element of the system description is (List Family-Name Family-Size).

Definition 4: (DECLARED-FAMILY-SIZE FAMILY-NAME SYSTEM-DESCRIPTION) =

```
(FIX (CADR (ASSOC FAMILY-NAME SYSTEM-DESCRIPTION)))
```

Returns a list of all the family names in the system description.

Definition 5: (FAMILY-NAMES SYSTEM-DESCRIPTION) =

```
(IF (LISTP SYSTEM-DESCRIPTION)
    (CONS (CAAR SYSTEM-DESCRIPTION)
          (FAMILY-NAMES (CDR SYSTEM-DESCRIPTION)))
    NIL)
```

Simplify family-size of append. This is very much like About-Length-And-Append.

Theorem 6: ABOUT-APPEND-AND-FAMILY-SIZE (REWRITE)

```
(EQUAL (FAMILY-SIZE FAMILY-NAME (APPEND L1 L2))
       (PLUS (FAMILY-SIZE FAMILY-NAME L1)
             (FAMILY-SIZE FAMILY-NAME L2)))
```

Simplify Family-Size of Make-Family.

Theorem 7: FAMILY-SIZE-OF-MAKE-FAMILY (REWRITE)

```
(EQUAL (FAMILY-SIZE FAMILY-NAME1
                    (MAKE-FAMILY FAMILY-NAME2 FAMILY-SIZE))
       (IF (EQUAL FAMILY-NAME1 FAMILY-NAME2)
           (FIX FAMILY-SIZE)
           0))
```

This induction scheme is only needed once. The prover will not choose the correct one.

Definition 8: (MAKE-PROCESS-LIST-RETURNS-CORRECT-FAMILY-SIZE-INDUCTION-SCHEME SYSTEM-DESCRIPTION FAMILY-NAME) =

```
(IF (LISTP SYSTEM-DESCRIPTION)
    (IF (EQUAL FAMILY-NAME (CAAR SYSTEM-DESCRIPTION))
        T
        (MAKE-PROCESS-LIST-RETURNS-CORRECT-
FAMILY-SIZE-INDUCTION-SCHEME
          (CDR SYSTEM-DESCRIPTION) FAMILY-NAME))
    T)
```

If the system description is properly constructed (no duplicate families) then the family size generated by Make-Process-List is correct.

Theorem 9: MAKE-PROCESS-LIST-RETURNS-CORRECT-FAMILY-SIZE (REWRITE)

```
(IMPLIES (SETP (FAMILY-NAMES SYSTEM-DESCRIPTION))
         (EQUAL (FAMILY-SIZE FAMILY-NAME
                             (MAKE-PROCESS-LIST
                               SYSTEM-DESCRIPTION))
                (DECLARED-FAMILY-SIZE FAMILY-NAME
                                      SYSTEM-DESCRIPTION)))

((INDUCT (MAKE-PROCESS-LIST-RETURNS-CORRECT-
FAMILY-SIZE-INDUCTION-SCHEME
```

```
    SYSTEM-DESCRIPTION FAMILY-NAME)))
```

Define the necessary and sufficient conditions for membership in a family.

Theorem 10: CORRECT-PROCESS-DESCRIPTION-MEMBER-MAKE-FAMILY (REWRITE)

```
(IFF (MEMBER PROCESS-DESCRIPTION
             (MAKE-FAMILY FAMILY-NAME FAMILY-SIZE))
     (AND (NUMBERP (CADR PROCESS-DESCRIPTION))
          (LESSP (CADR PROCESS-DESCRIPTION)
                 FAMILY-SIZE)
          (EQUAL (CAR PROCESS-DESCRIPTION)
                 FAMILY-NAME)
          (EQUAL (LENGTH PROCESS-DESCRIPTION)
                 2)
          (EQUAL (CDDR PROCESS-DESCRIPTION)
                 NIL)))
```

Defines the necessary and sufficient conditions for membership in the process list. This lemma is very important. It allows the translation from membership in a process list to distinct process names and indices. This allows the mechanization of the proof rules. The hypothesis of Setp is essential: otherwise Declared-Family-Size might not work properly. In any case, all System-Descriptions should satisfy the hypothesis. IFF is not suitable in place of Equal! (I don't know why.)

Theorem 11: CORRECT-PROCESS-DESCRIPTION-MEMBER-MAKE-PROCESS-LIST (REWRITE)

```
(IMPLIES (SETP (FAMILY-NAMES SYSTEM-DESCRIPTION))
         (EQUAL (MEMBER PROCESS-DESCRIPTION
                        (MAKE-PROCESS-LIST
                          SYSTEM-DESCRIPTION))
                (AND (NUMBERP (CADR PROCESS-DESCRIPTION))
                     (LESSP (CADR PROCESS-DESCRIPTION)
                            (DECLARED-FAMILY-SIZE
                              (CAR PROCESS-DESCRIPTION)
                             SYSTEM-DESCRIPTION))
                     (MEMBER (CAR PROCESS-DESCRIPTION)
                             (FAMILY-NAMES
                               SYSTEM-DESCRIPTION))
                     (EQUAL (LENGTH PROCESS-DESCRIPTION) 2)
                     (EQUAL (CDDR  PROCESS-DESCRIPTION)
                            NIL))))
```

Many of our lemmas require that the process list be a set. (See the Choose-Fairly lemma.) The third lemma here says that can be determined from the system description.

Theorem 12: DISJOINT-FAMILIES (REWRITE)

```
(EQUAL (DISJOINTP (MAKE-FAMILY FAMILY-NAME-1 FAMILY-SIZE-1)
                  (MAKE-FAMILY FAMILY-NAME-2 FAMILY-SIZE-2))
```

```
        (OR (NOT (EQUAL FAMILY-NAME-1 FAMILY-NAME-2))
            (ZEROP FAMILY-SIZE-1)
            (ZEROP FAMILY-SIZE-2)))

((INDUCT (SUB1-ON-BOTH FAMILY-SIZE-1 FAMILY-SIZE-2)))
```

Theorem 13: DISJOINT-FAMILY-NAMES (REWRITE)

```
(IMPLIES (AND (SETP (FAMILY-NAMES SYSTEM-DESCRIPTION))
              (NOT (MEMBER FAMILY-NAME
                           (FAMILY-NAMES
                             SYSTEM-DESCRIPTION)))))
         (DISJOINTP (MAKE-FAMILY FAMILY-NAME FAMILY-SIZE)
                    (MAKE-PROCESS-LIST
                      SYSTEM-DESCRIPTION)))
```

This cannot be made into an equality without much trouble. (One would have to check family sizes.)

Theorem 14: SETP-MAKE-PROCESS-LIST (REWRITE)

```
(IMPLIES (SETP (FAMILY-NAMES SYSTEM-DESCRIPTION))
         (SETP (MAKE-PROCESS-LIST SYSTEM-DESCRIPTION)))
```

Finally, the following four functions are disabled. For every system, only three sets of lemmas need to be proved about the system description. See the example in the definition of the system containing N instances of Me.

```
(DISABLE DECLARED-FAMILY-SIZE)                                  (1)
(DISABLE FAMILY-SIZE)
(DISABLE FAMILY-NAMES)
(DISABLE MAKE-PROCESS-LIST)
```

Appendix E
The Interpreter

This file defines an Operational Semantics for Distributed Systems.

Single-Step takes a list containing the description of the process to be run and returns the resultant state. The CAAR of Process-Description is the family name of the process and the CADAR is that process's index. Single-Step fixes the Clock for convenience (like Next-Clock and Choose-Fairly) otherwise, every lemma would need a precondition of (Numberp Clock).

Definition 1: (SINGLE-STEP PROCESS-DESCRIPTION PROCESS-LIST CLOCK SYSTEM-STATE) =

```
(IF (LISTP PROCESS-DESCRIPTION)
    (APPLY$ (CAAR PROCESS-DESCRIPTION)
            (LIST (CADAR PROCESS-DESCRIPTION)
                  (FAMILY-SIZE (CAAR PROCESS-DESCRIPTION)
                               PROCESS-LIST)
                  (FIX CLOCK)
                  SYSTEM-STATE))
    SYSTEM-STATE)
```

Update System-State from Clock to Max-Clock. Int is a mnemonic for Interpreter.

Definition 2: (INT PROCESS-LIST CLOCK MAX-CLOCK SYSTEM-STATE) =

```
(IF (LESSP CLOCK MAX-CLOCK)
    (INT PROCESS-LIST
         (ADD1 CLOCK)
         MAX-CLOCK
         (SINGLE-STEP (CHOOSE-FAIRLY PROCESS-LIST
                                     CLOCK)
                      PROCESS-LIST
                      CLOCK
                      SYSTEM-STATE))
    SYSTEM-STATE)

((LESSP (DIFFERENCE MAX-CLOCK CLOCK)))
```

This lemma is useful when Single-Step is disabled (for example when proving the proof rules).

Theorem 3: SINGLE-STEP-FIXES-CLOCK (REWRITE)

```
(IMPLIES (NOT (NUMBERP CLOCK))
         (EQUAL (SINGLE-STEP PROCESS-DESCRIPTION
                             PROCESS-LIST
                             CLOCK
                             SYSTEM-STATE)
                (SINGLE-STEP PROCESS-DESCRIPTION
```

```
                              PROCESS-LIST
                              0
                              SYSTEM-STATE)))

(DISABLE SINGLE-STEP-FIXES-CLOCK)                                          (1)
```

Appendix F
The Proof Rules

This file contains the Proof Rules, derived from Int, that we use when proving properties of distributed systems.

Int may be composed.

Theorem 1: COMPOSITIONALITY-OF-INT (REWRITE)

```
(IMPLIES (AND (NOT (LESSP TIME CLOCK))
              (NOT (LESSP MAX-CLOCK TIME)))
         (EQUAL (INT PROCESS-LIST
                     TIME
                     MAX-CLOCK
                     (INT PROCESS-LIST
                          CLOCK
                          TIME
                          SYSTEM-STATE))
                (INT PROCESS-LIST
                     CLOCK
                     MAX-CLOCK
                     SYSTEM-STATE)))

((DISABLE SINGLE-STEP)
 (ENABLE SINGLE-STEP-FIXES-CLOCK))
```

Int is ineffectual on empty intervals.

Theorem 2: SIMPLIFY-INT (REWRITE)

```
(IMPLIES (NOT (LESSP CLOCK MAX-CLOCK))
         (EQUAL (INT PROCESS-LIST
                     CLOCK
                     MAX-CLOCK
                     SYSTEM-STATE)
                SYSTEM-STATE))
```

Single-Step of Int really means run Int one step further.

Theorem 3: SIMPLIFY-SINGLE-STEP-OF-INT (REWRITE)

```
(IMPLIES (LESSP CLOCK MAX-CLOCK)
         (EQUAL (SINGLE-STEP (CHOOSE-FAIRLY
                               PROCESS-LIST
                               (SUB1 MAX-CLOCK))
                             PROCESS-LIST
                             (SUB1 MAX-CLOCK)
                             (INT PROCESS-LIST
                                  CLOCK
                                  (SUB1 MAX-CLOCK)
                                  SYSTEM-STATE))
                (INT PROCESS-LIST
```

```
                        CLOCK
                        MAX-CLOCK
                        SYSTEM-STATE)))

((DISABLE SINGLE-STEP)
 (ENABLE SINGLE-STEP-FIXES-CLOCK))
```

Bad-State is a skolem function that returns the Clock associated with the first system state between Clock and Max-Clock that does not satisfy the Invariant. If no such state exists, return a Clock greater than or equal to Max-Clock. By not satisfiying the Invariant, we mean, that the Invariant holds on the system state at Clock, and the Invariant does not hold on the next system state, derived from the system state by the Single-Step of the process chosen by Choose-Fairly at Clock.

Definition 4: (BAD-STATE INVARIANT CODE PROCESS-LIST CLOCK MAX-CLOCK SYSTEM-STATE) =

```
(IF (LESSP CLOCK MAX-CLOCK)
    (IF (NOT
          (IMPLIES
            (APPLY$ INVARIANT
                    (APPEND CODE (LIST SYSTEM-STATE)))
            (APPLY$ INVARIANT
                    (APPEND CODE
                            (LIST (SINGLE-STEP
                                    (CHOOSE-FAIRLY
                                      PROCESS-LIST CLOCK)
                                    PROCESS-LIST
                                    CLOCK
                                    SYSTEM-STATE))))))
        CLOCK
        (BAD-STATE INVARIANT
                   CODE
                   PROCESS-LIST
                   (ADD1 CLOCK)
                   MAX-CLOCK
                   (SINGLE-STEP (CHOOSE-FAIRLY
                                  PROCESS-LIST CLOCK)
                                PROCESS-LIST
                                CLOCK
                                SYSTEM-STATE)))
    CLOCK)

((LESSP (DIFFERENCE MAX-CLOCK CLOCK)))
```

When will Bad-State return the first clock?

Theorem 5: ABOUT-BAD-STATE-1 (REWRITE)

```
(IMPLIES (AND (LESSP CLOCK MAX-CLOCK)
              (NOT (IMPLIES
                     (APPLY$ INVARIANT
                             (APPEND CODE
```

```
                               (LIST SYSTEM-STATE)))
                (APPLY$
                  INVARIANT
                  (APPEND CODE
                          (LIST (SINGLE-STEP
                                  (CHOOSE-FAIRLY
                                    PROCESS-LIST
                                    CLOCK)
                                  PROCESS-LIST
                                  CLOCK
                                  SYSTEM-STATE))))))
    (EQUAL (BAD-STATE INVARIANT
                      CODE
                      PROCESS-LIST
                      CLOCK
                      MAX-CLOCK
                      SYSTEM-STATE)
           CLOCK))
```

Bad-State always returns a value greater than or equal to Clock.

Theorem 6: ABOUT-BAD-STATE-2 (REWRITE)

```
(NOT (LESSP (BAD-STATE INVARIANT
                       CODE
                       PROCESS-LIST
                       CLOCK
                       MAX-CLOCK
                       SYSTEM-STATE)
            CLOCK))

((DISABLE SINGLE-STEP)
 (ENABLE SINGLE-STEP-FIXES-CLOCK))
```

Using Bad-State, Invariance-Over-Single-Step defines what is necessary for the Invariant to be stable over the interval [Clock, Max-Clock).

Definition 7: (INVARIANCE-OVER-SINGLE-STEP INVARIANT CODE PROCESS-LIST CLOCK MAX-CLOCK SYSTEM-STATE) =

```
(IMPLIES (AND (LESSP (BAD-STATE INVARIANT
                                CODE
                                PROCESS-LIST
                                CLOCK
                                MAX-CLOCK
                                SYSTEM-STATE)
                     MAX-CLOCK)
              (APPLY$ INVARIANT
                      (APPEND
                        CODE
                        (LIST
                          (INT PROCESS-LIST
                               CLOCK
                               (BAD-STATE INVARIANT
```

```
                                             CODE
                                             PROCESS-LIST
                                             CLOCK
                                             MAX-CLOCK
                                             SYSTEM-STATE)
                                  SYSTEM-STATE)))))
                (APPLY$ INVARIANT
                        (APPEND
                          CODE
                          (LIST (SINGLE-STEP
                                  (CHOOSE-FAIRLY
                                    PROCESS-LIST
                                    (BAD-STATE INVARIANT
                                               CODE
                                               PROCESS-LIST
                                               CLOCK
                                               MAX-CLOCK
                                               SYSTEM-STATE))
                                  PROCESS-LIST
                                  (BAD-STATE INVARIANT
                                             CODE
                                             PROCESS-LIST
                                             CLOCK
                                             MAX-CLOCK
                                             SYSTEM-STATE)
                                  (INT PROCESS-LIST
                                       CLOCK
                                       (BAD-STATE INVARIANT
                                                  CODE
                                                  PROCESS-LIST
                                                  CLOCK
                                                  MAX-CLOCK
                                                  SYSTEM-STATE)
                                       SYSTEM-STATE))))))
```

This is really a stablitiy rather than an interval proof rule.

Theorem 8: INVARIANCE (REWRITE)

```
(IMPLIES (INVARIANCE-OVER-SINGLE-STEP INVARIANT
                                      CODE
                                      PROCESS-LIST
                                      CLOCK
                                      MAX-CLOCK
                                      SYSTEM-STATE)
         (EQUAL (IMPLIES (APPLY$ INVARIANT
                                 (APPEND
                                   CODE
                                   (LIST SYSTEM-STATE)))
                         (APPLY$ INVARIANT
                                 (APPEND
                                   CODE
                                   (LIST
                                     (INT PROCESS-LIST
                                          CLOCK
                                          MAX-CLOCK
```

```
                                        SYSTEM-STATE)))))
                T))

((DISABLE SINGLE-STEP)
 (ENABLE SINGLE-STEP-FIXES-CLOCK)
 (INDUCT (BAD-STATE INVARIANT CODE PROCESS-LIST
                    CLOCK MAX-CLOCK SYSTEM-STATE)))
```

The Progress Proof Rule. If the Intermediate-Property is stable over [Clock, Max-Clock-1), and the final transition causes the Final-Property to hold, then we have demonstrated progress from the Intermediate-Property to the Final-Property, over that interval.

Theorem 9: GENERAL-PROGRESS (REWRITE)

```
(IMPLIES (AND (LESSP CLOCK MAX-CLOCK)
              (INVARIANCE-OVER-SINGLE-STEP
                INTERMEDIATE-PROPERTY
                I-CODE
                PROCESS-LIST
                CLOCK
                (SUB1 MAX-CLOCK)
                SYSTEM-STATE)
              (IMPLIES (APPLY$
                         INTERMEDIATE-PROPERTY
                         (APPEND I-CODE
                                 (LIST (INT PROCESS-LIST
                                            CLOCK
                                            (SUB1 MAX-CLOCK)
                                            SYSTEM-STATE))))
                       (APPLY$
                         FINAL-PROPERTY
                         (APPEND
                           F-CODE
                           (LIST
                             (SINGLE-STEP
                               (CHOOSE-FAIRLY
                                 PROCESS-LIST
                                 (SUB1 MAX-CLOCK))
                               PROCESS-LIST
                               (SUB1 MAX-CLOCK)
                               (INT PROCESS-LIST
                                    CLOCK
                                    (SUB1 MAX-CLOCK)
                                    SYSTEM-STATE)))))))
         (EQUAL
           (IMPLIES (APPLY$ INTERMEDIATE-PROPERTY
                            (APPEND I-CODE
                                    (LIST
                                      SYSTEM-STATE)))
                    (APPLY$
                      FINAL-PROPERTY
                      (APPEND
                        F-CODE
                        (LIST (INT PROCESS-LIST
```

```
                                            CLOCK
                                            MAX-CLOCK
                                            SYSTEM-STATE)))))
          T))

((DISABLE SINGLE-STEP INVARIANCE-OVER-SINGLE-STEP
          INVARIANCE)
 (ENABLE SINGLE-STEP-FIXES-CLOCK)
 (USE (INVARIANCE (INVARIANT INTERMEDIATE-PROPERTY)
                  (CODE I-CODE)
                  (CLOCK CLOCK)
                  (MAX-CLOCK (SUB1 MAX-CLOCK))
                  (PROCESS-LIST PROCESS-LIST))))
```

Int is now disabled, since the semantics that we need from it are provided by these Proof Rules. However, we make no claim that these proof rules completely define Int.

```
(DISABLE INT)                                                    (1)
```

Appendix G
State Accessors

This file defines functions that are useful, when a process accesses the system state.

Return the Name that a process uses to access its locals in the system state.

Definition 1: (PROCESS-NAME PROCESS-DESCRIPTION) =

```
(CONS (CAR PROCESS-DESCRIPTION)
      (CADR PROCESS-DESCRIPTION))
```

Replace the state associated with Process-Name in System-State with New-Process-State. (Update-State is a better name than Update-Assoc).

Definition 2: (UPDATE-STATE PROCESS-NAME NEW-PROCESS-STATE SYSTEM-STATE) =

```
(UPDATE-ASSOC PROCESS-NAME
              NEW-PROCESS-STATE
              SYSTEM-STATE)
```

Returns the current state of process Process-Name. This function is also commonly used for channels.

Definition 3: (GET-STATE PROCESS-NAME SYSTEM-STATE) =

```
(CDR (ASSOC PROCESS-NAME SYSTEM-STATE))
```

Now, define the channel communication functions. Channels are FIFO queues, modeled by lists.

The channel is empty if its value is not a list.

Definition 4: (CHANNEL-EMPTYP CHANNEL-NAME SYSTEM-STATE) =

```
(NLISTP (CDR (ASSOC CHANNEL-NAME SYSTEM-STATE)))
```

Always return the Car of the channel (when the channel is not empty, that is the first message on the channel). Use Channel-Emptyp to see whether there is a message in the channel.

Definition 5: (GET-HEAD CHANNEL-NAME SYSTEM-STATE) =

```
(CADR (ASSOC CHANNEL-NAME SYSTEM-STATE))
```

Add the message to the the end of the specified channel.

Definition 6: (SEND MESSAGE CHANNEL-NAME SYSTEM-STATE) =

```
(UPDATE-ASSOC CHANNEL-NAME
              (APPEND (CDR (ASSOC CHANNEL-NAME
                                  SYSTEM-STATE))
                      (LIST MESSAGE))
              SYSTEM-STATE)
```

Shorten the specified channel.

Definition 7: (RECEIVE CHANNEL-NAME SYSTEM-STATE) =

```
(UPDATE-ASSOC CHANNEL-NAME
              (CDDR (ASSOC CHANNEL-NAME SYSTEM-STATE))
              SYSTEM-STATE)
```

Appendix H
Defintion of Me

This file defines the system of processes that solves the N-Processor Mutual Exclusion problem.

A Non-Deterministic Boolean function.

Definition 1:

```
(DCL RANDOM-BOOLEAN (CODE))
```

Random-Boolean is total. This should do it, but doesn’t.

Theorem 2:

```
(ADD-AXIOM RANDOM-BOOLEAN-IS-SUBRP (REWRITE)
        (SUBRP ’RANDOM-BOOLEAN))

(PUT1 ’RANDOM-BOOLEAN *1*T ’SUBRP)                             (1)
(PUT1 ’RANDOM-BOOLEAN ’((RANDOM-BOOLEAN . T)) ’TOTALP-LST)
```

Theorem 3:

```
(ADD-AXIOM RANDOM-BOOLEAN-IS-BOOLEAN (REWRITE)
           (OR (TRUEP (RANDOM-BOOLEAN CODE))
               (FALSEP (RANDOM-BOOLEAN CODE))))
```

A "Random" Number function.

Definition 4:

```
(DCL RANDOM-NUMBER (CODE))
```

Random-Number is total. This should do it, but doesn’t.

Theorem 5:

```
(ADD-AXIOM RANDOM-NUMBER-IS-SUBRP (REWRITE)
           (SUBRP ’RANDOM-NUMBER))

(PUT1 ’RANDOM-NUMBER *1*T ’SUBRP)                              (2)
(PUT1 ’RANDOM-NUMBER ’((RANDOM-NUMBER . T)) ’TOTALP-LST)
```

Theorem 6:

```
(ADD-AXIOM RANDOM-NUMBER-IS-NUMBERP (REWRITE)
           (NUMBERP (RANDOM-NUMBER CODE)))
```

Define a process ME of three states: Non-Critical, Wait, and Critical (not Non-Critical or Wait ; usually a number.). The process moves from Non-Critical to Wait nondeterministically. Otherwise, it passes a message from its incoming to its outgoing channel. The process moves from Wait to Critical upon receipt of a message on its distinguished incoming channel. Upon each transition in a Critical state, the process decrements its counter. If it is zero, the process goes to its Non-Critical state and sends a token its outgoing channel.

Definition 7: (ME INDEX FAMILY-SIZE CLOCK SYSTEM-STATE) =

```
(IF (EQUAL (GET-STATE (CONS 'ME INDEX) SYSTEM-STATE)
           'NON-CRITICAL)
    (IF (RANDOM-BOOLEAN CLOCK)
        (UPDATE-STATE (CONS 'ME INDEX)
                      'WAIT
                      SYSTEM-STATE)
        (IF (CHANNEL-EMPTYP (CONS 'C INDEX) SYSTEM-STATE)
            SYSTEM-STATE
            (SEND 'TOKEN
                  (CONS 'C (ADD1-MOD FAMILY-SIZE INDEX))
                  (RECEIVE (CONS 'C INDEX)
                           SYSTEM-STATE))))
    (IF (EQUAL (GET-STATE (CONS 'ME INDEX) SYSTEM-STATE)
               'WAIT)
        (IF (CHANNEL-EMPTYP (CONS 'C INDEX) SYSTEM-STATE)
            SYSTEM-STATE
            (UPDATE-STATE (CONS 'ME INDEX)
                          (RANDOM-NUMBER CLOCK)
                          (RECEIVE (CONS 'C INDEX)
                                   SYSTEM-STATE)))
        (IF (ZEROP (GET-STATE (CONS 'ME INDEX)
                              SYSTEM-STATE))
            (UPDATE-STATE
              (CONS 'ME INDEX) 'NON-CRITICAL
              (SEND 'TOKEN
                    (CONS 'C (ADD1-MOD FAMILY-SIZE INDEX))
                    SYSTEM-STATE))
            (UPDATE-STATE
              (CONS 'ME INDEX)
              (SUB1 (GET-STATE (CONS 'ME INDEX)
                               SYSTEM-STATE))
              SYSTEM-STATE))))
```

The system contains N instances of Me.

Definition 8: (SYSTEM-DESCRIPTION N) =

```
(LIST (LIST 'ME N))
```

Three sets of lemmas that are needed for mechanication.

The family names in the system description are a set.

Theorem 9: SETP-FAMILY-NAMES-SYSTEM-DESCRIPTION (REWRITE)

```
(SETP (FAMILY-NAMES (SYSTEM-DESCRIPTION N)))

((ENABLE FAMILY-NAMES))
```

For each process, simplify the declared family size. This system has only one process, Me.

Theorem 10: DECLARED-FAMILY-SIZE-OF-ME-IS-N (REWRITE)

```
(EQUAL (DECLARED-FAMILY-SIZE 'ME (SYSTEM-DESCRIPTION N))
       (FIX N))

((ENABLE DECLARED-FAMILY-SIZE))
```

Provide necessary and sufficient conditions for membership in the family-names. With more than one family, the second argument is a disjunct of equalities.

Theorem 11: MEMBER-FAMILY-NAMES-OF-SYSTEM-DESCRIPTION (REWRITE)

```
(EQUAL (MEMBER PROCESS-NAME
               (FAMILY-NAMES (SYSTEM-DESCRIPTION N)))
       (EQUAL PROCESS-NAME 'ME))

((ENABLE FAMILY-NAMES))
```

System-Description is then disabled as its executable counterpart.

```
(DISABLE SYSTEM-DESCRIPTION)                                    (3)
(DISABLE *1*SYSTEM-DESCRIPTION)
```

Appendix I
The Proof of Invariance

Proof of Invariance.

Prove that "at most one process is Critical" is invariant.

First prove that the weight of the system is stable across any transition of Me. This proof is done in two parts: for ring sizes of 0 and 1, and for larger ring sizes. First, larger ring sizes. Split ring into two pieces, an arbitrary process and its left and right channels, and the rest of the ring. Show that each is stable for a transition of that arbitrary process. Second, prove stability by case analysis for ring sizes of 0 and 1.

The weight of a process is one if it is Critical.

Definition 1: (WEIGHT-OF-PROCESS INDEX SYSTEM-STATE) =

```
(IF (OR (EQUAL (GET-STATE (CONS 'ME INDEX) SYSTEM-STATE)
               'NON-CRITICAL)
        (EQUAL (GET-STATE (CONS 'ME INDEX) SYSTEM-STATE)
               'WAIT))
    0
    1)
```

The weight of the Index'ed processes Left and Right channels.

Definition 2: (WEIGHT-OF-DOUBLE INDEX N SYSTEM-STATE) =

```
(PLUS (LENGTH (GET-STATE (CONS 'C INDEX) SYSTEM-STATE))
      (LENGTH (GET-STATE (CONS 'C (ADD1-MOD N INDEX))
                         SYSTEM-STATE)))
```

The weight of the process and its surrounding channels.

Definition 3: (WEIGHT-OF-TRIPLE INDEX N SYSTEM-STATE) =

```
(PLUS (WEIGHT-OF-DOUBLE INDEX N SYSTEM-STATE)
      (WEIGHT-OF-PROCESS INDEX SYSTEM-STATE))
```

Prove, by case analysis that the weight of the triple is stable for a single transition of the process at the center of the triple.

Theorem 4: WEIGHT-OF-TRIPLE-INVARIANT-OVER-ME (REWRITE)

```
(IMPLIES (LESSP 1 N)
```

```
(EQUAL (WEIGHT-OF-TRIPLE INDEX N (ME INDEX
                                     N
                                     CLOCK
                                     SYSTEM-STATE))
       (WEIGHT-OF-TRIPLE INDEX N SYSTEM-STATE)))
```

The non-interference property for channels outside of the triple.

Theorem 5: ME-DOES-NOT-TOUCH-CHANNEL (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL I INDEX))
              (NOT (EQUAL I (ADD1-MOD N INDEX))))
         (EQUAL (CDR (ASSOC (CONS 'C I)
                            (ME INDEX
                                N
                                CLOCK
                                SYSTEM-STATE)))
                (CDR (ASSOC (CONS 'C I) SYSTEM-STATE))))
```

The non-interference property for local states outside of the triple.

Theorem 6: ME-DOES-NOT-TOUCH-LOCAL (REWRITE)

```
(IMPLIES (NOT (EQUAL I INDEX))
         (EQUAL (CDR (ASSOC (CONS 'ME I)
                            (ME INDEX
                                N
                                CLOCK
                                SYSTEM-STATE)))
                (CDR (ASSOC (CONS 'ME I) SYSTEM-STATE))))
```

The weight of all the channels outside of the triple.

Definition 7: (WEIGHT-OF-REST-OF-CHANNELS RECURSE INDEX N SYSTEM-STATE) =

```
(IF (ZEROP RECURSE)
    0
    (IF (EQUAL INDEX (SUB1 RECURSE))
        (WEIGHT-OF-REST-OF-CHANNELS
          (SUB1 RECURSE) INDEX N SYSTEM-STATE)
        (IF (EQUAL (ADD1-MOD N INDEX) (SUB1 RECURSE))
            (WEIGHT-OF-REST-OF-CHANNELS
              (SUB1 RECURSE) INDEX N SYSTEM-STATE)
            (PLUS (LENGTH (GET-STATE
                            (CONS 'C (SUB1 RECURSE))
                            SYSTEM-STATE))
                  (WEIGHT-OF-REST-OF-CHANNELS
                    (SUB1 RECURSE) INDEX N
                    SYSTEM-STATE)))))
```

The weight of all the processes except for the Index'ed process.

Definition 8: (WEIGHT-OF-REST-OF-PROCESSES N INDEX SYSTEM-STATE) =

```
(IF (ZEROP N)
    0
    (IF (EQUAL INDEX (SUB1 N))
        (WEIGHT-OF-REST-OF-PROCESSES
          (SUB1 N) INDEX SYSTEM-STATE)
        (PLUS (WEIGHT-OF-PROCESS (SUB1 N) SYSTEM-STATE)
              (WEIGHT-OF-REST-OF-PROCESSES
                (SUB1 N)
                INDEX SYSTEM-STATE))))
```

Proved by the non-interfence over channels property.

Theorem 9: WEIGHT-OF-REST-OF-CHANNELS-INVARIANT-OVER-ME (REWRITE)

```
(EQUAL (WEIGHT-OF-REST-OF-CHANNELS RECURSE INDEX N
                                   (ME INDEX
                                       N
                                       CLOCK
                                       SYSTEM-STATE))
       (WEIGHT-OF-REST-OF-CHANNELS
         RECURSE INDEX N SYSTEM-STATE))

((DISABLE ADD1-MOD ME))
```

Proved by the non-interference over local states property.

Theorem 10: WEIGHT-OF-REST-OF-PROCESSES-INVARIANT-OVER-ME (REWRITE)

```
(EQUAL (WEIGHT-OF-REST-OF-PROCESSES RECURSE INDEX
                                    (ME INDEX
                                        N
                                        CLOCK
                                        SYSTEM-STATE))
       (WEIGHT-OF-REST-OF-PROCESSES
         RECURSE INDEX SYSTEM-STATE))

((DISABLE ADD1-MOD ME))
```

The weight of all the channels.

Definition 11: (WEIGHT-OF-CHANNELS N SYSTEM-STATE) =

```
(IF (ZEROP N)
    0
    (PLUS (LENGTH (GET-STATE (CONS 'C (SUB1 N))
                             SYSTEM-STATE))
          (WEIGHT-OF-CHANNELS
            (SUB1 N) SYSTEM-STATE)))
```

The weight of all the processes.

Definition 12: (WEIGHT-OF-PROCESSES N SYSTEM-STATE) =

```
(IF (ZEROP N)
    0
    (PLUS (WEIGHT-OF-PROCESS
            (SUB1 N) SYSTEM-STATE)
          (WEIGHT-OF-PROCESSES
            (SUB1 N) SYSTEM-STATE)))
```

The weight of the system is the sum of the weight of the triple and the weight of the rest of the channels and processes. To do this by induction requires a more general theorem.

Theorem 13: WEIGHT-OF-SYSTEM-STATE-IS-SUM-GENERAL ()

```
(IMPLIES (AND (NOT (LESSP N RECURSE))
              (NUMBERP INDEX)
              (LESSP INDEX N))
         (EQUAL (PLUS (WEIGHT-OF-CHANNELS
                        RECURSE SYSTEM-STATE)
                      (WEIGHT-OF-PROCESSES
                        RECURSE SYSTEM-STATE))
(IF (ZEROP RECURSE)
    0
(IF (EQUAL RECURSE 1)
    (PLUS (LENGTH (CDR (ASSOC (CONS 'C 0)
                              SYSTEM-STATE)))
          (WEIGHT-OF-PROCESS 0 SYSTEM-STATE))
    (PLUS (WEIGHT-OF-REST-OF-PROCESSES
            RECURSE INDEX SYSTEM-STATE)
          (WEIGHT-OF-REST-OF-CHANNELS
            RECURSE INDEX N SYSTEM-STATE)
(IF (EQUAL INDEX (SUB1 N))
(IF (LESSP INDEX RECURSE)
    (WEIGHT-OF-TRIPLE INDEX N SYSTEM-STATE)
    (LENGTH (CDR (ASSOC (CONS 'C 0)
                        SYSTEM-STATE))))
(IF (LESSP (ADD1 INDEX) RECURSE)
    (WEIGHT-OF-TRIPLE INDEX N SYSTEM-STATE)
(IF (EQUAL (ADD1 INDEX) RECURSE)
    (PLUS (WEIGHT-OF-PROCESS INDEX SYSTEM-STATE)
          (LENGTH (CDR (ASSOC (CONS 'C INDEX)
                              SYSTEM-STATE))))
    0))))))))

((DISABLE WEIGHT-OF-PROCESS)
 (INDUCT (INTEGER-INDUCTION-SCHEME-1 RECURSE)))
```

Simply that the weight of the channels and processes is the sum of the stable parts: the triple, the rest of channels, and the rest of processes.

Theorem 14: WEIGHT-OF-SYSTEM-STATE-IS-SUM-OF-REST-AND-TRIPLE (REWRITE)

```
(IMPLIES (AND (LESSP 1 N)
              (NUMBERP INDEX)
```

```
              (LESSP INDEX N))
         (EQUAL (PLUS (WEIGHT-OF-CHANNELS N SYSTEM-STATE)
                      (WEIGHT-OF-PROCESSES N SYSTEM-STATE))
                (PLUS (WEIGHT-OF-TRIPLE
                        INDEX N SYSTEM-STATE)
                      (WEIGHT-OF-REST-OF-CHANNELS
                        N INDEX N SYSTEM-STATE)
                      (WEIGHT-OF-REST-OF-PROCESSES
                        N INDEX SYSTEM-STATE))))

((DISABLE WEIGHT-OF-TRIPLE)
 (USE (WEIGHT-OF-SYSTEM-STATE-IS-SUM-GENERAL
        (INDEX INDEX) (RECURSE N) (N N))))
```

The stability result for larger ring sizes (greater than 1).

Theorem 15: WEIGHT-OF-SYSTEM-STATE-INVARIANT-OVER-ME-N>1 ()

```
(IMPLIES (AND (LESSP 1 N)
              (NUMBERP INDEX)
              (LESSP INDEX N))
         (EQUAL (PLUS (WEIGHT-OF-CHANNELS N
                                          (ME INDEX
                                             N
                                              CLOCK
                                                SYSTEM-STATE))
                      (WEIGHT-OF-PROCESSES N
                                             (ME INDEX
                                                 N
                                                 CLOCK
                                              SYSTEM-STATE)))
                (PLUS (WEIGHT-OF-CHANNELS N SYSTEM-STATE)
                    (WEIGHT-OF-PROCESSES N SYSTEM-STATE))))

((DISABLE ME WEIGHT-OF-TRIPLE))

(DISABLE WEIGHT-OF-SYSTEM-STATE-IS-SUM-OF-REST-AND-TRIPLE)       (1)
(DISABLE WEIGHT-OF-TRIPLE-INVARIANT-OVER-ME)
(DISABLE WEIGHT-OF-REST-OF-CHANNELS-INVARIANT-OVER-ME)
(DISABLE WEIGHT-OF-REST-OF-PROCESSES-INVARIANT-OVER-ME)
```

Prove by case analysis the stability result for ring size 1. The previous technique cannot be used, since the left and right channels are the same.

Theorem 16: WEIGHT-OF-SYSTEM-STATE-INVARIANT-OVER-ME-N=1 (REWRITE)

```
(EQUAL (PLUS (WEIGHT-OF-CHANNELS 1 (ME 0
                                       1
                                       CLOCK
                                       SYSTEM-STATE))
             (WEIGHT-OF-PROCESSES 1 (ME 0
                                        1
                                        CLOCK
```

```
                                              SYSTEM-STATE)))
        (PLUS (WEIGHT-OF-CHANNELS 1 SYSTEM-STATE)
              (WEIGHT-OF-PROCESSES 1 SYSTEM-STATE)))

((USE (WEIGHT-OF-PROCESSES (N 1)
                           (SYSTEM-STATE
                             (ME 0 1
                                 CLOCK
                                 SYSTEM-STATE)))
      (WEIGHT-OF-CHANNELS (N 1)
                          (SYSTEM-STATE
                            (ME 0 1
                                CLOCK
                                SYSTEM-STATE)))
      (WEIGHT-OF-PROCESSES (N 1)
                           (SYSTEM-STATE SYSTEM-STATE))
      (WEIGHT-OF-CHANNELS (N 1)
                          (SYSTEM-STATE SYSTEM-STATE))))
```

Show that if N is less that or equal to 1 and there exists an Index less that it, the N is 1 and the Index is 0.

Theorem 17: ABOUT-LESSP-LESSP-COMBINATION ()

```
(EQUAL (AND (NOT (LESSP 1 N))
            (NUMBERP INDEX)
            (LESSP INDEX N))
       (AND (EQUAL N 1)
            (EQUAL INDEX 0)))
```

For N=1, true by case anayis (done earlier). For N=0, no index exists.

Theorem 18: WEIGHT-OF-SYSTEM-STATE-INVARIANT-OVER-ME-N<2 ()

```
(IMPLIES (AND (NOT (LESSP 1 N))
              (NUMBERP INDEX)
              (LESSP INDEX N))
         (EQUAL (PLUS (WEIGHT-OF-CHANNELS
                        N
                        (ME INDEX N
                            CLOCK
                            SYSTEM-STATE))
                      (WEIGHT-OF-PROCESSES
                        N
                        (ME INDEX N
                            CLOCK
                            SYSTEM-STATE)))
                (PLUS (WEIGHT-OF-CHANNELS
                        N SYSTEM-STATE)
                      (WEIGHT-OF-PROCESSES
                        N SYSTEM-STATE))))

((DISABLE ME)
 (USE (ABOUT-LESSP-LESSP-COMBINATION)))
```

```
(DISABLE WEIGHT-OF-SYSTEM-STATE-INVARIANT-OVER-ME-N=1)          (2)
```

Combining the two proofs (for large and small ring sizes) yields stability over the ring, for all ring sizes.

Theorem 19: WEIGHT-OF-SYSTEM-STATE-INVARIANT-OVER-ME (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX N))
         (EQUAL (PLUS (WEIGHT-OF-CHANNELS
                        N
                        (ME INDEX N
                            CLOCK
                            SYSTEM-STATE))
                      (WEIGHT-OF-PROCESSES
                        N
                        (ME INDEX N
                            CLOCK
                            SYSTEM-STATE)))
                (PLUS (WEIGHT-OF-CHANNELS
                        N SYSTEM-STATE)
                      (WEIGHT-OF-PROCESSES
                        N SYSTEM-STATE))))

((DISABLE ME)
 (USE (WEIGHT-OF-SYSTEM-STATE-INVARIANT-OVER-ME-N>1)
      (WEIGHT-OF-SYSTEM-STATE-INVARIANT-OVER-ME-N<2)))
```

Now prove that mutual exclusion is invariant. Mutual Exclusion is the property that the weight of the system is always one.

Is the weight of the system one?

Definition 20: (MUTUAL-EXCLUSIONP N SYSTEM-STATE) =

```
(EQUAL (PLUS (WEIGHT-OF-CHANNELS N SYSTEM-STATE)
             (WEIGHT-OF-PROCESSES N SYSTEM-STATE))
       1)
```

Prove the process level invariance of Mutual-Exclusionp. Because the weight of the system is stable, mutual exclusion certainly is.

Theorem 21: MUTUAL-EXCLUSION-INVARIANT-OVER-ME (REWRITE)

```
(IMPLIES (AND (LESSP INDEX N)
              (NUMBERP INDEX)
              (MUTUAL-EXCLUSIONP N
                                 SYSTEM-STATE))
         (MUTUAL-EXCLUSIONP N
                            (ME INDEX N
                                CLOCK
```

```
                                   SYSTEM-STATE)))

((DISABLE ME))
```

Using the process level invariance, prove the system level invariance.

Theorem 22: MUTUAL-EXCLUSION-IS-INVARIANT (REWRITE)

```
(IMPLIES (MUTUAL-EXCLUSIONP N
                            SYSTEM-STATE)
         (MUTUAL-EXCLUSIONP N
                            (INT (MAKE-PROCESS-LIST
                                   (SYSTEM-DESCRIPTION N))
                                 CLOCK
                                 MAX-CLOCK
                                 SYSTEM-STATE)))

((DISABLE ME MUTUAL-EXCLUSIONP)
Disable the family specifications (Me), and the
predicate that is being proved invariant.
Use the Invariance Proof Rule, by instantiating
appropriately.
 (USE (INVARIANCE
        (INVARIANT 'MUTUAL-EXCLUSIONP)
        (CODE (LIST N))
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N)))
        (SYSTEM-STATE SYSTEM-STATE)
        (CLOCK CLOCK)
        (MAX-CLOCK MAX-CLOCK))
Use the Choose-Fairly-Returns-Member-0 theorem
since every process can be run in an arbitrary interval.
Instantiate Clock with the skolemized bad value.
      (CHOOSE-FAIRLY-RETURNS-MEMBER-0
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N)))
        (CLOCK (BAD-STATE 'MUTUAL-EXCLUSIONP
                          (LIST N)
                          (MAKE-PROCESS-LIST
                            (SYSTEM-DESCRIPTION N))
                          CLOCK
                          MAX-CLOCK
                          SYSTEM-STATE)))))
```

Appendix J
The Proof of Liveness

The Proof of Liveness.

Prove that every Waiting process eventually becomes Critical.

First, some consequences of mutual exclusion. Three theorems: A Waiting process with an empty left channel stays Waiting, over the next transition, no matter what that transition is. One full channel precludes any other full channels. One Critical process precludes any full channels.

Definition 1: (WEIGHT-REFLECTS-INDUCTION-SCHEME INDEX N) =

```
(IF (ZEROP N)
    T
    (IF (LESSP INDEX N)
        (IF (EQUAL INDEX (SUB1 N))
            T
            (WEIGHT-REFLECTS-INDUCTION-SCHEME
              INDEX (SUB1 N)))
        T))
```

The weight of all the processes is at the least weight of any single process.

Theorem 2: WEIGHT-OF-PROCESSES-REFLECTS-WEIGHT-OF-COMPONENTS (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX N))
         (NOT (LESSP (WEIGHT-OF-PROCESSES N SYSTEM-STATE)
                     (WEIGHT-OF-PROCESS INDEX SYSTEM-STATE))))

((DISABLE WEIGHT-OF-PROCESS)
 (INDUCT (WEIGHT-REFLECTS-INDUCTION-SCHEME INDEX N)))
```

The weight of all the channels is at least the weight of any single channel.

Theorem 3: WEIGHT-OF-CHANNELS-REFLECTS-WEIGHT-OF-COMPONENTS (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX)
                (LESSP INDEX N))
          (NOT (LESSP (WEIGHT-OF-CHANNELS N SYSTEM-STATE)
                      (LENGTH (CDR (ASSOC (CONS 'C INDEX)
                                          SYSTEM-STATE))))))

((INDUCT (WEIGHT-REFLECTS-INDUCTION-SCHEME INDEX N)))
```

Theorem 4: ABOUT-WEIGHT-OF-REST-OF-CHANNELS (REWRITE)

```
(IMPLIES (AND (EQUAL (WEIGHT-OF-CHANNELS N SYSTEM-STATE)
```

```
                       1)
                (NUMBERP INDEX)
                (LESSP INDEX N)
                (LESSP 0 (LENGTH (CDR (ASSOC (CONS 'C INDEX)
                                             SYSTEM-STATE)))))
           (EQUAL (WEIGHT-OF-CHANNELS INDEX SYSTEM-STATE)
                  0))

  ((INDUCT (WEIGHT-REFLECTS-INDUCTION-SCHEME INDEX N)))
```

Theorem 5: ONE-CHANNEL-FULL-ALL-OTHERS-EMPTY-1 ()

```
  (IMPLIES (AND (EQUAL (WEIGHT-OF-CHANNELS N SYSTEM-STATE)
                       1)
                (NUMBERP I)
                (LESSP I N)
                (LESSP 0 (LENGTH (CDR (ASSOC (CONS 'C I)
                                             SYSTEM-STATE))))
                (NUMBERP J)
                (LESSP J N)
                (LESSP I J))
           (EQUAL (LENGTH (CDR (ASSOC (CONS 'C J)
                                      SYSTEM-STATE)))
                  0))
```

Theorem 6: ONE-CHANNEL-FULL-ALL-OTHERS-EMPTY-2 (REWRITE)

```
  (IMPLIES (AND (EQUAL (WEIGHT-OF-CHANNELS N SYSTEM-STATE)
                       1)
                (NUMBERP I)
                (LESSP I N)
                (LESSP 0 (LENGTH (CDR (ASSOC (CONS 'C I)
                                             SYSTEM-STATE))))
                (NUMBERP J)
                (LESSP J N)
                (NOT (EQUAL I J)))
           (EQUAL (LENGTH (CDR (ASSOC (CONS 'C J)
                                      SYSTEM-STATE)))
                  0))

  ((DISABLE ABOUT-WEIGHT-OF-REST-OF-CHANNELS)
   (USE (ONE-CHANNEL-FULL-ALL-OTHERS-EMPTY-1
          (I I) (J J) (N N) (SYSTEM-STATE SYSTEM-STATE))
        (ONE-CHANNEL-FULL-ALL-OTHERS-EMPTY-1
          (I J) (J I) (N N) (SYSTEM-STATE SYSTEM-STATE))))
```

This is a hack around the linear arithmetic package. It ought to recognize this immediately.

Theorem 7: ABOUT-PLUS-0 (REWRITE)

```
  (IMPLIES (ZEROP Y)
           (EQUAL (PLUS X Y)
                  (FIX X)))
```

Since the weight of the system is the weight of two things, and the weight of the system equals 1, either component must be one, and the other zero.

Theorem 8: ABOUT-PLUS-EQUALS-ONE (REWRITE)

```
(EQUAL (EQUAL (PLUS X Y)
              1)
       (OR (AND (EQUAL X 1)
                (ZEROP Y))
           (AND (ZEROP X)
                (EQUAL Y 1))))
```

If one channel is full, every other is empty.

Theorem 9: ONE-CHANNEL-FULL-ALL-OTHERS-EMPTY (REWRITE)

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (NUMBERP I)
              (LESSP I N)
              (LESSP 0 (LENGTH (CDR (ASSOC (CONS 'C I)
                                           SYSTEM-STATE))))
              (NUMBERP J)
              (LESSP J N)
              (NOT (EQUAL I J)))
         (EQUAL (LENGTH (CDR (ASSOC (CONS 'C J)
                                    SYSTEM-STATE)))
                0))
```

If one process is Critical, all channels are empty.

Theorem 10: ONE-PROCESS-CRITICAL-ALL-CHANNELS-EMPTY (REWRITE)

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (NUMBERP I)
              (LESSP I N)
              (EQUAL (WEIGHT-OF-PROCESS I SYSTEM-STATE)
                     1)
              (NUMBERP J)
              (LESSP J N))
         (EQUAL (LENGTH (CDR (ASSOC (CONS 'C J)
                                    SYSTEM-STATE)))
                0))
```

A bad rewrite rule. It will be disabled soon.

Theorem 11: NOT-LISTPS-HAVE-ZERO-LENGTH (REWRITE)

```
(EQUAL (EQUAL (LENGTH L) 0)
       (NOT (LISTP L)))
```

A Waiting process with an empty channel remains waiting.

Theorem 12: PROCESS-WAIT-WITH-EMPTY-LEFT-STAYS-WAIT (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX N)
              (NUMBERP CINDEX)
              (LESSP CINDEX N)
              (EQUAL (LENGTH (CDR (ASSOC (CONS 'C INDEX)
                                         SYSTEM-STATE)))
                     0)
              (EQUAL (CDR (ASSOC (CONS 'ME INDEX)
                                 SYSTEM-STATE))
                     'WAIT))
         (EQUAL (CDR (ASSOC (CONS 'ME INDEX)
                            (ME CINDEX
                                N
                                CLOCK
                                SYSTEM-STATE)))
                'WAIT))

((USE (ME-DOES-NOT-TOUCH-LOCAL (I INDEX) (INDEX CINDEX))))

(DISABLE NOT-LISTPS-HAVE-ZERO-LENGTH)                            (1)
(DISABLE ABOUT-PLUS-EQUALS-ONE)
(DISABLE ABOUT-PLUS-0)
(DISABLE ONE-CHANNEL-FULL-ALL-OTHERS-EMPTY-2)
(DISABLE ABOUT-WEIGHT-OF-REST-OF-CHANNELS)
```

Some basic abbreviation to describe channels and processes. Some predicates with which we prove progress statements.

Is the Index'th channel full?

Definition 13: (LEFT-CHANNEL-FULL INDEX SYSTEM-STATE) =

```
(LESSP 0 (LENGTH (GET-STATE (CONS 'C INDEX)
                            SYSTEM-STATE)))
```

Is the Index'th+1 Mod N channel full?

Definition 14: (RIGHT-CHANNEL-FULL INDEX N SYSTEM-STATE) =

```
(LESSP 0 (LENGTH (GET-STATE (CONS 'C (ADD1-MOD N INDEX))
                            SYSTEM-STATE)))
```

Is the Index'th process Non-Critical?

Definition 15: (PROCESS-NON-CRITICAL INDEX SYSTEM-STATE) =

```
(EQUAL (GET-STATE (CONS 'ME INDEX)
                  SYSTEM-STATE)
       'NON-CRITICAL)
```

Is the Index'th process Waiting?

Definition 16: (PROCESS-WAIT INDEX SYSTEM-STATE) =

```
(EQUAL (GET-STATE (CONS 'ME INDEX)
                  SYSTEM-STATE)
       'WAIT)
```

Is the Index'th process Critical?

Definition 17: (PROCESS-CRITICAL INDEX SYSTEM-STATE) =

```
(EQUAL (WEIGHT-OF-PROCESS INDEX SYSTEM-STATE)
       1)
```

Is the Index'th process Critical and have a critical counter numerically equal to Ticks?

Definition 18: (PROCESS-CRITICAL-WITH-TICKS TICKS INDEX SYSTEM-STATE) =

```
(AND (PROCESS-CRITICAL INDEX SYSTEM-STATE)
     (EQUAL (FIX (GET-STATE (CONS 'ME INDEX)
                            SYSTEM-STATE))
            (FIX TICKS)))
```

Is the Index1'th channel full, the Index1'th process Non-Critical, and the Index2'th process Waiting, and the system satisfiying mutual exclusion?

Definition 19: (LEFT-FULL-NON-CRITICAL-WAIT INDEX1 INDEX2 N SYSTEM-STATE) =

```
(AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
     (LEFT-CHANNEL-FULL INDEX1 SYSTEM-STATE)
     (PROCESS-NON-CRITICAL INDEX1 SYSTEM-STATE)
     (PROCESS-WAIT INDEX2 SYSTEM-STATE))
```

We now prove four progress statements. For each progress statement, prove that some property is Stable over an interval contaning everything but a specific transition then prove that that Key transition produces the final property.

Progress Statement #1.

Theorem 20: LEFT-FULL-NON-CRITICAL-WAIT-IS-STABLE-1 (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
              (NUMBERP CINDEX)
              (LESSP CINDEX N)
              (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (LEFT-FULL-NON-CRITICAL-WAIT INDEX1 INDEX2 N
```

```
                                         SYSTEM-STATE))
          (LEFT-CHANNEL-FULL INDEX1
                             (ME CINDEX
                                 N
                                 CLOCK
                                 SYSTEM-STATE)))
```

Theorem 21: LEFT-FULL-NON-CRITICAL-WAIT-IS-STABLE-2 (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
              (NUMBERP CINDEX)
              (LESSP CINDEX N)
              (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (LEFT-FULL-NON-CRITICAL-WAIT INDEX1 INDEX2 N
                                           SYSTEM-STATE))
         (PROCESS-NON-CRITICAL INDEX1
                               (ME CINDEX
                                   N
                                   CLOCK
                                   SYSTEM-STATE)))

((DISABLE ME))
```

Theorem 22: LEFT-FULL-NON-CRITICAL-WAIT-IS-STABLE-3 (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
              (LESSP INDEX1 N)
              (NUMBERP INDEX1)
              (LESSP CINDEX N)
              (NUMBERP CINDEX)
              (LESSP INDEX2 N)
              (NUMBERP INDEX2)
              (LEFT-FULL-NON-CRITICAL-WAIT INDEX1 INDEX2 N
                                           SYSTEM-STATE))
         (PROCESS-WAIT INDEX2
           (ME CINDEX
               N
               CLOCK
               SYSTEM-STATE)))

((DISABLE ME MUTUAL-EXCLUSIONP)
 (USE (ONE-CHANNEL-FULL-ALL-OTHERS-EMPTY
        (I INDEX1) (J INDEX2))))
```

Theorem 23: LEFT-FULL-NON-CRITICAL-WAIT-IS-STABLE (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
              (NUMBERP CINDEX)
              (LESSP CINDEX N)
              (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (LEFT-FULL-NON-CRITICAL-WAIT INDEX1 INDEX2 N
                                           SYSTEM-STATE))
```

```
        (LEFT-FULL-NON-CRITICAL-WAIT INDEX1 INDEX2 N
                                     (ME CINDEX
                                         N
                                         CLOCK
                                         SYSTEM-STATE)))

((DISABLE ME MUTUAL-EXCLUSIONP PROCESS-WAIT
          LEFT-CHANNEL-FULL PROCESS-NON-CRITICAL))

(DISABLE LEFT-FULL-NON-CRITICAL-WAIT-IS-STABLE-3)               (2)
(DISABLE LEFT-FULL-NON-CRITICAL-WAIT-IS-STABLE-2)
(DISABLE LEFT-FULL-NON-CRITICAL-WAIT-IS-STABLE-1)
```

Based on Boolean, did the Index1'th process change to Non-Critical or Wait? If Non-Critical, was the token passed to the right channel? Is the Index2'th process still Waiting?

Definition 24: (NON-CRITICAL-OR-WAIT-WAIT BOOLEAN INDEX1 INDEX2 N SYSTEM-STATE) =

```
(AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
     (PROCESS-WAIT INDEX2 SYSTEM-STATE)
     (IF BOOLEAN
         (AND (LEFT-CHANNEL-FULL INDEX1 SYSTEM-STATE)
              (PROCESS-WAIT INDEX1 SYSTEM-STATE))
         (AND (RIGHT-CHANNEL-FULL INDEX1 N SYSTEM-STATE)
              (PROCESS-NON-CRITICAL
                INDEX1 SYSTEM-STATE))))
```

Theorem 25: LEFT-FULL-NON-CRITICAL-WAIT-TO-NON-CRITICAL-OR-WAIT-WAIT-1 (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (LEFT-FULL-NON-CRITICAL-WAIT
                INDEX1 INDEX2 N SYSTEM-STATE)
              (RANDOM-BOOLEAN CLOCK))
         (AND (LEFT-CHANNEL-FULL INDEX1
                                 (ME INDEX1
                                     N
                                     CLOCK
                                     SYSTEM-STATE))
              (PROCESS-WAIT INDEX1
                (ME INDEX1
                    N
                    CLOCK
                    SYSTEM-STATE))))

((DISABLE MUTUAL-EXCLUSIONP))
```

Theorem 26: LEFT-FULL-NON-CRITICAL-WAIT-TO-NON-CRITICAL-OR-WAIT-WAIT-2 (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
```

```
                 (LESSP INDEX1 N)
                 (NUMBERP INDEX2)
                 (LESSP INDEX2 N)
                 (NOT (EQUAL INDEX1 INDEX2))
                 (LEFT-FULL-NON-CRITICAL-WAIT
                   INDEX1 INDEX2 N SYSTEM-STATE)
                 (NOT (RANDOM-BOOLEAN CLOCK)))
            (AND (RIGHT-CHANNEL-FULL INDEX1 N
                                     (ME INDEX1
                                         N
                                         CLOCK
                                         SYSTEM-STATE))
                 (PROCESS-NON-CRITICAL INDEX1
                                       (ME INDEX1
                                           N
                                           CLOCK
                                           SYSTEM-STATE))))

((DISABLE MUTUAL-EXCLUSIONP))
```

Theorem 27: LEFT-FULL-NON-CRITICAL-WAIT-TO-NON-CRITICAL-OR-WAIT-WAIT (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (LEFT-FULL-NON-CRITICAL-WAIT
                INDEX1 INDEX2 N SYSTEM-STATE))
         (NON-CRITICAL-OR-WAIT-WAIT (RANDOM-BOOLEAN CLOCK)
                                    INDEX1 INDEX2 N
                                    (ME INDEX1
                                        N
                                        CLOCK
                                        SYSTEM-STATE)))

((DISABLE ME PROCESS-NON-CRITICAL LEFT-CHANNEL-FULL
          RIGHT-CHANNEL-FULL MUTUAL-EXCLUSIONP))

(DISABLE LEFT-FULL-NON-CRITICAL-WAIT-                          (3)
TO-NON-CRITICAL-OR-WAIT-WAIT-2)
(DISABLE LEFT-FULL-NON-CRITICAL-WAIT-
TO-NON-CRITICAL-OR-WAIT-WAIT-1)
```

This progress statement says that a Non-Critical process with a full left channel will eventually become Wait with a full left channel, or pass a token to its right channel. Throughout this interval, some other process will remain Waiting.

Theorem 28: UNTIL-NON-CRITICAL-OR-WAIT-WAIT (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
```

```
               (LESSP INDEX2 N)
               (NOT (EQUAL INDEX1 INDEX2))
Flag is here, because we wish to use this lemma when
the first argument to Non-Critical-Or-Wait-Wait is
T or F, and not some Random-Boolean.
               (EQUAL FLAG (RANDOM-BOOLEAN
                              (NEXT-CLOCK (LIST 'ME INDEX1)
                                          CLOCK))))
          (IMPLIES (LEFT-FULL-NON-CRITICAL-WAIT
                     INDEX1 INDEX2 N
                     SYSTEM-STATE)
                   (NON-CRITICAL-OR-WAIT-WAIT
                     FLAG INDEX1 INDEX2 N
                     (INT (MAKE-PROCESS-LIST
                            (SYSTEM-DESCRIPTION N))
                          CLOCK
                          (ADD1 (NEXT-CLOCK
                                  (LIST 'ME INDEX1)
                                  CLOCK))
                          SYSTEM-STATE))))

((DISABLE ME LEFT-FULL-NON-CRITICAL-WAIT
          NON-CRITICAL-OR-WAIT-WAIT)
Disable the family specifications (Me) and every predicate
used in the theorem.  Enable the rewrite rule
Equal-Lists-Of-Length-Two.
 (ENABLE EQUAL-LISTS-OF-LENGTH-TWO)
Use the General-Progress Proof Rule, by instantiating
appropriately.
 (USE (GENERAL-PROGRESS
        (INTERMEDIATE-PROPERTY 'LEFT-FULL-NON-CRITICAL-WAIT)
        (FINAL-PROPERTY 'NON-CRITICAL-OR-WAIT-WAIT)
        (I-CODE (LIST INDEX1 INDEX2 N))
        (F-CODE (LIST (RANDOM-BOOLEAN
                        (NEXT-CLOCK (LIST 'ME INDEX1)
                                    CLOCK))
                      INDEX1
                      INDEX2
                      N))
        (CLOCK CLOCK)
        (MAX-CLOCK (ADD1 (NEXT-CLOCK (LIST 'ME INDEX1)
                                     CLOCK)))
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N))))
Use the Choose-Fairly-Returns-Member-1 theorem,
since every process but one will one during the
stable interval.
Instantiate Clock with the skolemized bad value.
      (CHOOSE-FAIRLY-RETURNS-MEMBER-1
        (CLOCK CLOCK)
        (TIME (BAD-STATE 'LEFT-FULL-NON-CRITICAL-WAIT
                         (LIST INDEX1 INDEX2 N)
                         (MAKE-PROCESS-LIST
                           (SYSTEM-DESCRIPTION N))
                         CLOCK
                         (NEXT-CLOCK
                           (LIST 'ME INDEX1)
```

```
                                         CLOCK)
                                       SYSTEM-STATE))
                 (PROCESS-DESCRIPTION (LIST 'ME INDEX1))
                 (PROCESS-LIST (MAKE-PROCESS-LIST
                                  (SYSTEM-DESCRIPTION N))))))

(DISABLE LEFT-FULL-NON-CRITICAL-WAIT-                                (4)
TO-NON-CRITICAL-OR-WAIT-WAIT)
(DISABLE LEFT-FULL-NON-CRITICAL-WAIT-IS-STABLE)
```

Is the Index1'th channel full, the Index1'th process Waiting, the Index2'th process Waiting, and the system satisfiying mutual exclusion?

Definition 29: (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N SYSTEM-STATE) =

```
(AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
     (LEFT-CHANNEL-FULL INDEX1 SYSTEM-STATE)
     (PROCESS-WAIT INDEX1 SYSTEM-STATE)
     (PROCESS-WAIT INDEX2 SYSTEM-STATE))
```

Theorem 30: LEFT-FULL-WAIT-WAIT-IS-STABLE-1 (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
              (NUMBERP CINDEX)
              (LESSP CINDEX N)
              (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                                   SYSTEM-STATE))
         (AND (LEFT-CHANNEL-FULL INDEX1
                                 (ME CINDEX
                                     N
                                     CLOCK
                                     SYSTEM-STATE))
              (PROCESS-WAIT INDEX1
                (ME CINDEX
                    N
                    CLOCK
                    SYSTEM-STATE))))

((DISABLE MUTUAL-EXCLUSIONP))
```

Theorem 31: LEFT-FULL-WAIT-WAIT-IS-STABLE-2 (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
              (NUMBERP CINDEX)
              (LESSP CINDEX N)
              (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                                   SYSTEM-STATE))
```

```
          (PROCESS-WAIT INDEX2
            (ME CINDEX
                N
                CLOCK
                SYSTEM-STATE)))

((DISABLE ME MUTUAL-EXCLUSIONP)
 (USE (ONE-CHANNEL-FULL-ALL-OTHERS-EMPTY
        (I INDEX1) (J INDEX2))))
```

Theorem 32: LEFT-FULL-WAIT-WAIT-IS-STABLE (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
              (NUMBERP CINDEX)
              (LESSP CINDEX N)
              (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                                   SYSTEM-STATE))
         (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                              (ME CINDEX
                                  N
                                  CLOCK
                                  SYSTEM-STATE)))

((DISABLE ME MUTUAL-EXCLUSIONP
          PROCESS-WAIT LEFT-CHANNEL-FULL))

(DISABLE LEFT-FULL-WAIT-WAIT-IS-STABLE-2)                        (5)
(DISABLE LEFT-FULL-WAIT-WAIT-IS-STABLE-1)
```

Is the Index2'th process Waiting, and the Index1'th process critical with critical counter Ticks, while the system is in mutual exclusion?

Definition 33: (CRITICAL-WAIT TICKS INDEX1 INDEX2 N SYSTEM-STATE) =

```
(AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
     (PROCESS-WAIT INDEX2 SYSTEM-STATE)
     (PROCESS-CRITICAL-WITH-TICKS
       TICKS INDEX1 SYSTEM-STATE))
```

Theorem 34: LEFT-FULL-WAIT-WAIT-TO-CRITICAL-WAIT-1 (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                                   SYSTEM-STATE))
         (PROCESS-WAIT INDEX2
           (ME INDEX1
               N
```

```
                 CLOCK
                 SYSTEM-STATE)))

((DISABLE ME MUTUAL-EXCLUSIONP))
```

Theorem 35: LEFT-FULL-WAIT-WAIT-TO-CRITICAL-WAIT-2 (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                                   SYSTEM-STATE))
         (PROCESS-CRITICAL-WITH-TICKS
           (RANDOM-NUMBER CLOCK) INDEX1
           (ME INDEX1
               N
               CLOCK
               SYSTEM-STATE)))

((DISABLE MUTUAL-EXCLUSIONP))
```

Theorem 36: LEFT-FULL-WAIT-WAIT-TO-CRITICAL-WAIT (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                                   SYSTEM-STATE))
         (CRITICAL-WAIT (RANDOM-NUMBER CLOCK)
                        INDEX1
                        INDEX2
                        N
                        (ME INDEX1
                            N
                            CLOCK
                            SYSTEM-STATE)))

((DISABLE MUTUAL-EXCLUSIONP PROCESS-WAIT
          PROCESS-CRITICAL-WITH-TICKS ME))

(DISABLE LEFT-FULL-WAIT-WAIT-TO-CRITICAL-WAIT-2)                    (6)
(DISABLE LEFT-FULL-WAIT-WAIT-TO-CRITICAL-WAIT-1)
```

The progress statement says that a Waiting process with a full left channel, will become Critical, with a particular critical counter, upon its first transition. Throughout the interval, some other process will remain Waiting.

Theorem 37: UNTIL-CRITICAL-WAIT (REWRITE)

```
(IMPLIES (AND (LESSP INDEX1 N)
              (NUMBERP INDEX1)
```

```
                (LESSP INDEX2 N)
                (NUMBERP INDEX2)
                (NOT (EQUAL INDEX1 INDEX2)))
           (IMPLIES (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                                          SYSTEM-STATE)
                    (CRITICAL-WAIT
                      (RANDOM-NUMBER
                        (NEXT-CLOCK (LIST 'ME INDEX1)
                                    CLOCK))
                      INDEX1
                      INDEX2
                      N
                      (INT (MAKE-PROCESS-LIST
                              (SYSTEM-DESCRIPTION N))
                           CLOCK
                           (ADD1 (NEXT-CLOCK (LIST 'ME INDEX1)
                                             CLOCK))
                           SYSTEM-STATE))))

((DISABLE ME LEFT-FULL-WAIT-WAIT CRITICAL-WAIT)
 (ENABLE EQUAL-LISTS-OF-LENGTH-TWO)
 (USE (GENERAL-PROGRESS
        (INTERMEDIATE-PROPERTY 'LEFT-FULL-WAIT-WAIT)
        (FINAL-PROPERTY 'CRITICAL-WAIT)
        (I-CODE (LIST INDEX1 INDEX2 N))
        (F-CODE (LIST (RANDOM-NUMBER
                        (NEXT-CLOCK (LIST 'ME INDEX1)
                                    CLOCK))
                      INDEX1
                      INDEX2
                      N))
        (CLOCK CLOCK)
        (MAX-CLOCK (ADD1 (NEXT-CLOCK (LIST 'ME INDEX1)
                                     CLOCK)))
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N))))
      (CHOOSE-FAIRLY-RETURNS-MEMBER-1
        (CLOCK CLOCK)
        (TIME (BAD-STATE 'LEFT-FULL-WAIT-WAIT
                         (LIST INDEX1 INDEX2 N)
                         (MAKE-PROCESS-LIST
                           (SYSTEM-DESCRIPTION N))
                         CLOCK
                         (NEXT-CLOCK (LIST 'ME INDEX1)
                                     CLOCK)
                         SYSTEM-STATE))
        (PROCESS-DESCRIPTION (LIST 'ME INDEX1))
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N))))))

(DISABLE LEFT-FULL-WAIT-WAIT-TO-CRITICAL-WAIT)                   (7)
(DISABLE LEFT-FULL-WAIT-WAIT-IS-STABLE)
```

Theorem 38: CRITICAL-WAIT-IS-STABLE-1 (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
              (NUMBERP CINDEX)
```

```
                  (LESSP CINDEX N)
                  (NUMBERP INDEX1)
                  (LESSP INDEX1 N)
                  (NUMBERP INDEX2)
                  (LESSP INDEX2 N)
                  (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                                 SYSTEM-STATE))
             (PROCESS-WAIT INDEX2
               (ME CINDEX
                   N
                   CLOCK
                   SYSTEM-STATE)))

    ((DISABLE ME MUTUAL-EXCLUSIONP)
     (USE (ONE-PROCESS-CRITICAL-ALL-CHANNELS-EMPTY
            (I INDEX1) (J INDEX2))))
```

Theorem 39: CRITICAL-WAIT-IS-STABLE-2 (REWRITE)

```
    (IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
                  (NUMBERP CINDEX)
                  (LESSP CINDEX N)
                  (NUMBERP INDEX1)
                  (LESSP INDEX1 N)
                  (NUMBERP INDEX2)
                  (LESSP INDEX2 N)
                  (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                                 SYSTEM-STATE))
             (PROCESS-CRITICAL-WITH-TICKS TICKS INDEX1
                                          (ME CINDEX
                                              N
                                              CLOCK
                                              SYSTEM-STATE)))
```

I cannot figure out why the rewriter will not use Critical-Wait-is-Stable-1. Break-Lemma seems to reveal that the proper substitution was found!

Theorem 40: CRITICAL-WAIT-IS-STABLE (REWRITE)

```
    (IMPLIES (AND (NOT (EQUAL CINDEX INDEX1))
                  (NUMBERP CINDEX)
                  (LESSP CINDEX N)
                  (NUMBERP INDEX1)
                  (LESSP INDEX1 N)
                  (NUMBERP INDEX2)
                  (LESSP INDEX2 N)
                  (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                                 SYSTEM-STATE))
             (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                            (ME CINDEX
                                N
                                CLOCK
                                SYSTEM-STATE)))

    ((DISABLE ME PROCESS-WAIT
```

```
         PROCESS-CRITICAL-WITH-TICKS MUTUAL-EXCLUSIONP)
 (USE (CRITICAL-WAIT-IS-STABLE-1)))

(DISABLE CRITICAL-WAIT-IS-STABLE-2)                             (8)
(DISABLE CRITICAL-WAIT-IS-STABLE-1)
```

Did the Index'th process follow the correct transition, given that it was Critical with Ticks on its critical counter

Definition 41: (CRITICAL-OR-NON-CRITICAL TICKS INDEX N SYSTEM-STATE) =

```
(IF (ZEROP TICKS)
    (RIGHT-CHANNEL-FULL INDEX N
                        SYSTEM-STATE)
    (PROCESS-CRITICAL-WITH-TICKS (SUB1 TICKS) INDEX
                                 SYSTEM-STATE))
```

Does Critical-Or-Non-Critical hold, along with mutual exclusion, and the Index2'th process Waiting?

Definition 42: (CRITICAL-OR-NON-CRITICAL-WAIT TICKS INDEX1 INDEX2 N SYSTEM-STATE) =

```
(AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
     (PROCESS-WAIT INDEX2 SYSTEM-STATE)
     (CRITICAL-OR-NON-CRITICAL TICKS INDEX1 N
                               SYSTEM-STATE))
```

Theorem 43: CRITICAL-WAIT-TO-CRITICAL-OR-NON-CRITICAL-WAIT-1 (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                             SYSTEM-STATE))
         (PROCESS-WAIT INDEX2
           (ME INDEX1
               N
               CLOCK
               SYSTEM-STATE)))

((DISABLE ME MUTUAL-EXCLUSIONP))
```

Theorem 44: CRITICAL-WAIT-TO-CRITICAL-OR-NON-CRITICAL-WAIT-2 (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                             SYSTEM-STATE))
         (CRITICAL-OR-NON-CRITICAL TICKS INDEX1 N
                                   (ME INDEX1
                                       N
```

```
                                                  CLOCK
                                                  SYSTEM-STATE)))

((DISABLE MUTUAL-EXCLUSIONP))
```

Theorem 45: CRITICAL-WAIT-TO-CRITICAL-OR-NON-CRITICAL-WAIT (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                             SYSTEM-STATE))
         (CRITICAL-OR-NON-CRITICAL-WAIT
           TICKS INDEX1 INDEX2 N
           (ME INDEX1
               N
               CLOCK
               SYSTEM-STATE)))

((DISABLE PROCESS-CRITICAL-WITH-TICKS
          CRITICAL-OR-NON-CRITICAL
          ME PROCESS-WAIT
          MUTUAL-EXCLUSIONP)
 (USE (CRITICAL-WAIT-TO-CRITICAL-OR-NON-CRITICAL-WAIT-1)))

(DISABLE CRITICAL-WAIT-TO-CRITICAL-OR-NON-CRITICAL-WAIT-2)         (9)
(DISABLE CRITICAL-WAIT-TO-CRITICAL-OR-NON-CRITICAL-WAIT-1)
```

This progress statements says that a Critical process will either remain Critical, when it finally runs, or will release a token upon its right channel. Throughout the interval, some other process will remain Waiting.

Theorem 46: UNTIL-CRITICAL-OR-NON-CRITICAL-WAIT (REWRITE)

```
(IMPLIES (AND (LESSP INDEX1 N)
              (NUMBERP INDEX1)
              (LESSP INDEX2 N)
              (NUMBERP INDEX2)
              (NOT (EQUAL INDEX1 INDEX2)))
         (IMPLIES (CRITICAL-WAIT
                    TICKS INDEX1 INDEX2 N
                    SYSTEM-STATE)
                  (CRITICAL-OR-NON-CRITICAL-WAIT
                    TICKS INDEX1 INDEX2 N
                    (INT (MAKE-PROCESS-LIST
                           (SYSTEM-DESCRIPTION N))
                         CLOCK
                         (ADD1 (NEXT-CLOCK
                                 (LIST 'ME INDEX1)
                                 CLOCK))
                         SYSTEM-STATE))))

((DISABLE ME CRITICAL-WAIT CRITICAL-OR-NON-CRITICAL-WAIT)
 (ENABLE EQUAL-LISTS-OF-LENGTH-TWO))
```

```
 (USE (GENERAL-PROGRESS
        (INTERMEDIATE-PROPERTY 'CRITICAL-WAIT)
        (FINAL-PROPERTY 'CRITICAL-OR-NON-CRITICAL-WAIT)
        (I-CODE (LIST TICKS INDEX1 INDEX2 N))
        (F-CODE (LIST TICKS INDEX1 INDEX2 N))
        (CLOCK CLOCK)
        (MAX-CLOCK (ADD1 (NEXT-CLOCK
                           (LIST 'ME INDEX1)
                           CLOCK)))
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N))))
      (CHOOSE-FAIRLY-RETURNS-MEMBER-1
        (CLOCK CLOCK)
        (TIME (BAD-STATE 'CRITICAL-WAIT
                         (LIST TICKS INDEX1 INDEX2 N)
                         (MAKE-PROCESS-LIST
                           (SYSTEM-DESCRIPTION N))
                         CLOCK
                         (NEXT-CLOCK (LIST 'ME INDEX1)
                                     CLOCK)
                         SYSTEM-STATE))
        (PROCESS-DESCRIPTION (LIST 'ME INDEX1))
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N))))))

(DISABLE CRITICAL-WAIT-TO-CRITICAL-OR-NON-CRITICAL-WAIT)        (10)
(DISABLE CRITICAL-WAIT-IS-STABLE)
```

Definition 47: (LEFT-CHANNEL-FULL-PROCESS-WAIT INDEX SYSTEM-STATE) =

```
(AND (LEFT-CHANNEL-FULL INDEX SYSTEM-STATE)
       (PROCESS-WAIT INDEX SYSTEM-STATE))
```

Theorem 48: LEFT-CHANNEL-FULL-PROCESS-WAIT-IS-STABLE (REWRITE)

```
(IMPLIES (AND (NOT (EQUAL CINDEX INDEX))
              (LESSP CINDEX N)
              (NUMBERP CINDEX)
              (LESSP INDEX N)
              (NUMBERP INDEX)
              (LEFT-CHANNEL-FULL-PROCESS-WAIT
                INDEX
                SYSTEM-STATE))
         (LEFT-CHANNEL-FULL-PROCESS-WAIT
           INDEX
           (ME CINDEX
               N
               CLOCK
               SYSTEM-STATE)))
```

Theorem 49: LEFT-CHANNEL-FULL-PROCESS-WAIT-TO-PROCESS-CRITICAL (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX)
              (LESSP INDEX N)
              (LEFT-CHANNEL-FULL-PROCESS-WAIT
                INDEX
                SYSTEM-STATE))
         (PROCESS-CRITICAL-WITH-TICKS (RANDOM-NUMBER CLOCK)
```

```
                                             INDEX
                                             (ME INDEX
                                                 N
                                                 CLOCK
                                                 SYSTEM-STATE)))
```

This is the simplest progress statement. It says that a Waiting process with a full left channel will become Critical when it finally runs.

Theorem 50: UNTIL-PROCESS-CRITICAL (REWRITE)

```
(IMPLIES (AND (LESSP INDEX N)
              (NUMBERP INDEX))
         (IMPLIES (LEFT-CHANNEL-FULL-PROCESS-WAIT
                    INDEX
                    SYSTEM-STATE)
                  (PROCESS-CRITICAL-WITH-TICKS
                    (RANDOM-NUMBER (NEXT-CLOCK
                                     (LIST 'ME INDEX)
                                     CLOCK))
                    INDEX
                    (INT (MAKE-PROCESS-LIST
                           (SYSTEM-DESCRIPTION N))
                         CLOCK
                         (ADD1 (NEXT-CLOCK
                                 (LIST 'ME INDEX) CLOCK))
                         SYSTEM-STATE))))

((DISABLE ME LEFT-CHANNEL-FULL-PROCESS-WAIT
          PROCESS-CRITICAL-WITH-TICKS)
 (ENABLE EQUAL-LISTS-OF-LENGTH-TWO)
 (USE (GENERAL-PROGRESS
        (INTERMEDIATE-PROPERTY
          'LEFT-CHANNEL-FULL-PROCESS-WAIT)
        (FINAL-PROPERTY 'PROCESS-CRITICAL-WITH-TICKS)
        (I-CODE (LIST INDEX))
        (F-CODE (LIST (RANDOM-NUMBER
                        (NEXT-CLOCK (LIST 'ME INDEX)
                                    CLOCK))
                      INDEX))
        (CLOCK CLOCK)
        (MAX-CLOCK (ADD1 (NEXT-CLOCK
                           (LIST 'ME INDEX) CLOCK)))
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N))))
      (CHOOSE-FAIRLY-RETURNS-MEMBER-1
        (CLOCK CLOCK)
        (TIME (BAD-STATE 'LEFT-CHANNEL-FULL-PROCESS-WAIT
                         (LIST INDEX)
                         (MAKE-PROCESS-LIST
                           (SYSTEM-DESCRIPTION N))
                         CLOCK
                         (NEXT-CLOCK (LIST 'ME INDEX)
                                     CLOCK)
                         SYSTEM-STATE))
```

```
        (PROCESS-DESCRIPTION (LIST 'ME INDEX))
        (PROCESS-LIST (MAKE-PROCESS-LIST
                        (SYSTEM-DESCRIPTION N))))))

(DISABLE LEFT-CHANNEL-FULL-PROCESS-WAIT-TO-PROCESS-CRITICAL)    (11)
(DISABLE LEFT-CHANNEL-FULL-PROCESS-WAIT-IS-STABLE)
```

We have proved four progress statement. We now combine them by moving the token along the ring to the Waiting processes left channel, and then, using the fourth progres statement, have the Waiting process become Critical.

We build the clock function that "predicts" the time when the Waiting process becomes Critical.

The Clock at which a Critical process releases a token on its right channel. The process must run Ticks+1 times.

Definition 51: (CLOCK-OF-CRITICAL-TO-NON-CRITICAL INDEX CLOCK TICKS) =

```
(IF (ZEROP TICKS)
    (ADD1 (NEXT-CLOCK (LIST 'ME INDEX) CLOCK))
    (ADD1 (NEXT-CLOCK
            (LIST 'ME INDEX)
            (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
              INDEX CLOCK (SUB1 TICKS))))))
```

Theorem 52: CLOCK-OF-CRITICAL-TO-NON-CRITICAL-LARGER (REWRITE)

```
(LESSP CLOCK
       (CLOCK-OF-CRITICAL-TO-NON-CRITICAL INDEX
                                          CLOCK
                                          TICKS))
```

Theorem 53: CLOCK-OF-CRITICAL-TO-NON-CRITICAL-CANNONICAL (REWRITE)

```
(IMPLIES (NOT (ZEROP TICKS))
         (EQUAL (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
                  INDEX
                  (ADD1 (NEXT-CLOCK (LIST 'ME INDEX)
                                    CLOCK))
                  (SUB1 TICKS))
                (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
                  INDEX
                  CLOCK
                  TICKS)))
```

Theorem 54: CRITICAL-WAIT-IMPLIED-BY-CRITICAL-OR -NON-CRITICAL-WAIT-NOT-ZERO-TICKS (REWRITE)

```
(IMPLIES (AND (NOT (ZEROP TICKS))
              (CRITICAL-OR-NON-CRITICAL-WAIT
                TICKS INDEX1 INDEX2 N
```

```
                  SYSTEM-STATE))
           (CRITICAL-WAIT (SUB1 TICKS) INDEX1 INDEX2 N
                          SYSTEM-STATE))
```

Theorem 55: CRITICAL-WAIT-FIXES-TICKS (REWRITE)

```
(IMPLIES (NOT (NUMBERP TICKS))
         (EQUAL (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                               SYSTEM-STATE)
                (CRITICAL-WAIT 0 INDEX1 INDEX2 N
                               SYSTEM-STATE)))
```

Does mutual exclusion hold? Is the Index1'th right channel full? Is the Index2'th process Waiting?

Definition 56: (RIGHT-CHANNEL-FULL-WAIT INDEX1 INDEX2 N SYSTEM-STATE) =

```
(AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
     (RIGHT-CHANNEL-FULL INDEX1 N SYSTEM-STATE)
     (PROCESS-WAIT INDEX2 SYSTEM-STATE))
```

Theorem 57: RIGHT-CHANNEL-FULL-WAIT-IMPLIED-BY-CRITICAL-OR -NON-CRITICAL-WAIT-ZERO-TICKS (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (CRITICAL-OR-NON-CRITICAL-WAIT
                0 INDEX1 INDEX2 N
                SYSTEM-STATE))
         (RIGHT-CHANNEL-FULL-WAIT INDEX1 INDEX2 N
                                  SYSTEM-STATE))
```

Definition 58: (CRITICAL-TO-NON-CRITICAL-INDUCTION-SCHEME N INDEX CLOCK TICKS SYSTEM-STATE) =

```
(IF (ZEROP TICKS)
    T
    (CRITICAL-TO-NON-CRITICAL-INDUCTION-SCHEME
      N INDEX
      (ADD1 (NEXT-CLOCK (LIST 'ME INDEX) CLOCK))
      (SUB1 TICKS)
      (INT (MAKE-PROCESS-LIST (SYSTEM-DESCRIPTION N))
           CLOCK
           (ADD1 (NEXT-CLOCK (LIST 'ME INDEX) CLOCK))
           SYSTEM-STATE)))
```

This lemma proves that Clock-Of-Critical-To-Non-Critical works. It inducts on Ticks, using Until-Critical-Or-Non-Critical-Wait progress statement repeatedly. Also, another Waiting process remains Waiting.

Theorem 59: CRITICAL-TO-NON-CRITICAL (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
```

```
                 (LESSP INDEX2 N)
                 (NOT (EQUAL INDEX1 INDEX2))
                 (CRITICAL-WAIT TICKS INDEX1 INDEX2 N
                               SYSTEM-STATE))
            (RIGHT-CHANNEL-FULL-WAIT
              INDEX1 INDEX2 N
              (INT (MAKE-PROCESS-LIST
                     (SYSTEM-DESCRIPTION N))
                   CLOCK
                   (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
                     INDEX1
                     CLOCK
                     TICKS)
                   SYSTEM-STATE)))

((INDUCT (CRITICAL-TO-NON-CRITICAL-INDUCTION-SCHEME
           N INDEX1 CLOCK TICKS SYSTEM-STATE))
 (DISABLE CRITICAL-WAIT RIGHT-CHANNEL-FULL-WAIT
          CRITICAL-OR-NON-CRITICAL-WAIT))

(DISABLE RIGHT-CHANNEL-FULL-WAIT-IMPLIED-BY-                    (12)
CRITICAL-OR-NON-CRITICAL-WAIT-ZERO-TICKS)
(DISABLE CRITICAL-WAIT-FIXES-TICKS)
(DISABLE CRITICAL-WAIT-IMPLIED-BY-CRITICAL-
OR-NON-CRITICAL-WAIT-NOT-ZERO-TICKS)
(DISABLE CLOCK-OF-CRITICAL-TO-NON-CRITICAL-CANNONICAL)
```

Now we can move the token from a Critical process to its right channel, and another process remains Waiting.

The Clock value when a Waiting process with a full left channel will release a token on its right channel. Simply let it become Critical then use Clock-Of-Critical-To-Non-Critical.

Definition 60: (CLOCK-OF-WAIT-TO-NON-CRITICAL INDEX CLOCK) =

```
(CLOCK-OF-CRITICAL-TO-NON-CRITICAL
  INDEX
  (ADD1 (NEXT-CLOCK (LIST 'ME INDEX) CLOCK))
  (RANDOM-NUMBER (NEXT-CLOCK (LIST 'ME INDEX) CLOCK)))
```

Prove that Clock-Of-Wait-To-Non-Critical-Works, and that another process remains Waiting during that interval. Uses the progress statement Until-Critical-Wait.

Theorem 61: WAIT-TO-NON-CRITICAL (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                                   SYSTEM-STATE))
```

```
          (RIGHT-CHANNEL-FULL-WAIT
            INDEX1 INDEX2 N
            (INT (MAKE-PROCESS-LIST (SYSTEM-DESCRIPTION N))
                 CLOCK
                 (CLOCK-OF-WAIT-TO-NON-CRITICAL
                   INDEX1
                   CLOCK)
                 SYSTEM-STATE)))

((DISABLE LEFT-FULL-WAIT-WAIT RIGHT-CHANNEL-FULL-WAIT
          CRITICAL-WAIT)
 (USE (CRITICAL-TO-NON-CRITICAL
        (CLOCK (ADD1 (NEXT-CLOCK (LIST 'ME INDEX1) CLOCK)))
        (SYSTEM-STATE (INT (MAKE-PROCESS-LIST
                              (SYSTEM-DESCRIPTION N))
                           CLOCK
                           (ADD1 (NEXT-CLOCK
                                   (LIST 'ME INDEX1)
                                   CLOCK))
                           SYSTEM-STATE))
        (TICKS (RANDOM-NUMBER
                 (NEXT-CLOCK (LIST 'ME INDEX1) CLOCK))))))
```

Theorem 62: CLOCK-OF-WAIT-TO-NON-CRITICAL-LARGER (REWRITE)

```
(LESSP CLOCK
       (CLOCK-OF-WAIT-TO-NON-CRITICAL INDEX
                                      CLOCK))
```

Now we can move the token from the left channel of a Waiting process to its right channel, while another process remains Waiting.

Compute the Clock value that a Non-Critical with a full left channel will release a token upon its right channel.

Definition 63: (CLOCK-OF-NON-CRITICAL-TO-NON-CRITICAL INDEX CLOCK) =

```
(IF (RANDOM-BOOLEAN (NEXT-CLOCK (LIST 'ME INDEX) CLOCK))
    (CLOCK-OF-WAIT-TO-NON-CRITICAL
      INDEX (ADD1 (NEXT-CLOCK (LIST 'ME INDEX) CLOCK)))
    (ADD1 (NEXT-CLOCK (LIST 'ME INDEX) CLOCK)))
```

Theorem 64: CLOCK-OF-NON-CRITICAL-TO-NON-CRITICAL-LARGER (REWRITE)

```
(LESSP CLOCK
       (CLOCK-OF-NON-CRITICAL-TO-NON-CRITICAL INDEX
                                              CLOCK))
```

Theorem 65: LEFT-FULL-WAIT-WAIT-IMPLIED-BY-NON-CRITICAL-OR-WAIT-WAIT (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
```

```
               (NON-CRITICAL-OR-WAIT-WAIT T INDEX1 INDEX2 N
                                           SYSTEM-STATE))
          (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2 N
                               SYSTEM-STATE))
```

Theorem 66: RIGHT-CHANNEL-FULL-WAIT-IMPLIED-BY-NON-CRITICAL -OR-WAIT-WAIT (REWRITE)

```
(IMPLIES (AND (LESSP INDEX1 N)
              (NUMBERP INDEX1)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (NON-CRITICAL-OR-WAIT-WAIT F INDEX1 INDEX2 N
                                          SYSTEM-STATE))
         (RIGHT-CHANNEL-FULL-WAIT INDEX1 INDEX2 N
                                  SYSTEM-STATE))
```

Prove that Clock-Of-Non-Critical-To-Non-Critical works, and that another process remains Waiting in the interval. Uses progress statement Until-Non-Critical-Or-Wait-Wait.

Theorem 67: NON-CRITICAL-TO-NON-CRITICAL (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (NOT (EQUAL INDEX1 INDEX2))
              (LEFT-FULL-NON-CRITICAL-WAIT INDEX1 INDEX2 N
                                            SYSTEM-STATE))
         (RIGHT-CHANNEL-FULL-WAIT
           INDEX1 INDEX2 N
           (INT (MAKE-PROCESS-LIST
                  (SYSTEM-DESCRIPTION N))
                CLOCK
                (CLOCK-OF-NON-CRITICAL-TO-NON-CRITICAL
                  INDEX1 CLOCK)
                SYSTEM-STATE)))

((DISABLE CLOCK-OF-WAIT-TO-NON-CRITICAL
          LEFT-FULL-NON-CRITICAL-WAIT
          RIGHT-CHANNEL-FULL-WAIT
          LEFT-FULL-WAIT-WAIT
          NON-CRITICAL-OR-WAIT-WAIT)
 (USE (WAIT-TO-NON-CRITICAL
        (CLOCK (ADD1 (NEXT-CLOCK (LIST 'ME INDEX1) CLOCK)))
        (SYSTEM-STATE (INT (MAKE-PROCESS-LIST
                              (SYSTEM-DESCRIPTION N))
                            CLOCK
                            (ADD1 (NEXT-CLOCK
                                    (LIST 'ME INDEX1)
                                    CLOCK))
                            SYSTEM-STATE)))))

(DISABLE RIGHT-CHANNEL-FULL-WAIT-IMPLIED-BY                      (13)
```

```
-NON-CRITICAL-OR-WAIT-WAIT)

(DISABLE LEFT-FULL-WAIT-WAIT-IMPLIED-BY
-NON-CRITICAL-OR-WAIT-WAIT)
```

Now that we can move a token from a left channel to a right channel, we can move the token along the ring to the Waiting process's left channel.

Move the token from the Index1'th channel to the Index2'th channel Since the token is in the left channel, the next process is either Waiting or Non-Critical.

Definition 68: (DEFN CLOCK-OF-TOKEN-MOVING INDEX1 INDEX2 N CLOCK SYSTEM-STATE) =

```
(IF (AND (NUMBERP INDEX1)
         (LESSP INDEX1 N)
         (NUMBERP INDEX2)
         (LESSP INDEX2 N))
    (IF (EQUAL INDEX1 INDEX2)
        CLOCK
        (IF (PROCESS-NON-CRITICAL
              INDEX1 SYSTEM-STATE)
            (CLOCK-OF-TOKEN-MOVING
              (ADD1-MOD N INDEX1)
              INDEX2
              N
              (CLOCK-OF-NON-CRITICAL-TO-NON-CRITICAL
                INDEX1 CLOCK)
              (INT (MAKE-PROCESS-LIST
                     (SYSTEM-DESCRIPTION N))
                   CLOCK
                   (CLOCK-OF-NON-CRITICAL-TO-NON-CRITICAL
                     INDEX1 CLOCK)
                   SYSTEM-STATE))
            (CLOCK-OF-TOKEN-MOVING
              (ADD1-MOD N INDEX1)
              INDEX2
              N
              (CLOCK-OF-WAIT-TO-NON-CRITICAL
                INDEX1 CLOCK)
              (INT (MAKE-PROCESS-LIST
                     (SYSTEM-DESCRIPTION N))
                   CLOCK
                   (CLOCK-OF-WAIT-TO-NON-CRITICAL
                     INDEX1 CLOCK)
                   SYSTEM-STATE))))
    CLOCK)

((LESSP (DIFFERENCE (IF (LESSP INDEX2 INDEX1)
                        (PLUS INDEX2 N)
                        INDEX2)
                    INDEX1)))
```

Theorem 69: CLOCK-OF-TOKEN-MOVING-LARGER (REWRITE)

```
(NOT (LESSP (CLOCK-OF-TOKEN-MOVING INDEX1 INDEX2 N
                                   CLOCK SYSTEM-STATE)
            CLOCK))
```

Is the Index1'th left channel fulll, the Index2'th process Waiting, and the system satisfying mutual exclusion?

Definition 70: (FULL-WAIT INDEX1 INDEX2 N SYSTEM-STATE) =

```
(AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
     (LEFT-CHANNEL-FULL INDEX1 SYSTEM-STATE)
     (PROCESS-WAIT INDEX2 SYSTEM-STATE))
```

Theorem 71: LEFT-FULL-NON-CRITICAL-WAIT-IMPLIED-BY (REWRITE)

```
(IMPLIES (AND (LESSP INDEX1 N)
              (NUMBERP INDEX1)
              (LESSP INDEX2 N)
              (NUMBERP INDEX2)
              (FULL-WAIT INDEX1 INDEX2 N SYSTEM-STATE)
              (PROCESS-NON-CRITICAL INDEX1 SYSTEM-STATE))
         (LEFT-FULL-NON-CRITICAL-WAIT INDEX1 INDEX2
                                      N SYSTEM-STATE))
```

Theorem 72: LEFT-FULL-WAIT-WAIT-IMPLIED-BY (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (FULL-WAIT INDEX1 INDEX2 N SYSTEM-STATE)
              (PROCESS-WAIT INDEX1 SYSTEM-STATE))
         (LEFT-FULL-WAIT-WAIT INDEX1 INDEX2
                              N SYSTEM-STATE))
```

Theorem 73: PROCESS-IS-NON-CRITICAL-OR-WAIT ()

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (NUMBERP INDEX)
              (LESSP INDEX N)
              (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (LEFT-CHANNEL-FULL INDEX SYSTEM-STATE))
         (OR (PROCESS-WAIT INDEX SYSTEM-STATE)
             (PROCESS-NON-CRITICAL INDEX SYSTEM-STATE)))

((ENABLE ABOUT-PLUS-EQUALS-ONE))
```

Theorem 74: PROCESS-WAIT-IMPLIED-BY (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NOT (PROCESS-NON-CRITICAL
                     INDEX1 SYSTEM-STATE))
              (FULL-WAIT INDEX1 INDEX2 N SYSTEM-STATE))
         (PROCESS-WAIT INDEX1 SYSTEM-STATE))

((USE (PROCESS-IS-NON-CRITICAL-OR-WAIT (INDEX INDEX1))))
```

Theorem 75: FULL-WAIT-IMPLIED-BY (REWRITE)

```
(IMPLIES (AND (LESSP INDEX1 N)
              (NUMBERP INDEX1)
              (RIGHT-CHANNEL-FULL-WAIT (SUB1-MOD N INDEX1)
                                       INDEX2 N
                                       SYSTEM-STATE))
         (FULL-WAIT INDEX1 INDEX2 N SYSTEM-STATE))
```

Clock-Of-Token-Moving moves the token from a left channel to the Waiting process's left channel.

Theorem 76: CLOCK-OF-TOKEN-MOVING-WORKS-FROM-WAIT-OR-NON-CRITICAL (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (FULL-WAIT INDEX1 INDEX2 N SYSTEM-STATE))
         (FULL-WAIT INDEX2 INDEX2 N
                    (INT (MAKE-PROCESS-LIST
                           (SYSTEM-DESCRIPTION N))
                         CLOCK
                         (CLOCK-OF-TOKEN-MOVING
                           INDEX1 INDEX2
                           N
                           CLOCK
                           SYSTEM-STATE)
                         SYSTEM-STATE)))

((DISABLE ADD1-MOD SUB1-MOD
          PROCESS-WAIT PROCESS-NON-CRITICAL
          RIGHT-CHANNEL-FULL-WAIT
          LEFT-FULL-NON-CRITICAL-WAIT
          LEFT-FULL-WAIT-WAIT
          FULL-WAIT
          CLOCK-OF-NON-CRITICAL-TO-NON-CRITICAL
          CLOCK-OF-WAIT-TO-NON-CRITICAL))
```

Theorem 77: NOT-EQUAL-IMPLIED-BY (REWRITE)

```
(IMPLIES (CRITICAL-WAIT TICKS INDEX1
                        INDEX2 N SYSTEM-STATE)
         (NOT (EQUAL INDEX1 INDEX2)))
```

Now we can move a token from a left channel to a left channel. All that remains is to move a token from a Critical process to a left channel.

By giving Clock-Of-Token-Moving the proper Clock argument, the token can be moved from a Critical process to a Waiting process's left channel.

Theorem 78: CLOCK-OF-TOKEN-MOVING-WORKS-FROM-CRITICAL (REWRITE)

```
(IMPLIES (AND (NUMBERP INDEX1)
              (LESSP INDEX1 N)
              (NUMBERP INDEX2)
              (LESSP INDEX2 N)
              (CRITICAL-WAIT TICKS INDEX1
                             INDEX2 N SYSTEM-STATE))
         (FULL-WAIT
           INDEX2 INDEX2 N
           (INT (MAKE-PROCESS-LIST
                  (SYSTEM-DESCRIPTION N))
                CLOCK
                (CLOCK-OF-TOKEN-MOVING
                  (ADD1-MOD N INDEX1) INDEX2
                  N
                  (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
                    INDEX1 CLOCK TICKS)
                  (INT (MAKE-PROCESS-LIST
                         (SYSTEM-DESCRIPTION N))
                       CLOCK
                       (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
                         INDEX1 CLOCK TICKS)
                       SYSTEM-STATE))
                SYSTEM-STATE)))

((DISABLE FULL-WAIT CRITICAL-WAIT ADD1-MOD SUB1-MOD)
 (USE (CLOCK-OF-TOKEN-MOVING-WORKS-FROM-WAIT-OR-NON-CRITICAL
        (INDEX1 (ADD1-MOD N INDEX1))
        (INDEX2 INDEX2)
        (CLOCK (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
                 INDEX1 CLOCK TICKS))
        (SYSTEM-STATE
          (INT (MAKE-PROCESS-LIST
                 (SYSTEM-DESCRIPTION N))
               CLOCK
               (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
                 INDEX1 CLOCK TICKS)
               SYSTEM-STATE)))))

(DISABLE NOT-EQUAL-IMPLIED-BY)                                  (14)
(DISABLE FULL-WAIT-IMPLIED-BY)
(DISABLE PROCESS-WAIT-IMPLIED-BY)
(DISABLE LEFT-FULL-WAIT-WAIT-IMPLIED-BY)
(DISABLE LEFT-FULL-NON-CRITICAL-WAIT-IMPLIED-BY)
```

Now we can tie loose ends together. Given a Waiting process, find the token in the ring, and compute the Clock required to move the token to the Waiting process's left channel.

If a process is Critical, return its Index. Else return F.

Definition 79: (FIND-TOKEN-IN-PROCESS N SYSTEM-STATE) =

```
(IF (ZEROP N)
    F
```

```
(IF (EQUAL (WEIGHT-OF-PROCESS (SUB1 N) SYSTEM-STATE)
           1)
    (SUB1 N)
    (FIND-TOKEN-IN-PROCESS (SUB1 N) SYSTEM-STATE)))
```

If a channel is full, returns its Index. Else return F.

Definition 80: (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE) =

```
(IF (ZEROP N)
    F
    (IF (LESSP 0
               (LENGTH (GET-STATE (CONS 'C (SUB1 N))
                                  SYSTEM-STATE)))
        (SUB1 N)
        (FIND-TOKEN-IN-CHANNEL (SUB1 N) SYSTEM-STATE)))
```

Theorem 81: PROCESS-CRITICAL-WITH-TICKS-EQUALS (REWRITE)

```
(EQUAL (PROCESS-CRITICAL-WITH-TICKS
         (CDR (ASSOC (CONS 'ME INDEX)
                     SYSTEM-STATE))
         INDEX SYSTEM-STATE)
       (EQUAL (WEIGHT-OF-PROCESS INDEX SYSTEM-STATE)
              1))
```

Prove that Find-Token-In... works (i.e., find the token). Prove that one and only one of Find-Token-In... is non-F, and that one is numeric.

Theorem 82: FIND-TOKEN-IN-PROCESS-WORKS (REWRITE)

```
(IMPLIES (FIND-TOKEN-IN-PROCESS N SYSTEM-STATE)
         (PROCESS-CRITICAL-WITH-TICKS
           (CDR (ASSOC (CONS 'ME
                             (FIND-TOKEN-IN-PROCESS
                               N SYSTEM-STATE))
                       SYSTEM-STATE))
           (FIND-TOKEN-IN-PROCESS N SYSTEM-STATE)
           SYSTEM-STATE))

((DISABLE WEIGHT-OF-PROCESS))
```

Theorem 83: FIND-TOKEN-IN-CHANNEL-WORKS (REWRITE)

```
(IMPLIES (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE)
         (LEFT-CHANNEL-FULL (FIND-TOKEN-IN-CHANNEL
                              N SYSTEM-STATE)
                            SYSTEM-STATE))
```

Theorem 84: FIND-TOKEN-IN-PROCESS-LESSP-N (REWRITE)

```
(EQUAL (LESSP (FIND-TOKEN-IN-PROCESS N SYSTEM-STATE) N)
       (LESSP 0 N))
```

Theorem 85: FIND-TOKEN-IN-CHANNEL-LESSP-N (REWRITE)

```
(EQUAL (LESSP (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE) N)
       (LESSP 0 N))
```

Theorem 86: FIND-TOKEN-IN-PROCESS-IMPLIED-BY (REWRITE)

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (NOT (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE)))
         (FIND-TOKEN-IN-PROCESS N SYSTEM-STATE))
```

Theorem 87: FIND-TOKEN-IN-PROCESS-NUMBERP-IMPLIED-BY (REWRITE)

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (NOT (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE)))
         (NUMBERP (FIND-TOKEN-IN-PROCESS N SYSTEM-STATE)))
```

The Clock of the token moving from where-ever it is to the Index'ed channel.

Definition 88: (CLOCK-OF-TOKEN-MOVING-TO-WAIT INDEX N CLOCK SYSTEM-STATE)
=

```
(IF (AND (LESSP INDEX N)
         (NUMBERP INDEX))
    (IF (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE)
        (CLOCK-OF-TOKEN-MOVING
          (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE)
          INDEX N
          CLOCK SYSTEM-STATE)
        (CLOCK-OF-TOKEN-MOVING
          (ADD1-MOD N (FIND-TOKEN-IN-PROCESS
                        N SYSTEM-STATE))
          INDEX
          N
          (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
            (FIND-TOKEN-IN-PROCESS N SYSTEM-STATE)
            CLOCK
            (GET-STATE (CONS 'ME
                             (FIND-TOKEN-IN-PROCESS
                               N SYSTEM-STATE))
                       SYSTEM-STATE))
          (INT (MAKE-PROCESS-LIST (SYSTEM-DESCRIPTION N))
               CLOCK
               (CLOCK-OF-CRITICAL-TO-NON-CRITICAL
                 (FIND-TOKEN-IN-PROCESS N SYSTEM-STATE)
                 CLOCK
                 (GET-STATE
                   (CONS 'ME
                         (FIND-TOKEN-IN-PROCESS
                           N SYSTEM-STATE))
                   SYSTEM-STATE))
               SYSTEM-STATE)))
    CLOCK)
```

Theorem 89: FULL-WAIT-IMPLIED-BY-1 (REWRITE)

```
(IMPLIES (AND (PROCESS-WAIT INDEX SYSTEM-STATE)
```

```
          (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
          (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE))
     (FULL-WAIT (FIND-TOKEN-IN-CHANNEL N SYSTEM-STATE)
                INDEX N
                SYSTEM-STATE))
```

Theorem 90: CRITICAL-WAIT-IMPLIED-BY (REWRITE)

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (PROCESS-WAIT INDEX2 SYSTEM-STATE)
              (PROCESS-CRITICAL-WITH-TICKS
                (CDR (ASSOC (CONS 'ME INDEX1)
                            SYSTEM-STATE))
                INDEX1 SYSTEM-STATE))
         (CRITICAL-WAIT (CDR (ASSOC (CONS 'ME INDEX1)
                                    SYSTEM-STATE))
                        INDEX1
                        INDEX2
                        N
                        SYSTEM-STATE))

((DISABLE PROCESS-WAIT
          PROCESS-CRITICAL-WITH-TICKS
          PROCESS-CRITICAL-WITH-TICKS-EQUALS
          MUTUAL-EXCLUSIONP))
```

Clock-Of-Token-Moving-to-Wait does move the token to the left channel of the Waiting process.

Theorem 91: CLOCK-OF-TOKEN-MOVING-TO-WAIT-WORKS (REWRITE)

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (NUMBERP INDEX)
              (LESSP INDEX N)
              (PROCESS-WAIT INDEX SYSTEM-STATE))
         (FULL-WAIT INDEX INDEX N
                    (INT (MAKE-PROCESS-LIST
                           (SYSTEM-DESCRIPTION N))
                         CLOCK
                         (CLOCK-OF-TOKEN-MOVING-TO-WAIT
                           INDEX N CLOCK SYSTEM-STATE)
                         SYSTEM-STATE)))

((DISABLE FULL-WAIT PROCESS-WAIT
          PROCESS-CRITICAL-WITH-TICKS CRITICAL-WAIT
          PROCESS-CRITICAL-WITH-TICKS-EQUALS
          MUTUAL-EXCLUSIONP
          ADD1-MOD SUB1-MOD))

(DISABLE CRITICAL-WAIT-IMPLIED-BY)                                   (15)
(DISABLE FULL-WAIT-IMPLIED-BY-1)
(DISABLE FIND-TOKEN-IN-PROCESS-NUMBERP-IMPLIED-BY)
(DISABLE FIND-TOKEN-IN-PROCESS-IMPLIED-BY)
(DISABLE FIND-TOKEN-IN-CHANNEL-LESSP-N)
(DISABLE FIND-TOKEN-IN-CHANNEL-WORKS)
(DISABLE FIND-TOKEN-IN-PROCESS-LESSP-N)
(DISABLE FIND-TOKEN-IN-PROCESS-WORKS)
```

```
(DISABLE PROCESS-CRITICAL-WITH-TICKS-EQUALS)
```

Theorem 92: CLOCK-OF-TOKEN-MOVING-TO-WAIT-LARGER (REWRITE)

```
(NOT (LESSP (CLOCK-OF-TOKEN-MOVING-TO-WAIT
              INDEX N CLOCK SYSTEM-STATE)
            CLOCK))
```

Once the token is moved to a Wating processes left channel, simply extend the interval until that process runs.

Definition 93: (CLOCK-OF-TOKEN-MOVING-TO-CRITICAL INDEX N CLOCK SYSTEM-STATE) =

```
(IF (AND (LESSP INDEX N)
         (NUMBERP INDEX))
    (ADD1 (NEXT-CLOCK (LIST 'ME INDEX)
                      (CLOCK-OF-TOKEN-MOVING-TO-WAIT
                        INDEX N CLOCK SYSTEM-STATE)))
    CLOCK)
```

Theorem 94: LEFT-CHANNEL-FULL-PROCESS-WAIT-IMPLIED-BY (REWRITE)

```
(IMPLIES (FULL-WAIT INDEX INDEX N
                    SYSTEM-STATE)
         (LEFT-CHANNEL-FULL-PROCESS-WAIT INDEX
                                         SYSTEM-STATE))
```

By progress statement Until-Critical, Clock-Of-Token-Moving-To-Critical computes the Clock when a Waiting process becomes Critical.

Theorem 95: CLOCK-OF-TOKEN-MOVING-TO-CRITICAL-WORKS (REWRITE)

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (NUMBERP INDEX)
              (LESSP INDEX N)
              (PROCESS-WAIT INDEX SYSTEM-STATE))
         (PROCESS-CRITICAL-WITH-TICKS
           (RANDOM-NUMBER
             (SUB1 (CLOCK-OF-TOKEN-MOVING-TO-CRITICAL
                     INDEX N CLOCK SYSTEM-STATE)))
           INDEX
           (INT (MAKE-PROCESS-LIST (SYSTEM-DESCRIPTION N))
                CLOCK
                (CLOCK-OF-TOKEN-MOVING-TO-CRITICAL
                  INDEX N CLOCK SYSTEM-STATE)
                SYSTEM-STATE)))

((DISABLE PROCESS-WAIT PROCESS-CRITICAL-WITH-TICKS
         CLOCK-OF-TOKEN-MOVING-TO-WAIT
         LEFT-CHANNEL-FULL-PROCESS-WAIT LEFT-FULL-WAIT-WAIT)
 (USE (UNTIL-PROCESS-CRITICAL
        (CLOCK (CLOCK-OF-TOKEN-MOVING-TO-WAIT
                 INDEX N CLOCK SYSTEM-STATE))
        (SYSTEM-STATE (INT (MAKE-PROCESS-LIST
                             (SYSTEM-DESCRIPTION N))
```

```
                           CLOCK
                           (CLOCK-OF-TOKEN-MOVING-TO-WAIT
                             INDEX N CLOCK SYSTEM-STATE)
                           SYSTEM-STATE)))
     (LEFT-CHANNEL-FULL-PROCESS-WAIT-IMPLIED-BY
       (N N)
       (INDEX INDEX)
       (SYSTEM-STATE (INT (MAKE-PROCESS-LIST
                             (SYSTEM-DESCRIPTION N))
                           CLOCK
                           (CLOCK-OF-TOKEN-MOVING-TO-WAIT
                             INDEX N CLOCK SYSTEM-STATE)
                           SYSTEM-STATE)))))
```

Theorem 96: WEIGHT-OF-PROCESS-ONE-IMPLIED-BY (REWRITE)

```
(IMPLIES (PROCESS-CRITICAL-WITH-TICKS TICKS INDEX
                                      SYSTEM-STATE)
         (EQUAL (WEIGHT-OF-PROCESS INDEX SYSTEM-STATE)
                1))
```

This version of Liveness is all that we want. We are not concerned with the value of the critical counter, as specified in Clock-Of-Token-Moving-To-Critical-Works, just that the Waiting Process does become Critical.

Theorem 97: LIVENESS (REWRITE)

```
(IMPLIES (AND (MUTUAL-EXCLUSIONP N SYSTEM-STATE)
              (NUMBERP INDEX)
              (LESSP INDEX N)
              (PROCESS-WAIT INDEX SYSTEM-STATE))
         (PROCESS-CRITICAL
           INDEX
           (INT (MAKE-PROCESS-LIST (SYSTEM-DESCRIPTION N))
                CLOCK
                (CLOCK-OF-TOKEN-MOVING-TO-CRITICAL
                  INDEX N CLOCK SYSTEM-STATE)
                SYSTEM-STATE)))

((DISABLE PROCESS-WAIT WEIGHT-OF-PROCESS
          PROCESS-CRITICAL-WITH-TICKS
          MUTUAL-EXCLUSIONP
          CLOCK-OF-TOKEN-MOVING-TO-CRITICAL
          CLOCK-OF-TOKEN-MOVING-TO-CRITICAL-WORKS)
 (USE (CLOCK-OF-TOKEN-MOVING-TO-CRITICAL-WORKS)))

(DISABLE WEIGHT-OF-PROCESS-ONE-IMPLIED-BY)                      (16)
```

References

1. Boyer, R. S., Moore, J S., *A Computational Logic*, Academic Press, 1979.

2. Boyer, R. S., Moore, J S., ‘‘The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover’’, Tech. report ICSCA-CMP-52, Institute for Computer Sciences and Computer Applications, January 1987.

3. Chandy, K. M., Misra, J., *Parallel Program Design: A Foundation*, To be published by Addison-Wesley, 1987.

4. Manna, A., Pnueli, A., ‘‘How to cook a Temporal Proof System for your Pet Language’’, *Proc. 10th ACM POPL*, ACM, 1983.

[5] Owicki, S., Lamport, L., ‘‘Proving Liveness Properties of Concurrent Programs’’, *ACM TOPLAS 4,3*, 1982, pp. 455-495.

6. Pnueli, A., *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. Current Trends in Concurrency. Overviews and Tutorials*, Springer Verlag LNCS 224, 1986, J.W. de Bakker, W.P. de Roever and G. Rozenberg eds.

7. Lengauer, C., Huang, C. H., ‘‘A Mechanically Certified Theorem About Optimal Concurrency of Sorting Networks’’, *Proc. 13th ACM POPL*, ACM, 1986, pp. 307-317.

Table of Contents

List of Figures

List of Tables