A Mechanically Verified Proof System for Concurrent Programs

David M. Goldschlag

Technical Report 32 January, 1989

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

This work was supported in part 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 author 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 report describes a mechanically verified proof system for concurrent programs. The proof system is based on Unity [7], but is defined with respect to an operational semantics of the transition system model of concurrency. All proof rules are justified by this operational semantics. This methodology makes a clear distinction between the theorems in the proof system and the logical inference rules and syntax which define the logic. The proof system has been implemented on the Boyer-Moore prover.

This proof system is suitable for the mechanical verification of concurrent programs on the Boyer-Moore prover. This paper presents a mechanically verified proof of an n-processor program satisfying both mutual exclusion and absence of starvation. The program and correctness statement are presented, along with the key lemmas that aided the automatic verification.

1. Introduction

Since the semantics of a programming language can be precisely specified, programs are mathematical objects whose correctness can be assured by formal proof. Mechanical verification, which uses a computer program to validate a formal proof, greatly increases one's confidence in the correctness of the validated proof. This paper describes a mechanically verified proof system for concurrent programs, and demonstrates the use of this proof system by a mechanically verified proof of an n-processor program satisfying both mutual exclusion and absence of starvation.

The proof system for concurrent programs presented in this paper is based on Unity [7], which has two important characteristics:

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

The proof system presented here differs from Unity and most other proof systems because its proof rules are theorems. All proof rules are justified by an operational definition of concurrency, which has also been formalized. This methodology makes a clear distinction between the theorems in the proof system and the logical inference rules and syntax which define the logic. Since the underlying logic is sound, the resulting theory is sound. Furthermore, the proof system is complete (with respect to the underlying logic) because all properties can be proved directly from the operational semantics, although the proof of certain properties is simplified by using the provided proof rules.

2. The Boyer-Moore Prover

In a mechanically verified proof, all proof steps are validated by a computer program called a theorem prover. Hence, whether a mechanically verified proof is correct is really a question of whether the theorem prover is sound. This question, which may be difficult to answer, need be answered only once for all proofs validated by the theorem prover. The theorem prover used in this work is the Boyer-Moore prover [3, 5]. This prover has been carefully coded and extensively tested. The Boyer-Moore logic, which is mechanized by the Boyer-Moore prover, has been proved sound [10, 4].

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

Interaction with the theorem prover is through a sequence of events, the most important of which are definitions and lemmas. A definition defines a new function symbol and is accepted if the prover can prove that the new function terminates. A lemma is accepted if the prover can prove it, using the logic's inference rules, from axioms, definitions and previously proved lemmas.

2.1 The Boyer-Moore Logic

This proof system is specified in the Nqthm version of the Boyer-Moore logic [5, 6]. Nqthm is a quantifier free first order logic that permits recursive definitions. It also defines an interpreter function for the quotation of terms in the logic. Nqthm uses a prefix syntax similar to pure Lisp. This notation is completely unambiguous, easy to parse, and easy to read after some practice. Informal definitions of functions used in this paper follow:

- **`T`** is an abbreviation for **`(TRUE)`** which is not equal to **`F`** which is an abbreviation for **`(FALSE)`**.
- **`(EQUAL A B)`** is **`T`** if **`A=B`**, **`F`** otherwise.
- The value of the term **`(AND X Y)`** is **`T`** if both **`X`** and **`Y`** are not **`F`**, **`F`** otherwise. **`OR`**, **`IMPLIES`**, **`NOT`**, and **`IFF`** are similarly defined.
- The value of the term **`(IF A B C)`** is **`C`** if **`A=F`**, **`B`** otherwise.
- **`(NUMBERP A)`** tests whether **`A`** is a number.
- **`(ZEROP A)`** is **`T`** if **`A=0`** or **`(NOT (NUMBERP A))`**.
- **`(ADD1 A)`** returns the successor to **`A`** (i.e., **`A+1`**). If **`(NUMBERP A)`** is false then **`(ADD1 A)`** is **`1`**.
- **`(SUB1 A)`** returns the predecessor of **`A`** (i.e., **`A-1`**). If **`(ZEROP A)`** is true, then **`(SUB1 A)`** is **`0`**.
- **`(PLUS A B)`** is **`A+B`**, and is defined recursively using **`ADD1`**.
- **`(LESSP A B)`** is **`A<B`**, and is defined recursively using **`SUB1`**.
- Literals are quoted. For example, **`'ABC`** is a literal. **`NIL`** is an abbreviation for **`'NIL`**.
- **`(CONS A B)`** represents a pair. **`(CAR (CONS A B))`** is **`A`**, and **`(CDR (CONS A B))`** is **`B`**. Compositions of car's and cdr's can be abbreviated: **`(CADR A)`** is read as **`(CAR (CDR A))`**.
- **`(LISTP A)`** is true if **`A`** is a pair.
- **`(LIST A)`** is an abbreviation for **`(CONS A NIL)`**. **`LIST`** can take an arbitrary number of arguments: **`(LIST A B C)`** is read as **`(CONS A (CONS B (CONS C NIL)))`**.
- **`'(A)`** is an abbreviation for **`(LIST 'A)`**. Similarly, **`'(A B C)`** is an abbreviation for **`(LIST 'A 'B 'C)`**.[1]
- **`(LENGTH L)`** returns the length of the list **`L`**.
- **`(MEMBER X L)`** tests whether **`X`** is an element of the list **`L`**.
- **`(APPLY$ FUNC ARGS)`** is the result of applying the function **`FUNC`** to the arguments **`ARGS`**.[2] For example, **`(APPLY$ 'PLUS (LIST 1 2))`** is **`(PLUS 1 2)`** which is **`3`**.

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

[1]Actually, the quote mechanism is a facility of the Lisp reader [16].

[2]This simple definition is only true for total functions but is sufficient for this paper [6].

Definition.

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

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

2.2 Eval$

`EVAL$` is an interpreter function in Nqthm. Informally, the term **`(EVAL$ T TERM ALIST)`** represents the value obtained by applying the outermost function symbol in **`TERM`** to the **`EVAL$`** of the arguments in **`TERM`**. If **`TERM`** is a literal atom, then **`(EVAL$ T TERM ALIST)`** is the second element of the first pair in **`ALIST`** whose first element is **`TERM`**.

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

The Lisp backquote syntax [16] can be used to write an equivalent expression.[3] Backquote (**`‘`**) is similar to quote (**`'`**) except that under backquote, terms preceded by a comma are not evaluated, which is precisely the desired effect. Therefore, the terms **`‘(PLUS X (QUOTE ,Y))`** and **`(LIST 'PLUS 'X (LIST 'QUOTE Y))`** are equal. So, **`(EVAL$ T (LIST 'PLUS 'X (LIST 'QUOTE Y)) (LIST (CONS 'X 5)))`** can be rewritten as **`(EVAL$ T ‘(PLUS X (QUOTE ,Y)) (LIST (CONS 'X 5)))`**.

2.3 Functional Variables

Functional variables are function symbols characterized by a set of constraints. A theorem may be instantiated with substitutions for functional variables if all the constraints about the functional variables being substituted for can be proved using the same substitutions.

To ensure the consistency of the constraints, one must present one old function symbol as a model for each functional variable. Every constraint, with each functional variable substituted by its model, must be provable.

2.4 The Kaufmann Proof Checker

The Boyer-Moore prover automatically proves a lemma by heuristically applying sound inference rules to simplify it to a value other than **`F`**. Sometimes, it is easier to direct the proof process at a lower level. The Kaufmann Proof Checker [11] is an interactive enhancement to the Boyer-Moore prover. It allows the user to manipulate a formula (the original goal) using sound operations; once all remaining goals have been proved, the original formula has been proved. The prover will then accept the new theorem, which will be used as if it were proved automatically.

[3]Thanks to Matt Kaufmann for showing me how backquote would be useful in this context.

2.5 Abbreviations

It is often useful to be able to include quantifiers in the body of a definition. Since the Boyer-Moore logic does not define quantifiers, this cannot be done directly. However, using a technique called skolemization [12], one can derive an equivalent quantifier free formula from a definition. If the definition is not recursive, the formula can be added as an axiom, while maintaining the theory's consistency.

For example, suppose we wish to define:
Definition.

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

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

Then, we may consider this definition to be the conjunction of two formulas:

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

We then skolemize (positive skolemization—to preserve consistency) both conjuncts by substituting for each existential a skolem function. The resulting formula is quantifier free and can be added as an axiom. Consistency is preserved since such a definition is truly an abbreviation (there is no explicit recursion and no interpreter axioms are added). Finally, the meaning of the skolemized formula is the same as the original definition because of the correctness of skolemization.

3. The Operational Semantics

The first level of this proof system formalizes an operational definition of concurrency based on the transition system model [14, 15, 7]. A *transition system* is a set of statements that effect transitions on the system state. A *computation* is the sequence of states generated by the composition of an infinite sequence of transitions on an initial state. *Fair computations* are computations where every statement is responsible for an infinite number of transitions. This type of fairness is often called weak fairness [7]; the corresponding computations are often called just computations [14].

We are only interested in fair computations, since those permit the proof of liveness properties (if a statement is ignored forever, certain properties may not be provable). Notice however, that fairness is a very weak restriction on the scheduling of statements. The transition system model accurately depicts an execution of a concurrent program if all statements are atomic, since the simultaneous execution of atomic statements is equivalent to some sequential execution of the statements. Since atomicity is implementation dependent, we will not be concerned with the atomicity of statements here.

3.1 A Concurrent Program

To permit non-deterministic program statements,[4] each statement is a relation from previous states to next states [13]. We define the function **`N`** so the term **`(N OLD NEW E)`** is true if and only if **`NEW`** is a possible successor state to **`OLD`** under the transition specified by **`E`**. The definition of **`N`** is:
Definition.

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

`N` applies the **`CAR`** of the statement to the previous and next states, along with any other arguments encoded into the **`CDR`** of the statement. A state can be any data structure.

We restrict ourselves to statements which always specify at least one successor state. (A transition may be the identity transition; however, without this restriction, executions may not be infinite.) Programs containing only such statements satisfy the predicate **`TOTAL`** which is defined as:
Definition.

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

3.2 A Computation

The execution of a concurrent program is an interleaving of statements in the program. The term **`(S PRG IN I)`** represents the **`I`**'th state in the execution of program **`PRG`** starting in state **`IN`**. The function **`S`** is characterized by the following three constraints:
S-Transition

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

This constraint states that if the program is non-empty (i.e., contains at least one statement) and is total, then consecutive states in the program execution satisfy the successor relation specified by the statement scheduled by the function **`CHOOSE`** at that point in the execution. We will characterize **`CHOOSE`** later.
S-Initial

```
(IMPLIES (AND (LISTP PRG)
              (TOTAL PRG))
         (EQUAL (S PRG IN 0)
                IN))
```

This constraint states that the zero'th state in the execution sequence is the initial state.

[4]In Unity, all statements are deterministic. Non-determinism sometimes simplifies specification. As in Unity, fixed points can still be proved, if all statements preserve the property in question.

S-Fixes

```
(IMPLIES (AND (LISTP PRG)
              (TOTAL PRG)
              (NOT (NUMBERP I)))
         (EQUAL (S PRG IN I)
                (S PRG IN 0)))
```

This constraint states that the function `S` coerces its third argument to a number.

3.3 Fairness

The function `CHOOSE` is a scheduler. It is characterized by the following constraints:

Choose-Chooses

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

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

Choose-Fixes

```
(IMPLIES (AND (LISTP PRG)
              (NOT (NUMBERP I)))
         (EQUAL (CHOOSE PRG I)
                (CHOOSE PRG 0)))
```

As in `S`, this constraint states that `CHOOSE` coerces its second argument to a number.

The function `S` is a computation. We wish to restrict `S` to fair computations. A scheduler is fair if it schedules every statement an infinite number of times. An equivalent constraint is that fair schedulers always schedule each statement again. This is specified by the function `NEXT` and its relationship to `CHOOSE`:

Next-Is-At-Or-After

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

This constraint states that for statements in the program, `NEXT` returns a value at or after `I`.

Numberp-Next

```
(IMPLIES (MEMBER E PRG)
         (NUMBERP (NEXT PRG E I)))
```

Furthermore, `NEXT` always returns a number.

Choose-Next

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

This constraint states that for a statement in the program, `NEXT` returns a future point in the schedule when that statement is scheduled.

Next-Fixes

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

As with `S`, `NEXT` coerces its third argument to a number.

This completes the definition of the operational semantics of concurrency. Since `S`, `CHOOSE`, and `NEXT` are characterized only by the constraints listed above, `S` defines an arbitrary fair computation of a concurrent program. Statements proved about **S** are true for any fair computation.[5] So theorems in which **PRG** is a free variable are really proof rules, and this is the focus of the next sections.

4. Specification Predicates

The interesting properties of concurrent programs are safety and liveness (progress). Safety properties are those which state that something bad will never happen [2]; examples are invariant properties such as mutual exclusion and freedom from deadlock. Liveness properties guarantee that something good will eventually happen [1]; examples are termination and freedom from starvation. Unity defines predicates which specify subsets of these properties. Stable properties, a subset of safety properties, are specified using **UNLESS**; progress properties, a subset of liveness properties, are specified using **ENSURES** and **LEADS-TO**. We now present definitions for these three predicates in the context of this proof system.

4.1 Unless

The function **EVAL** evaluates a formula (its first argument) in the context of a state (its second argument). Its definition is:

Definition.

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

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

Eval-Or.

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

That is, **EVAL** distributes over **OR**. Similarly, **EVAL** distributes over the other logical connectives. The definition of **UNLESS** is:

Definition.

```
(UNLESS P Q PRG)
   ⇔
(AND (TOTAL PRG)
     (FORALL OLD
             (FORALL NEW
                     (FORALL E (IMPLIES (AND (MEMBER E PRG)
                                             (N OLD NEW E)
                                             (EVAL P OLD))
                                        (EVAL (LIST ’OR P Q)
                                              NEW))))))
```

(UNLESS P Q PRG) states that every statement in the program **PRG** takes **P** states to states where **P** or **Q** holds. Intuitively, this means that once **P** holds in a computation, it continues to hold (it is stable), at least until **Q** holds. Notice that if **(UNLESS P ’(FALSE) PRG)**[6] is true for program **PRG** and **P** holds on the initial state, then **P** is an invariant of **PRG** (that is, **P** is true of every state in the computation).

[5] That is, **S**, **CHOOSE**, and **NEXT** are functional variables, and theorems proved about them can be instantiated with terms representing any fair computation.

[6] Notice the theorem: **(UNLESS P ’(FALSE) PRG)=(UNLESS P P PRG)**.

4.2 Ensures

The definition of **ENSURES** is:
Definition.

```
(ENSURES P Q PRG)
   ⇔
(AND (UNLESS P Q PRG)
     (EXISTS E (AND (MEMBER E PRG)
                    (FORALL OLD
                            (FORALL NEW
                                    (IMPLIES (AND (N OLD NEW E)
                                                  (EVAL P OLD))
                                             (EVAL Q NEW)))))))
```

(ENSURES P Q PRG) states that every statement in the program takes **P** states to **P** or **Q** states (that is, **(UNLESS P Q PRG)**) and there exists at least one statement that takes all **P** states to **Q** states. **ENSURES** is defined so the key statement is effective for all states (i.e., the existential is before the universal quantifier). This is important for program composition (section 5.2, page 11). Ensures specifies a progress property since **(ENSURES P Q PRG)** means that once **P** holds in a computation, it continues to hold for a finite number of states, after which **Q** holds.

4.3 Leads-To

LEADS-TO is the general progress predicate. It is a consequence of **ENSURES** and is defined as follows:
Definition.[7]

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

(LEADS-TO P Q PRG IN) states that if **P** holds at some point in an execution of program **PRG** starting with initial state **IN**, then **Q** holds at some later point in the computation.

Theorems about **LEADS-TO** often use elements of the definition of **LEADS-TO** in their statement (as opposed to in their proof). The skolemization of the abbreviation **LEADS-TO**, using the technique described in section 2.5 on page 4, is:

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

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

[7]As with **UNLESS** and **ENSURES**, **LEADS-TO** looks like an abbreviation. However, it is really a functional variable that is constrained just like an abbreviation defined using the model of **S**, since abbreviations are defined using axioms and for soundness, functional variables are not permitted in axioms.

LEADS-TO.

The second conjunct states that **LEADS-TO** is true if for an arbitrary starting point in the computation at which **P** holds, one can find a later **J** at which **Q** holds. The function **ILEADS** serves to fix the arbitrary point and it replaces the universal **FORALL I** in the definition of **LEADS-TO**. This conjunct is used to prove **LEADS-TO**.

4.4 Comparison with Unity Predicates

The definitions of **UNLESS**, **ENSURES**, and **LEADS-TO** presented here differ from Unity's definitions. In Unity, using Hoare triples [8], the definition of **UNLESS** for a program **PRG** is:

$$\mathtt{P\ UNLESS\ Q} \equiv \langle \forall \mathtt{S} : \mathtt{S\ IN\ PRG} :: \{\mathtt{P} \wedge \neg \mathtt{Q}\}\ \mathtt{S}\ \{\mathtt{P} \vee \mathtt{Q}\} \rangle$$

Unity's definition strengthens the precondition. However, the two definitions are interchangeable. For a program **PRG**, the Unity specification **P UNLESS Q** is **(UNLESS ‘(AND ,P (NOT ,Q)) Q PRG)**,[8] and **(UNLESS P Q PRG)** is **P UNLESS ¬P∧Q**.

Unity's definition of **ENSURES** differs similarly:

$$\mathtt{P\ ENSURES\ Q} \equiv (\mathtt{P\ UNLESS\ Q} \wedge \langle \exists \mathtt{S} : \mathtt{S\ IN\ PRG} :: \{\mathtt{P} \wedge \neg \mathtt{Q}\}\ \mathtt{S}\ \{\mathtt{Q}\} \rangle)$$

In this case, the definition presented here is more general: for a program **PRG**, the Unity specification **P ENSURES Q** is **(ENSURES ‘(AND ,P (NOT ,Q)) Q PRG)**.

Finally, Unity does not provide a definition for **LEADS-TO**. Rather, it presents three proof rules and defines **LEADS-TO** to be the strongest predicate satisfying those rules. In this way, Unity avoids formalizing an operational semantics that may be used to define **LEADS-TO**. Furthermore, Unity's method for defining **LEADS-TO** allows one to use induction on the length of proof (structural induction) to prove theorems about **LEADS-TO**. The soundness and completeness of Unity's **LEADS-TO** are discussed in [9].

However, if an operational semantics is formalized, and **LEADS-TO** is correctly defined using those functions, then the definition is sound and completely captures the intuitively desired meaning. All theorems about Unity's **LEADS-TO** are theorems of the **LEADS-TO** presented here and are proved by induction on the third argument to **S** (the index in the computation). Such theorems allow the proof of progress properties without appealing to the operational semantics.

5. Proof Rules

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

[8]This expression uses the Lisp backquote facility. See section 2.2, page 3.

5.1 Liveness

All liveness theorems will be expressed using **`LEADS-TO`**. However, we wish to be able to prove such theorems directly from the statements in a program, without reasoning about the computation. Since **`ENSURES`** is a predicate about program statements, and is itself a progress property, we can deduce simple progress properties using the following theorem:

Ensures-Proves-Leads-To

```
(IMPLIES (ENSURES P Q PRG)
         (LEADS-TO P Q PRG IN))
```

`LEADS-TO` is transitive. This property is especially important since it can be applied repeatedly by induction.

Leads-To-Transitive

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

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

Leads-To-Strengthen-Left

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

This theorem states that if **`(LEADS-TO P R PRG IN)`** holds, and **`Q`** is stronger than **`P`**, then one can deduce **`(LEADS-TO Q R PRG IN)`**. Since the obvious hypothesis, ∀ **`STATE (IMPLIES (EVAL Q STATE) (EVAL P STATE))`**, stating that **`Q`** is stronger than **`P`** cannot be stated in the Boyer-Moore logic, we must use a term that is implied by this hypothesis and still makes this statement a theorem. Such a term is obtained by taking advantage of the arbitrary initial point in the computation, using the function **`ILEADS`** (section 4.3, page 8). Notice that when using this theorem, one must still prove that **`Q`** is stronger than **`P`**. The next theorem states that the ending predicate can be weakened, using the function **`JLEADS`**.

Leads-To-Weaken-Right

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

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

Disjoin-Left

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

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

Cancellation-Leads-To

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

The next two proof rules demonstrate that an invariant is preserved throughout a computation. Therefore, it can be added as a conjunct to either the beginning or ending predicate in **LEADS-TO**. Notice that the program must be non-empty (since the execution of a non-empty program is not defined) and that the invariance of **P** is specified by **(UNLESS P '(FALSE) PRG)** and **(EVAL P IN)** where **IN** is the initial state.

Invariant-Adds-Left

```
(IMPLIES (AND (UNLESS P '(FALSE) PRG)
              (EVAL P IN)
              (LISTP PRG))
         (IFF (LEADS-TO Q R PRG IN)
              (LEADS-TO (LIST 'AND P Q) R PRG IN)))
```

Invariant-Adds-Right

```
(IMPLIES (AND (UNLESS P '(FALSE) PRG)
              (EVAL P IN)
              (LISTP PRG))
         (IFF (LEADS-TO Q R PRG IN)
              (LEADS-TO Q (LIST 'AND P R) PRG IN)))
```

The next theorem is extremely useful since it permits conjoining the hypotheses of a **LEADS-TO** theorem into the **LEADS-TO** term. This is necessary when a theorem is governed by typing rules (e.g., a variable is **NUMBERP**) and one wishes to use the **Disjoin-Left** theorem. Notice that the **Invariant-Adds-Left** theorem is a consequence of this one.

Insert-Into-Leads-To

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

The last theorem of this section, the **PSP** theorem, combines a progress and a safety property to yield a progress property [7].

PSP

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

This theorem is proved by induction on the computation. Intuitively, if some state satisfies both **P** and **R**, the **UNLESS** hypothesis states that **R** holds until **B** holds; furthermore, **Q** holds eventually. The only question is which of **Q** or **B** is reached first.

5.2 Program Composition

This section presents several theorems about program composition. Since programs are simply a list of statements, the composition of two programs is the concatenation of two lists (using the function **APPEND**). The predicates **TOTAL**, **UNLESS**, and **ENSURES** all compose.

The first theorem states that **TOTAL** composes.

Total-Union

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

A similar fact holds for **`UNLESS`**.

Unless-Union

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

For **`ENSURES`**, the equivalence may be generalized since **`ENSURES`** need not hold in both component programs. But **`ENSURES`** must hold for at least one.

Ensures-Union

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

6. A Sample Program

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

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

Informally, the program described here defines a ring of processes, each of which differ only by its location in the ring. A process can send a message to the next process in the ring and receive a message from the previous process in the ring. Each process has three states: non-critical, wait, and critical.[9] A non-critical process non-deterministically becomes waiting and remains waiting until a token becomes available on its incoming channel. It then absorbs that token and becomes critical, and remains critical for a finite number of steps after which it releases a token upon its outgoing channel and becomes non-critical. A non-critical process that does not become waiting will pass a token, if one is available, from its incoming to its outgoing channel.

The following sections are written using a bottom-up approach, where most functions are defined before they are used. Section 6.1 formalizes the transitions each process is permitted. Section 6.2 defines the set of statements that make up the ring. Section 6.3 specifies the correctness theorems for a solution to this mutual exclusion problem. Finally, sections 6.4 and 6.5 present the proofs of the correctness theorems.

[9]Actually, the critical state is any of a set of states. See section 6.1, page 13 for the definition of **`CRITICAL`**.

6.1 The Processes

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

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

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

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

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

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

Definition.

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

A process is critical if its status is neither non-critical nor wait.
Definition.

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

If a process is critical, then its status is a number representing the maximum number of steps for which the process may remain critical. The function **`FIX`** coerces non-numbers to zero.
Definition.

```
    (TICKS STATE INDEX)
       =
    (FIX (CDR (ASSOC (CONS 'ME INDEX) STATE)))
```

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

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

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

Definition.

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

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

Definition.

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

`ME` defines precisely what is a legal transition for each process. It must check whether there is only one process in the ring; in that case the incoming channel and the outgoing channels are really the same.

Furthermore, it must specify that although each process changes certain parts of the state, each process preserves other parts. This is akin to not disturbing the local variables of other processes. The term **`(UNCHANGED OLD NEW EXCPT)`** tests whether **`OLD`** and **`NEW`** agree on every key except for keys in the list **`EXCPT`**. Specifically **`(UNCHANGED OLD NEW NIL)`** means that **`ASSOC`** cannot infer a difference between the two lists. (For the definition of **`UNCHANGED`** see page 17.)

There are two places in **`ME`** where non-determinacy in used. The first is when the non-critical process may become wait or remain non-critical. The second is when the critical process decides whether to remain critical or become non-critical. **`ME`** guarantees only that if the process remains critical the counter on that process decreases, though it does not specify by how much. Therefore, if the counter is zero (tested by **`ZEROP`**), the process must become non-critical and release a token. If the counter is non-zero, the process can either decrease the counter or become non-critical and release a token.

6.2 The Program

A program is a list of statements. Each statement is a list **`(CONS FUNC ARGS)`** where **`FUNC`** is a function symbol. Recall (page 5) that the definition of the function **`N`**, which interprets statements, is:
Definition.

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

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

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

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

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

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

6.3 The Correctness Specification

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

Definition.

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

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

Definition.

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

The invariance of mutual exclusion is then the following theorem:

Mutual-Exclusionp-Prg

```
(UNLESS ‘(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
        ’(FALSE)
        (ME-PRG SIZE))
```

This theorem is in the form of invariance statements (all that is missing is whether **MUTUAL-EXCLUSIONP** holds on the initial state, and that will subsequently be a hypothesis to every theorem). Since **SIZE** is a variable (section 2.2, page 3), mutual exclusion is an invariant for all ring sizes.

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

Wait-Leads-To-Critical

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

The conclusion is a **LEADS-TO** statement, which specifies that for any process which is waiting during the computation there exists a later state when that same process is critical. The hypotheses state that the invariant **MUTUAL-EXCLUSIONP** holds on the initial state and that the process’s index must be a number less than **SIZE**. (This implies that the program is non-empty.)

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

6.4 The Proof of Mutual Exclusion

In this program, all statements are identical, except for an index which ranges between zero and **SIZE-1**. Therefore, when proving certain properties, it is simpler to prove that the property holds for an arbitrary process in the ring, rather than for each process in the ring. Recall that the definitions of **UNLESS** (page 7) and **TOTAL** (page 5) contain the term **(MEMBER E PRG)** where e is universally quantified. For this program, a useful theorem is:

Member-Me-Prg

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

This theorem rewrites `(MEMBER STATEMENT (ME-PRG SIZE))` to a conjunction which states that the name of the function is `ME`, the second element in the statement is a number less than `SIZE` and the last element in the statement is `SIZE`. Hence, this theorem rewrites formulas whose proof is inductive (because of `MEMBER`) to formulas whose proof is by case analysis on single element of the program.

Another useful theorem is about the function `UNCHANGED`. The definition of `UNCHANGED` is:

Definition.

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

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

Definition.

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

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

Uc-Commutative

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

Second, the order of elements in `KEY` is unimportant:

Uc-Commutative-2

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

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

About-Uc

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

This theorem canonicalizes `ASSOC`’s in a formula if there is a `UC` in the hypotheses. Typically, such formulas will state that values associated with certain `KEY`’s are changed and those keys will comprise

EXCPT. The remaining values are unchanged. This theorem rewrites the **ASSOC**'s of the unchanged keys to use the second argument of **UC** instead of the first. Remember that the previous two commutative theorems will have already canonicalized the **UC** term.

Now, we can prove that the program **(ME-PRG SIZE)** is total. This is proved in the following theorem:
Total-Prg

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

The key lemma used to prove this theorem is:
Total-Me

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

Recall that the definition of **TOTAL** (page 5) requires that there be a successor state for every previous state. The function **ME-FUNCTION** is a witness for these successor states (i.e., it computes a valid **NEW** state).

The proof of mutual exclusion is done in three steps. First, one proves that for the execution of any process the weight of its left channel, status, and right channel is preserved. Then one proves that the weight of every other part of the state is unchanged. Finally, one notes that the weight of the entire state is the sum of the weights of the triple and the rest. (This proof scheme is not valid for ring of size one, but that is simple case analysis.) We present simply the proof that the weight of the triple is constant.
Definition.

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

Weight-Of-Triple-Preserved

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

These theorems imply the invariance property:
Mutual-Exclusionp-Prg

```
(UNLESS '(MUTUAL-EXCLUSIONP STATE (QUOTE ,SIZE))
        '(FALSE)
        (ME-PRG SIZE))
```

6.5 The Proof of Absence of Starvation

The proof of liveness requires three **ENSURES** properties that demonstrate how a token moves around the ring. The **ENSURES** properties are also **LEADS-TO** properties. The first **ENSURES** property states that if a process is non-critical and has a token on its incoming channel, then it either passes the token to its outgoing channel, or becomes wait and leaves the token on its incoming channel.

Non-Critical-Left-Ensures-Wait-Left-Or-Right

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

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

Wait-And-Left-Channel-Ensures-Critical

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

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

Critical-Ensures-Less-Critical-Or-Right

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

This last **ENSURES** theorem is especially intersting because we can use induction to show that every critical process eventually releases a token on its outgoing channel. This is done by induction on the counter **TICKS**, and appealing to the proof rules **LEADS-TO-WEAKEN-RIGHT** and **CANCELLATION-LEADS-TO**. The theorem is:

Critical-Ticks-Leads-To-Right

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

Using the theorem, **LEADS-TO-STRENGTHEN-LEFT**, we can simplify this result to a process simply being critical, not critical with some value on its counter. This is possible, because every critical process certainly has some value on its counter.

Critical-Leads-To-Right

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

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

Any-Leads-To-Right

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

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

Wait-Unless-Critical

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

This lemma and and **ANY-LEADS-TO-RIGHT** are preconditions for the **PSP** theorem. The resulting **LEADS-TO** is:

Wait-Leads-To-Left-Wait-Or-Critical

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

It is simpler to include the unnecessary **(TRUE)** in **(AND (TRUE) . . .)** because of the structure of the proof. In any case, this simplifies, using **CANCELLATION-LEADS-TO** and the **LEADS-TO** theorem derived from **WAIT-AND-LEFT-CHANNEL-ENSURES-CRITICAL** to:

Wait-Leads-To-Critical

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

This is the liveness result we need to prove correctness.

All the theorems in this section were verified on the Boyer-Moore prover, with many extra lemmas. However, much of the case analysis, especially the possible transitions of each statement, was done automatically by the prover.

References

1. Bowen Alpern and Fred B. Schneider. "Defining Liveness". *Information Processing Letters 21* (1985), 181-185.

2. Bowen Alpern, Alan J. Demers, and Fred B. Schneider. "Safety Without Stuttering". *Information Processing Letters 23* (1986), 177-180.

3. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

4. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

5. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

6. R. S. Boyer and J S. Moore. The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover. Tech. Rept. ICSCA-CMP-52, Institute for Computer Science, University of Texas at Austin, January, 1988. To appear in the *Journal of Automated Reasoning*, 1988. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703..

7. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison Wesley, Massachusetts, 1988.

8. C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". *CACM 12* (1969), 271-281.

9. Charanjit S. Jutla, Edgar Knapp, and Josyula R. Rao. Extensional Semantics of Parallel Programs. Department of Computer Sciences, The University of Texas at Austin, November, 1988.

10. M. Kaufmann. A Formal Semantics and Proof of Soundness for the Logic of the NQTHM Version of the Boyer-Moore Theorem Prover. Institute for Computing Science, University of Texas at Austin, Austin, TX 78712, 1986. ICSCI Internal Note 229.

11. M. Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Tech. Rept. ICSCA-CMP-60, Institute for Computing Science, University of Texas at Austin, Austin, TX 78712, 1987. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

12. M. Kaufmann. Skolemization Explained Simply. Computational Logic, Inc., Austin, Texas 78703, 1987. CLI Internal Note 27.

13. Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. Tech. Rept. Research Report 15, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, January 1988.

14. Zohar Manna and Amir Pnueli. "Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs". *Science of Computer Programming 4* (1984), 257-289.

15. Z. Manna and A. Pnueli. Verification of Concurrent Programs: The Temporal Framework. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

16. G. L. Steele, Jr.. *Common Lisp The Language.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.

Table of Contents