The Underlying Semantics of Transition Systems

J. M. Crawford
D. M. Goldschlag

Technical Report 17 December 1987

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

Acknowledgements

Crawford's work was sponsored in part at the University of Texas at Austin by Navy Contract N00014-86-0763. Goldschlag's work was sponsored in part at Computational Logic, Inc. by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151, and at the University of Texas at Austin by NSF Grant 81-22039. 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.

Abstract

This paper formalizes an operational semantics for the transition system model of concurrency and presents proof rules justified by that formalization. The operational semantics and the proofs rules have been mechanically verified on an automated theorem prover, and have been used to mechanically verify the correctness of a message passing solution to the n-processor mutual exclusion problem.

1. Introduction

The transition system model is the interpretation underlying many proof systems for the verification of concurrent programs. Such proof systems contain proof rules which are justified by the interpretation. Typically, these proof rules are demonstrated to be complete with respect to interesting properties of the interpretation. In this paper, we define an operational semantics that formalizes an interpretation of the transition system model. We then show how this operational semantics can be used as the basis of a proof system for concurrent programs and how this proof system can be mechanized in the context of an automated theorem prover.

2. Motivation

Formally defining an operational semantics of the interpretation underlying a proof system provides several important advantages over simply proposing proof rules. First, the operational semantics is complete; though proof rules may facilitate certain proofs, all proofs can be derived directly from the operational semantics. Second, all proof rules derived from the operational semantics are, by definition, consistent with the interpretation. Third, such bottom up development encourages the derivation of proof rules in a careful and minimalistic manner. Finally, carefully defining the underlying semantics is a first and necessary step when adapting the proof system to an automated theorem prover.

In this development, we do not use a logic designed specifically for our purposes. Rather, we state our theorems in a version of first order logic which allows recursive definitions. This methodology makes a clear distinction between the theorems in the proof system and the logical inference rules and syntax which define the logic. This logic has been carefully studied and proved to be sound. Hence, the theorems presented are also sound.

3. The Transition System Model

A concurrent program is a set of *events* which change the state of the system. In the transition system model, one assumes that all the events are *atomic*; that is, each event causes a single transition. Hence, the progress of the system can be simulated by an interleaving of the atomic events. The infinite sequence of states generated is called the *system computation*. We annotate the system computation in the following way: a state Si is the state preceding the i'th transition (figure one). The infinite sequence of events which generate the computation is called the *trace*. A trace in which every event occurs an unbounded number of times is called a *fair* trace. Computations generated from fair traces are called fair computations.

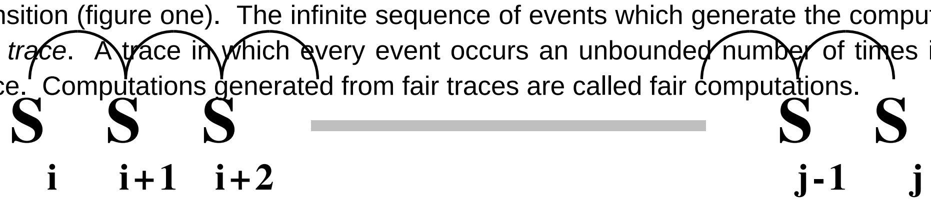

Figure 1: Figure One

The specifications of concurrent programs can be divided into the classes of safety and liveness properties. Safety properties are preserved for the entire system computation. Liveness properties state that some particular predicate will eventually hold. Because of liveness properties, we restrict our attention to fair traces (if every event were not guaranteed to happen infinitely often, liveness properties could not be proved). To prove that a program satisfies its specification, one proves that all fair computations of that program satisfy that specification. This amounts to assuming nothing about the actual ordering of the events, except that it is fair.

4. Formalizing the Transition System Model

We begin by formalizing the notion of event. An event is a function from states to states. That is:

```
P:  State → State
```

As an example consider an event which copies a single message from the left to the right channel (figure two). The function is defined as follows:

```
Copy(State)=
   If Emptyp("Left", State)
      Then State
      Else Send("Right",
                Head("Left", State),
                Receive("Left", State))
```

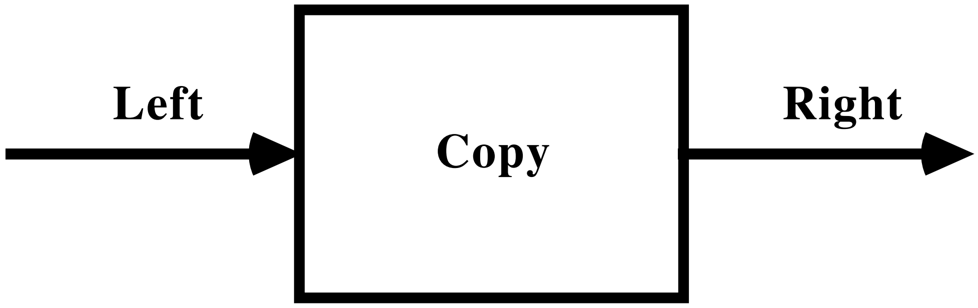

Figure 2

Figure 2: Figure Two

Copy tests the left channel (named "Left"); if the channel is empty, it returns the unchanged system state. If the channel contains a message, then that message (the head of the left channel) is sent upon the right channel (named "Right") in the state within which the left channel has been shortened.

In the Copy function, we model channels as first-in, first-out queues, using the functions Emptyp, Head, Send, and Receive. Emptyp tests whether a queue is empty, Head returns the first message in a non-empty queue, Send adds a message to the end of a queue, and Receive removes the head of a queue. The particular queue referenced is identified in the first argument to each function and the system state is the last argument to each function.

Each function may inspect and modify the entire system state. Even though in a typical implemented system the state is partitioned into local and global values and channels, the effect of each event is some function of the state.

5. Fairness

Each element of the trace is the name of the event that effects the corresponding transition in the system computation. We restrict ourselves to fair traces in the following manner:

Given the countable set U of all possible event names and the set N of the natural numbers, consider a total function Next:

```
Next:  U × N → N
```

such that:

```
(1)    Next(e, i) ≥ i
(2)    Next(e, i) = Next(d, i) → e=d
(3)    Next(e, i) > i → Next(e,i)=Next(e, i+1)
```

Next(e, i) is considered to be the next position of e in the trace at or after i (figure three). (1) guarantees that the position is at or after i. Since Next is total, such a position exists. (2) ensures that Next is one to one; no position in the trace may contain more than one element. (3) guarantees that Next is an honest scheduler: if Next(e, 2) is 5, then Next(e, 3) and Next(e, 4) are both 5. In fact, Next(e, 5) is also 5; at that point, since Next(e, i)=i, we say that e is scheduled at i.

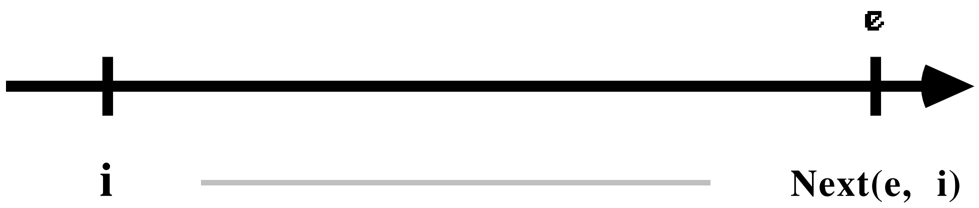

Figure 3
Figure 3: Next

When considering a Next function characterized only by axioms (1-3), we are considering any and all fair traces with elements in U (see proof in appendix). By using Next, we can identify the endpoints of finite intervals in the trace; by composing those segments of the trace, we can reconstruct the entire fair trace. Hence, Next gives us a concrete grasp on an arbitrary fair trace.

6. Interpretation of Next

Let ***P*** and ***Q*** be two distinct event names. Next(***P***, 1+Next(***P***, i)) is the position of the second occurrence of ***P*** in the trace at or after position i (figure four). Adding one to the inner term is necessary because Next returns a value greater than or *equal to* its second argument.

f

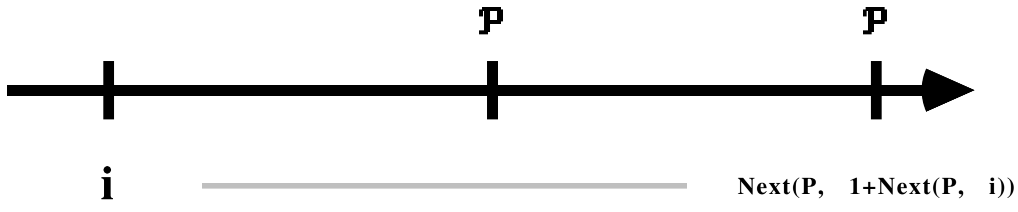

Figure 4: Figure Four

Min(Next(***P***, i), Next(***Q***, i)) is the next position of ***P*** or ***Q*** at or after i in the trace, whichever occurs first (figure five).

Max(Next(***P***, i), Next(***Q***, i)) is the earliest position in the trace, at or after i, such that both ***P*** and ***Q*** have occurred (figure five).

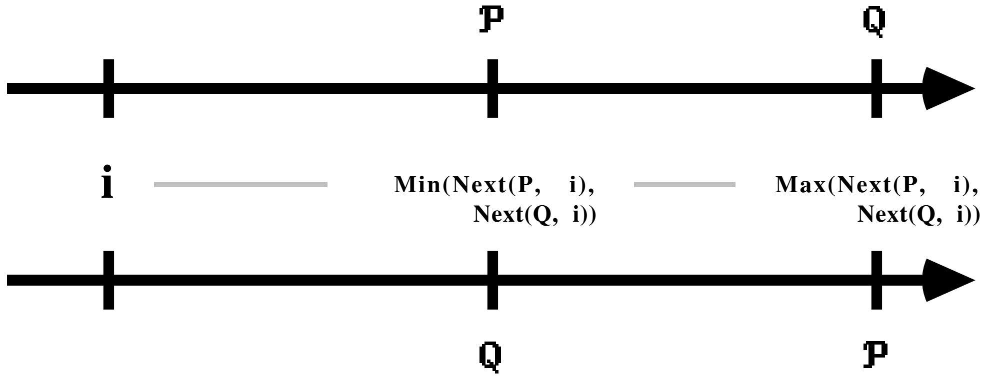

Figure 5: Figure 5

The latter two examples are easily generalized to n different events. Manipulations of Next are important when proving liveness properties.

7. The Operational Semantics

We now have a sufficient foundation to define the system computation. The state S_j at the end of the interval (i, j) beginning with state S_i is generated by effecting the i'th transition on S_i, returning state S_{i+1}, and repeating with the rest of the interval (figure one). This is defined in the following function:

```
Int(E, i, j, State) =
   If i<j
     Then Int(E, i+1, j,
              Step(E, i, State))
     Else State
```

Int (mnemonic for interpreter) returns the state at the end of the interval (i, j). Given a set of event names E and an initial state State, Int checks whether it has reached the end of the interval. If it has, it returns State. If not, Int recurses on the rest of the interval using the subsequent state in the computation.

The function Step computes the subsequent state by determining which event is scheduled at i, and applying the corresponding function to the state. The event scheduled at i is the e in E such that Next(e, i)=i. If no such e is scheduled, then Step simply returns the unchanged state (e.g., it effects the identity transition).

8. Use of Int

Since Int returns the state at the end of an interval in the computation, the state at the end of the arbitrary interval (i, j) is Int(E, i, j, State). Properties true about states in this form are safety properties.

The state just after the first occurrence of ***P*** at or after i is Int(E, i, 1+Next(***P***, i), State). Adding one to the third argument is necessary because the transition preceding state S_j is the j-1'th transition and we wish to include the first occurrence of ***P*** in this interval. States in this form, which identify effective transitions, are central to the proofs of liveness properties.

The definition of Int completes the formalization of the operational semantics of the transition system model. Since Int characterizes a computation, all properties of that computation may be derived from it. Furthermore, since Next is characterized only by axioms (1-3), Int characterizes an arbitrary fair computation, so proofs about Int are valid for all fair computations.

9. Proof Rules

Although all properties of the computation may be derived directly from Int, proofs of certain properties are facilitated by proof rules. We now describe the invariance and progress proof rules. These proof rules are theorems derived from Int.

10. Invariance

A property is invariant if once it is true of some state in the computation, it is true for all subsequent states. Stated in terms of Int, invariance properties have the following form:

```
Q(State)
→
Q(Int(E, i, j, State))
```

That is, for the set of events E, if the property Q holds on a state, then it will hold on the state at the end of the interval (i, j); since i and j are universally quantified, Q is invariant over all intervals of the computation. Some invariance properties rely upon previously proved invariants. For example, if Q is invariant, and R is an invariant only when Q is, the invariance of R is stated as follows:

```
(Q(State) ∧
        R(State))
→
```

```
R(Int(E, i, j, State))
```

An invariance proof rule is subsumed by a stability proof rule: if a predicate is stable over all intervals, then that predicate is invariant. To prove that a predicate is stable over an interval, we define a function that tests whether the predicate is preserved over all transitions in the interval. For a predicate Q, we state:

```
Preserved(E, i, j, State)=
        (∀ t. i≤t<j.
                Q(Int(E, i, t, State)) → Q(Int(E, i, t+1, State)))
```

This function tests whether Q is preserved across every transition in the interval (i, j). This is weaker than testing whether a predicate is preserved by all transitions for all states.

If Q is preserved across an interval and it holds on the state at the beginning of the interval, then it holds at the end of the interval as well (figure six). The proof rule is stated as follows:

```
(Preserved(E, i, j, State) ∧
        Q(State))
→
Q(Int(E, i, j, State))
```

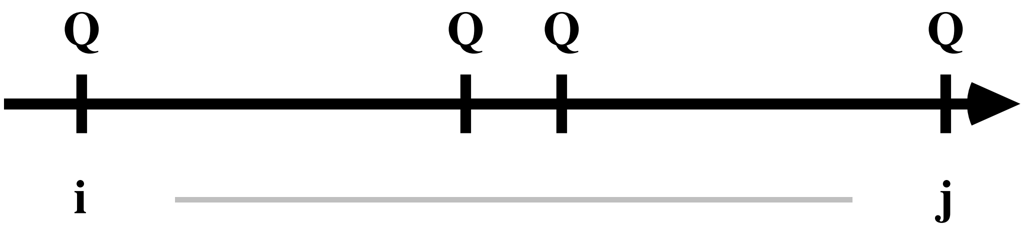

Figure 6
Figure 6: Figure 6

When using the invariance proof rule to prove an invariance property, one must prove that Q is preserved by all events for all states (just like the usual[1] invariance proof rule). That is, for every event P in E, prove statements of the form:

```
Q(State)
→
Q(P(State))
```

Such statements together satisfy the Preserved hypothesis in the proof rule, leaving the statement of invariance of Q.

[1]Unity, Temporal Logic.

11. Progress Proof Rule

Liveness properties have the following form:

```
(∃ j. Q(State)
        →
        R(Int(E, i, j, State)))
```

That is, for some set of events E, if Q holds on a state, then R will hold on some later state in the computation. Liveness properties are proved by composing more primitive progress properties. Progress properties require that Q be stable until some transition and that transition causes R to hold on the next state. The interval of the computation is described in figure seven.

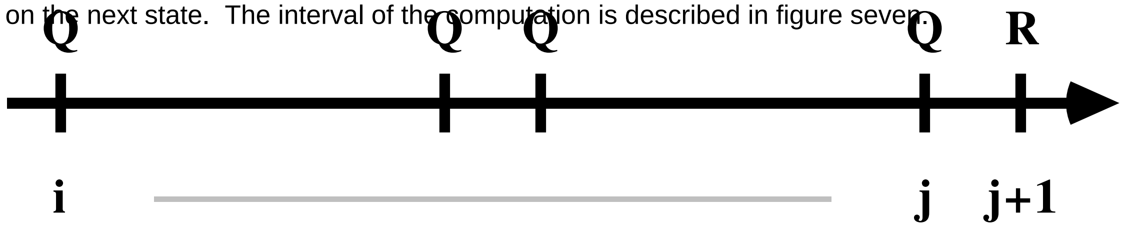

Figure 7: Figure Seven

The interval (i, j+1) can be divided into two parts. In the first part (i, j), Q is stable. The second part (j, j+1) is the single effective transition which yields R. This is stated in the progress proof rule:

```
(Preserved(E, i, j, State) ∧
        (Q(Int(E, i, j, State)) →
                R(Int(E, i, j+1, State))) ∧
        Q(State))
→
R(Int(E, i, j+1, State))
```

When using this proof rule to prove a progress property, one shows that for all states, every event but the effective one preserves Q; and that for all states, if Q holds and the effective event occurs, then R holds at the end of that transition. Typically, there is a single event (***P***) which is the effective transition. The corresponding interval of the computation is (i, 1+Next(***P***, i)). The final transition in this interval is the event ***P***. Hence, the progress statement is:

```
Q(State)
→
R(Int(E, i, 1+Next(P, i), State))
```

This is in the form of liveness statements. Specifying the end of the interval as a function of Next, instead of using an existential quantifier, provides a witness for that quantifier.

Progress statements are composed by induction or some other standard technique to prove more complex liveness properties.

12. Mechanization

The theorems presented here[2] have been verified by an automated theorem prover called the Boyer-Moore Prover [Boyer & Moore 79] An automated theorem prover is a program that will accept a statement as a theorem if and only if it can prove the statement from its axioms and from previously accepted theorems, using its inference rules. Although the Boyer-Moore Prover itself has not been mechanically verified, it has been carefully coded and extensively tested. It is very rare to find a soundness bug in the prover. The logic that it is based on, the Boyer-Moore Logic, has been proved sound. Mechanical verification ensures formality and increases one's confidence in the correctness of a theory.

13. An Example

We have mechanically verified a message passing solution to the n-processor mutual exclusion problem. The invariance property, mutual exclusion, states that at most one process may be in a special state (called a Critical state) at a time. The liveness property states that any process that wishes to enter its Critical state must eventually be allowed to do so. We now describe the solution informally, and present the statements, in our formalism, which verify the solution. The n processes are indexed zero to n-1. They are arranged in a ring, with a channel from the i'th to the i+1'th (modulo n) process (figure eight.)

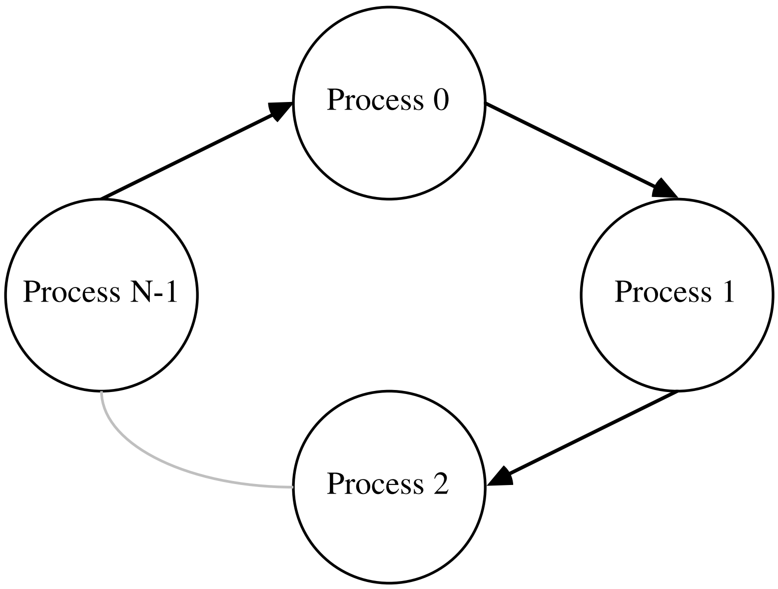

Figure 8: Figure Eight

Each process has three states: Non-Critical, Wait, and Critical (figure nine). A Non-Critical process non-deterministically goes into its Wait state, where it remains until it receives a token

[2] There are a few differences. Since the Boyer-Moore Logic does not define quantifiers [all variables are implicitly quantified], the predicate Preserved is defined recursively, using a Skolem function. Also, the definition of events provides for indexed events, where a single function defines an arbitrary number of events, each of which differs only by an index.

upon its incoming channel. At that point, it absorbs the token and becomes Critical for an arbitrary, but finite, number of transitions. Following the last Critical transition, the process releases a token upon its outgoing channel, and goes into its Non-Critical state. A Non-Critical process which does not become Wait, will pass a token from its incoming to its outgoing channel, if a token is available[3]. The ring of processes forms a non-deterministic network because of the non-deterministic transition between Non-Critical and Wait. Hence, fairness is required in this solution.

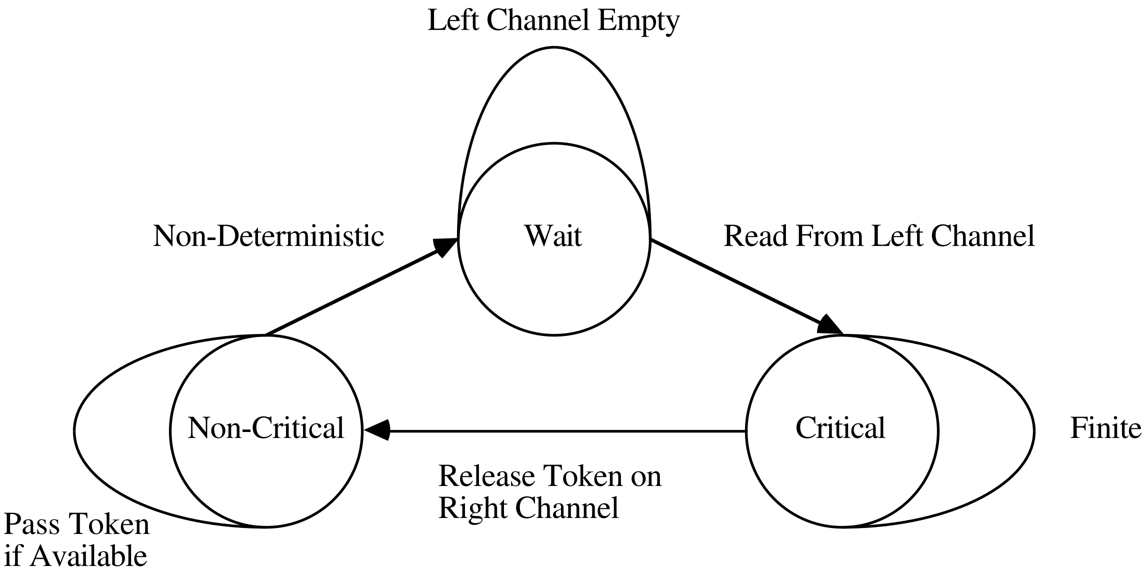

Figure 9: Figure Nine

The invariance statement is:

```
Mutual-Exclusionp(N, State)
→
Mutual-Exclusionp(N, Int(Processes(N), i, j, State))
```

Mutual-Exclusionp is a function which tests whether the sum of the number of tokens in the channels and the number of Critical processes is one. This ensures that at most one process is Critical at a time. Mutual-Exclusionp is a function of two arguments. The first, N, is the number of processes in the ring. The second is the state that it is to inspect. The function Processes, of the single argument N, forms a set of the names of the N processes in the ring. This statement is in the usual form of an invariance statement, and was proved using the invariance proof rule. Since N is universally quantified, Mutual-Exclusionp is invariant for all ring sizes.

The liveness statement is:

```
(∃ j.  (Mutual-Exclusionp(N, State) ∧
             Numberp(Index) ∧
             Index < N ∧
             Waiting(Index, State))
```

[3]The priority of becoming Wait is higher than the priority of passing a token. This ensures that a process can become Wait. A solution using the opposite priority also satisfies the specifications; yet, this ordering is better.

```
→
Critical(Index, Int(Processes(N), i, j, State)))
```

This theorem states that if a valid process (one with a numeric index from zero to n-1) is Waiting, then there exists some later state when that same process is Critical. Waiting tests whether the Index'ed process is Waiting and Critical tests whether the Index'ed process is Critical.

To prove this statement, we constructed a function that satisfies the j in the existentially quantified term by determining the effective transitions that move the token from its original position to the incoming channel of the Waiting process, and then adds one more transition of the Waiting process, to make it Critical. The sequence of effective transitions is specified as a composition of Next's. This function dovetails the composition of the Next's with the generation of the system computation. The liveness proof requires four progress proofs and is valid for all ring sizes.

14. Related Work

The emphasis of the proof rules' presented here on predicate transformation is due to Unity and temporal logic. However, in this work all the proof rules are justified by derivation from the operational semantics and have been mechanically verified. When using the invariance proof rule, one proves statements similar to the ones that one would prove before appealing to the respective invariance proof rules in Unity or temporal logic. The progress proof rule presented here is most similar to the Until proof rule of both Unity and temporal logic.

Lengauer [Lengauer 86] derives optimal concurrent executions of sequential programs. The transformation procedure has been mechanically verified. Lengauer's work differs from ours because it is not concerned with the direct mechanical verification of concurrent programs, nor are we concerened with the efficiency (except as it may be related to the correctness) of concurrent programs.

Clarke [Clarke 87] has also mechanically verified concurrent programs. His model is not completely general, as the programs must be of specific size (e.g., ring sizes of two or three, etc.). However, his verification procedure is completely automatic.

15. Conclusion

The operational semantics presented here formalize the transition system model. This formalization can be used as a basis of a proof system for concurrent programs by proving theorems about the operational semantics. Such theorems are typically called proof rules. In this way, the proof rules are implicitly justified by their derivation from a correct model. We feel that this bottom up methodology is a good one to follow when developing a proof system. The theorems have been formalized in a general purpose logic; this separates the concerns of proof system design from the design of a logic.

The Next function is a useful description of a fair trace, since it allows one to identify the position in the trace of the next effective transition and, hence, finite subsequences of the infinite trace. This also ties the proof rules closely to the trace, which yields strong proof rules. Using Next, we have also defined the transition system model in first order logic, instead of temporal logic.

Since these theorems have been proved on the Boyer-Moore Prover, we can now mechanically verify the correctness of concurrent programs. The general outline of a mechanical proof is very similar to a correct hand proof of the same properties. Such proofs are more reliable than hand proofs, but are more difficult to construct.

Appendix A
Proof that Next Represents an Arbitrary Fair Trace

We first define some notation. A trace is a sequence. A sequence is denoted $[t_{0,\ldots}, t_n]$ and the empty sequence is denoted []. The first index in a sequence is 0. The i'th element of a sequence T is T_i. A subsequence of a sequence T, from index i to j, is $T_{i\ldots j}$; the length of this subsequence is j-i+1. The prefix of length i+1 of a sequence T is $T_{0\ldots i}$. The sequence of all but the first i elements of T is $T_{i\ldots}$. The concatenation of a finite sequence S and a sequence T is S; T. The alphabet of sequence T is $\cup_i \{T_i\}$.

The set of Next functions is the set of functions satisfying axioms (1-3) characterizing a Next function, which were presented earlier. The set of fair traces is the set of all traces, the alphabet of each trace being a subset of U, where every element in a trace occurs in that trace an unbounded number of times.

We demonstrate an onto mapping from the set of Next functions to the set of fair traces.

Let Next be a Next function, and E be a subset of U. We first construct from Next a fair sequence with alphabet E.

The function Next *corresponds* to the sequence S iff:

S_i=e iff Next(e, i)=i $\quad \forall$ e $\in$ U, i $\in$ N.
There may exist some i for which no e satisfies Next(e, i)=i.

We restrict a prefix of length i of the sequence S to elements of the set E by:

Restrict(S, i, E)
=[] if i=0
=$[S_0]$; Restrict($S_{1\ldots}$, i-1, E) if i$\neq$0 $\wedge$ $S_0 \in$ E
=Restrict($S_{1\ldots}$, i-1, E) if i$\neq$0 $\wedge$ $S_0 \notin$ E

Let the *restriction* of S to E be the sequence $\mathrm{Lim}_{i\to\infty}$ Restrict(S, i, E). The alphabet of this sequence is E.

Lemma: The restriction of S to E is fair.

Proof: We prove by induction that for any e$\in$E and for any n, there exists a prefix of $\mathrm{Lim}_{i\to\infty}$ Restrict(S, i, E) containing n occurrences of e.

Base case: n=1. The prefix Restrict(S, 1+Next(e, 0), E) contains one occurrence of e.

Inductive step: Assume that the prefix Restrict(S, k, E) contains n>0 occurrences of e. Then, the prefix Restrict(S, 1+Next(e, k), E) contains n+1 occurrences of e.

This shows that the restriction to a set E subset of U of a sequence corresponding to a Next function is fair and has alphabet E.

Now, let T be a fair sequence with alphabet E subset of U. We construct a Next function such that the restriction to E of the sequence corresponding to that Next function is T.

Choose any fair sequence R, such that the alphabets of T and R are disjoint, and the union of the alphabets is U. We merge the first i elements of the sequences T and R by:

Merge(T, R, i)
    =[] if i=0
    =[T_0, R_0]; Merge($T_{1...}$, $R_{1...}$, i-1) if i≠0

Let S=$\text{Lim}_{i\to\infty}$ Merge(T, R, i). Lemma: The restriction of S to E is T.

Proof: Define half(n)=(n-1)/2 if n is odd and n/2-1 if n is even. We prove by induction that for any n, Restrict(S, n, E) equals the prefix $T_{0...half(n)}$.

Base case: n=1 $\vee$ n=2. Restrict(S, n, E)=[T_0]

Inductive step: Assume, for n>0, that Restrict(S, n, E)=$T_{0...half(n)}$. Then, Restrict(S, n+1, E)=$T_{0...half(n+1)}$.

We define the Next function that corresponds to S by:

Next(e, i) = Min{j≥i | S_j=e}

This Next function satisfies axioms (1-3) and corresponds to S. Furthermore, the restriction of S to E is T. Q.E.D.

Because the set of Next functions maps onto the set of fair traces, considering an arbitrary Next function is equivalent to considering any and all fair traces. In the function Step, we use the correspondence defined here between a Next function and a sequence, when we say that an event e is scheduled at i if Next(e, i)=i.

References

[Boyer & Moore 79]
R. S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Clarke 87] E.M. Clarke, O. Grumberg.
Research on Automatic Verification of Finite State Systems.
Technical Report CS-87-105, CMU, January, 1987.

[Lengauer 86] C. Lengauer, C.H. Huang.
A Mechanically Certified Theorem about Optimal Concurrency of Sorting Networks.
In *Proceedings 13th ACM POPL*, pages 307-317. ACM, 1986.

Table of Contents

List of Figures