The Verification of a Minimum Node Algorithm

David M. Goldschlag

Technical Report 50 November, 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 paper describes a proof system suitable for the mechanical verification of concurrent programs. Mechanical verification, which uses a computer program to validate a formal proof, greatly increases one's confidence in the correctness of the validated proof. If one can assume the correctness of the proof, then one need only read, understand, and accept the specification in order to trust the coded program.

This proof system is based on Unity [8], and may be used to specify and verify both safety and liveness properties. However, it is defined with respect to an operational semantics of the transition system model of concurrency. Proof rules are simply theorems of 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 underlying logic. Since this proof system essentially encodes Unity in another sound logic, and this encoding has been mechanically verified, this encoding proves the soundness of Unity.

This proof system has been mechanically verified by the Boyer-Moore prover, a computer program mechanizing the Boyer-Moore logic [5]. This proof system has been used to mechanically verify the correctness of a distributed algorithm that computes the minimum node value in a tree. This paper also describes this algorithm and its correctness theorems, and presents the key lemmas which aided the mechanical verification. The mechanized proof closely resembles a hand proof, but is longer, since all concepts are defined from first principles. This proof system is suitable for the mechanical verification of a wide class of theorems, since the underlying prover, though automatic, is guided by the user.

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 a distributed algorithm which computes the minimum node value in a tree.

Using a mechanized theorem prover to validate a proof presents an additional burden for the verifier, since machine validated proofs are longer and more detailed than a hand proof, and hence more difficult to produce. However, if one trusts the theorem prover, one may then focus one's attention on the specification which was proved. If the specification is appropriate, then the program is correct and may be trusted. In general, the specification is much shorter than the associated correctness proof. In the example in this paper, the specification is completely described in under two pages; the proof and program are incompletely summarized in fourteen.

The proof system for concurrent programs presented in this paper is based on Unity [8], 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. Since this proof system is essentially an encoding of Unity in another logic which has been proved sound, this proof system proves the soundness of Unity.

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] extended by the Kaufmann Proof Checker [15]. 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 [14, 4]. All of the definitions and theorems presented in this paper have been validated by the Boyer-Moore prover; many of the intermediate lemmas used in this process are not described.

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. By the judicious choice of lemmas and the order of their presentation, a user may guide the theorem prover through the verification of complicated theorems. Examples of such theorems are Goedel's Incompleteness Theorem [24, 25], and the verification of a microprocessor [12].

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**.

[1] Actually, the quote mechanism is a facility of the Lisp reader [27].

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

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

Definition.

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

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

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 [27] 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 Instantiation

Sometimes it is useful to describe classes of functions. This may be done by partially constraining functions; theorems about such functions are theorems for a whole class of functions.

To ensure the consistency of partially constrained function symbols, one must present one old function symbol as a model for each new symbol. Every constraint, with each new symbol substituted by its model, must be provable [7].

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 [15] 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 Definitions with Quantifiers

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, the quantifiers must be removed by a technique called skolemization. If the definition is not recursive, adding the skolemized definition preserves the theory's consistency [16].

For example, suppose we wish to define:
Definition.
(P X_1 X_2 ... X_N)
$\Leftrightarrow$
BODY
where **P** is a new function symbol of arity **N** and **BODY** is a quantified term mentioning only free variables in the set **{X_1, ... ,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:
((P X_1 X_2 ... X_N)
$\Rightarrow$
BODY)
$\wedge$
((P X_1 X_2 ... X_N)
$\Leftarrow$
BODY)

We then skolemize (positive skolemization—to preserve consistency) each 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.

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

3. The Operational Semantics

The first level of this proof system formalizes an operational definition of concurrency based on the transition system model [20, 19, 8]. 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 [8]; the corresponding computations are often called just computations [20].

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 [18]. 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.

For the example program in this paper, we will assume unconditional fairness, where every statement in the program is a total function. Therefore, 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 I)** represents the **I**'th state in the execution of program **PRG**. The function **S** is characterized by the following two constraints:

Constraint: S-Effective-Transition[5]

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

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

The two assumptions are:

- The program must be non-empty. This is stated by the term **(LISTP PRG)**. If the program

---

[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.

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

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

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

has no statements, then no execution may be deduced.

- There is some successor state from **(S PRG I)** under the statement scheduled by **(CHOOSE PRG I)**. (If the statement is disabled, no effective transition is possible. The next constraint specifies that the null transition occurs in this case.)

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

Constraint: S-Idle-Transition[6]

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

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

3.3 Fairness

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

Choose-Chooses

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

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

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**.

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.

There are four constraints that coerce non-numeric index arguments to **0** and identify **NEXT**'s type as

---

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

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

numeric. These are presented as a conjunction:

Constraint: Index-Is-Numeric

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

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

This completes the definition of the operational semantics of concurrency. Since **S**, **CHOOSE**, and **NEXT** are characterized only by the constraints listed above, **S** defines an arbitrary fair computation of a concurrent program. Statements proved about **S** are true for any fair computation.[7] 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)))
```

---

[7] That is, **S**, **CHOOSE**, and **NEXT** are constrained function symbols, and theorems proved about them can be instantiated with terms representing any fair computation.

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

Definition.

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

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

Notice that if **(UNLESS P '(FALSE) PRG)** is true for program **PRG** (that is **P** is a stable property) and **P** holds on the initial state (e.g., **(EVAL P (S PRG 0))**), then **P** is an invariant of **PRG** (that is, **P** is true of every state in the computation). In Unity, this implication is an equivalence: **P UNLESS FALSE** is true for every invariant. This is because Unity's **UNLESS** is defined with respect to the computation (reachable states). This definition has its advantages, but it complicates the precise statement of the union theorems (section 5.3). Instead, this **UNLESS** is not restricted to reachable states in the computation, but another predicate **INVARIANT** (section 4.4) is defined independently and is identical to Unity's definition of **INVARIANT** which is defined in terms of Unity's **UNLESS**.

4.2 Ensures

The definition of **ENSURES** is:

Definition.

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

(ENSURES P Q PRG) states that 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 5.3, page 13). In conjunction with **UNLESS**, **ENSURES** specifies a progress property since the conjunction of **(UNLESS P Q PRG)** and **(ENSURES P Q PRG)** implies that once **P** holds in a computation, it continues to hold for a finite number of states, after which **Q** holds. Suppose that every program statement takes **P** states to states where **P** or **Q** holds. Then we know that **P** persists, at least until **Q** holds. (This is formalized by **(UNLESS P Q PRG)**.) Furthermore, if there exists some statement that transforms all **P** states to **Q** states, then, by fairness, we know that statement will eventually be executed. If **Q** has not yet held, since **P** persists, **Q** will hold subsequent to the first execution of that statement.

4.3 Leads-To

LEADS-TO is the general progress predicate. It is a consequence of the conjunction of **ENSURES** and **UNLESS** (and **TOTAL**) and is defined as follows:

Definition.

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

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

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

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

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

4.4 Invariance Properties

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

Definition.

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

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

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

Definition:

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

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

4.5 Comparison with Unity Predicates

The major differences between the **UNLESS** and **ENSURES** defined here, and Unity's definitions, are that these predicates consider all states and not just reachable ones, and that statements here may be non-deterministic and may be disabled. Aside from this, however, the definition of **ENSURES** here differs slightly from Unity's. In Unity, using Hoare triples [11], the definition of **ENSURES** is:

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

Our **ENSURES** does not imply **UNLESS**; to achieve the same effect, one states that both hold (e.g., **(AND (UNLESS P Q PRG) (ENSURES P Q PRG))**).[8]

Unity does not provide a definition for **LEADS-TO**. Rather, it presents three proof rules and defines **LEADS-TO** to be the strongest predicate satisfying those rules. In this way, Unity avoids formalizing an operational semantics that may be used to define **LEADS-TO**. Furthermore, Unity's method for defining **LEADS-TO** allows one to use induction on the length of the proof (structural induction) to prove theorems about **LEADS-TO**. The soundness and completeness of Unity's **LEADS-TO** are discussed in [13].

However, if an operational semantics is formalized, and **LEADS-TO** is correctly defined using those functions, then the definition is sound. Furthermore, it is easy to infer the meaning of this **LEADS-TO**, and to see that it is the predicate that Unity's axioms mean to define. Also, **LEADS-TO** may be meaningfully negated; this negation implies eventual stability. All theorems (except for several new compositional theorems [21, 26]) about Unity's **LEADS-TO** are theorems of the **LEADS-TO** presented here and are proved by induction on the second argument to **S** (the index in the computation). Such theorems allow the proof of progress properties without appealing to the operational semantics and are equivalent to Unity's proof rules.

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

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, in conjunction with **UNLESS**, is a progress property, we can deduce simple liveness properties using the following theorem:

Theorem: Unconditional-Fairness

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

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

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.

Theorem: Leads-To-Strengthen-Left

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

This theorem states that if **(LEADS-TO P R PRG)** holds, and **Q** is stronger than **P**, then one can deduce **(LEADS-TO Q R PRG)**. Since the obvious hypothesis, ∀ **STATE (IMPLIES (EVAL Q STATE) (EVAL P STATE))**, stating that **Q** is stronger than **P** cannot be stated (easily) in the Boyer-Moore logic, we must use a term that is implied by this hypothesis and still makes this statement a theorem. Such a term is obtained by taking advantage of the arbitrary initial point in the computation, using the function **ILEADS** (section 4.3, page 9). Notice that when using this theorem, one must still prove that **Q** is stronger than **P**, and that one may assume any program invariants about **(S PRG (ILEADS Q PRG R))**, since it is a computation state. The next theorem states that the ending predicate can be weakened, using the function **JLEADS**.

Theorem: Leads-To-Weaken-Right

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

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

computation **(S PRG (ILEADS Q PRG R))**.[9]

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

Theorem: Disjoin-Left

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

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

Theorem: Cancellation-Leads-To

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

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

Theorem: Invariant-Implies

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

Using this proof rule and the **LEADS-TO** weakening and strengthening proof rules presented earlier, it is possible to add or remove an invariant, or any consequence of an invariant, to or from the beginning or ending predicate in a **LEADS-TO**. As an example, assume that **(LEADS-TO (LIST 'AND Q INV) R PRG)** is true and **INV** is an invariant of the program **PRG**. We wish to conclude that **(LEADS-TO Q R PRG)** is true as well. The latter property is obviously true by the following argument: if we are at a state satisfying **Q** and **INV** is an invariant of the program, then that state satisfies **(LIST 'AND Q INV)**. This makes the two **LEADS-TO** statements identical. Using the proof rules, this conclusion is proved by applying the theorem **Leads-To-Strengthen-Left** in the following way: let **Q** be **Q** and let **P** be **(LIST 'AND Q INV)**. The hypothesis requires that **Q** imply **(LIST 'AND Q INV)** when each is evaluated in the context of the state **(S PRG (ILEADS Q PRG R))**. **Q** holds trivially; it remains necessary to show that **INV** holds at that point. Appealing to the theorem **Invariant-Implies**, we see that if **INV** is an invariant, it holds on any other state in the computation. This satisfies the second proof obligation, so we may conclude that **(LEADS-TO Q R PRG)** is true. This use of the two theorems **Invariant-Implies** and **Leads-To-Strengthen-Left** (or **Leads-To-Weaken-Right**) replaces Unity's substitution axiom. In this proof system, Unity's substitution axiom only applies to computation properties [23, 22]; these are **LEADS-TO**, **INVARIANT**, and **EVENTUALLY-INVARIANT**.

The last theorem of this section, the **PSP** theorem, combines a progress and a safety property to yield a progress property [8].

Theorem: Psp

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

This theorem is proved by induction on the computation. Intuitively, if some state satisfies both **P** and **R**,

---

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

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

5.2 Safety Theorems

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

Theorem: Unless-Proves-Invariant

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

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

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

Theorem: Invariant-Consequence

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

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

5.3 Program Composition

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

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

Theorem: Total-Union

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

A similar fact holds for **UNLESS**.

Theorem: Unless-Union

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

ENSURES composes as well; however the key statement need be present in only one of the component

programs:[10]

Theorem: Ensures-Union

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

6. A Sample Program

This section presents a mechanically verified distributed algorithm that computes the minimum node value in a tree [17]. 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.

Although the algorithm is relatively simple, the proof was quite difficult. The proof here is based on a detailed hand proof prepared by Lamport, but is extended, since his proof only proved partial correctness. Furthermore, Lamport's proof did not analyze the inductive data structure, and was therefore incomplete. Although one may argue that proving the invariant adequately demonstrates correctness, the liveness proof was nearly as difficult.

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

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

The following sections are written using a bottom-up approach, where most functions are defined before they are used. Section 6.1 formalizes the transitions that each node is permitted. Section 6.2 defines the set of statements that comprises the program. Section 6.3 specifies the correctness theorems for a solution to this problem. Finally, section 6.4 presents the proof of the correctness theorems. The proofs have been validated on the Boyer-Moore prover extended with the Kaufmann proof checker.

6.1 The Transitions

The actions of each node are governed by several statements. Recall that the statement interpreter **N** (section 3.1, page 5), considers the **CAR** of the statement to be a function name, and uses the remainder of the statement to instantiate that generic function to a particular instance. We now present the names of these generic functions, from which we can build the program statements:

- The **START** statement initiates the process; the root node sends a message to each of its children requesting the minimum value of its subtree. We call these requests *find* requests.
- The **RECEIVE-FIND** statement is a **START** statement for non-root nodes.
- The **RECEIVE-REPORT** statement collects the result of a *find* request to a child node. If all

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

children are accounted for, the minimum value computed so far (considering the node's own value) is sent to the node's parent.

- The **ROOT-RECEIVE-REPORT** statement is a special **RECEIVE-REPORT** statement for the root, since the root has no parent node.

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

Each node possesses several variables. These are:

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

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

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

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

Definition:

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

NODES-REC is function that most naturally would have been written as two mutually recursive functions, one which traverses trees and the other which traverses forests. However, since the Boyer-Moore logic

does not admit mutually recursive definitions, we add a flag to **NODES-REC**'s argument list. Depending on the value of the flag, the function behaves as one or the other of the mutually recursive functions that we wanted.

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

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

Definition:

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

Definition:

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

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

Definition:

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

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

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

Definition:

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

Definition:

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

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

Definition:

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

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

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

Furthermore, using the function **CHANGED**, all variables and channels, except for the ones mentioned above, are unchanged. The term **(CHANGED OLD NEW EXCPT)** tests whether **OLD** and **NEW** agree on every key except for keys in the list **EXCPT**. (Only keys in **EXCPT** may change.) Specifically, **(CHANGED OLD NEW NIL)** means that **ASSOC** cannot infer a difference between the two lists.

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

Definition:

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

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

Definition:

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

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

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

Definition:

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

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

subtree.

MIN is defined in the following way:

Definition:

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

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

Definition:

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

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

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

6.2 The Program

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

Definition:

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

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

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

Definition:

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

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

Definition:

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

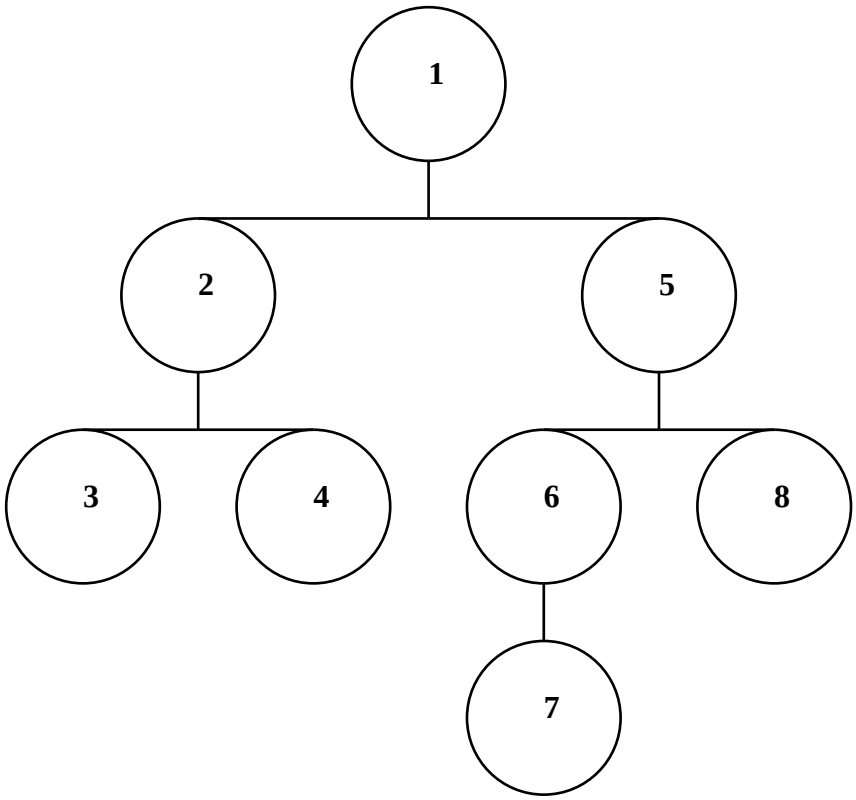
The entire program is the concatenation of all the program parts above. This is accomplished by the function **TREE-PRG**:

Definition:

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

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

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



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

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

where **(1 . 2)** is an abbreviation[11] for **(CONS 1 2)**.

6.3 The Correctness Specification

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

Definition:

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

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

Definition:

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

ALL-CHANNELS is defined as follows:

Definition:

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

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

[11] Lisp dot notation [27].

Definition:

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

Definition:

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

Definition:[12]

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

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

Definition:

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

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

Definition:

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

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

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

---

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

Theorem: Correctness-Condition

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

The conclusion implies that a state will eventually be reached where the correctness condition **(CORRECT TREE STATE)** is true. The hypotheses state that the tree is really a tree (no duplicate nodes, all nodes are numbers, and the tree is proper), and that the initial state satisfies the initial conditions (all statuses are **NOT-STARTED** and all channels are empty). Since **TREE** is a variable (section 2.2, page 3), this **LEADS-TO** statement holds for all the programs for all trees.

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

Theorem: Node-Values-Constant-Invariant

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

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

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

6.4 The Proof of Correctness

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

Theorem: Member-Tree-Prg [13]

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

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

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

syntax of one of the statements in the program. The latter information is much more easily used by the theorem prover, since it lends itself to case analysis.

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

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

Theorem: Total-Tree-Prg

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

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

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

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

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

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

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

Definition:

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

The augmented correctness condition is:

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

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

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

Definition:

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

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

Definition:

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

where **DONE** is:

Definition:

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

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

Definition:

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

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

Definition:

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

where **REPORTED** is:

Definition:

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

Definition:

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

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

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

Definition:

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

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

Theorem: Inv-Is-Invariant

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

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

Theorem: Initial-Conditions-Imply-Invariant

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

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

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

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

Theorem: Inv-Implies-Augmented-Correctness-Condition

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

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

Once the augmented correctness condition is proved, we prove the correctness condition by demonstrating the the measure **(TOTAL-OUTSTANDING (NODES TREE) TREE STATE)** decreases to **0**. This is proved

by demonstrating that whenever **TOTAL-OUTSTANDING** is non-zero, it will decrease. If **TOTAL-OUTSTANDING** is non-zero, one of the following scenarios must be true:

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

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

Theorem: Total-Outstanding-Decreases-Leads-To

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

By applying the theorem **Leads-To-Transitive** inductively on **COUNT**, it is possible to prove that **COUNT** decreases to zero. Furthermore, by application of the theorem **Leads-To-Strengthen-Left** (section 5.1, page 11), it is possible to substitute **(TRUE)** for the beginning condition, since every state certainly has some (numeric) **TOTAL-OUTSTANDING** measure. This is another example of the use of **Leads-To-Strengthen-Left** in infinite disjunction (section 5.1, page 11). The new theorem is:

Theorem: Termination

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

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

Theorem: Correctness-Condition

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

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

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

7. Conclusion

The proof system presented in this paper is based on Unity and has been verified on the Boyer-Moore prover. Proofs in this proof system resemble Unity hand proofs, except every concept must be defined and every theorem proved, in order to have the proof mechanically verified by the Boyer-Moore prover. This process of mechanical verification allows one to place greater trust in the correctness of a proof.

This trust comes at the expense of the extra work required to mechanically check a proof. However, if the theorem prover is trusted, then one can simply believe the (expensive) mechanically verified proof and focus on the more difficult issue of analyzing whether the specification that was proved is appropriate for the problem being solved.

The proof rules of Unity's **LEADS-TO** [8] are theorems of the **LEADS-TO** presented here. This augments the conceptual simplicity of non-operational correctness arguments with the soundness provided by an operational semantics. No axioms are present in this proof system; all concepts are developed from first principles.

The Boyer-Moore prover validates theories incrementally: each new lemma is accepted if the prover can automatically prove it from its current database of lemmas. Because of this, the complexity of theorems which may be proved depends on the skill of the user, rather than the theorem being in some class suitable for a decision procedure. Hence, this proof system is suitable for the mechanical verification of many concurrent programs [10]. Completely automatic systems for the mechanical verification of concurrent programs are cheaper to use [9]; however these proof checkers use decision procedures which are necessarily restricted to certain classes of theorems.

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, Boston, 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. R.S. Boyer, D. Goldschlag, M. Kaufmann, J Strother Moore. Functional Instantiation in First Order Logic. Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, 1991, pp. 7-26.

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

9. E.M. Clarke and O. Grumberg. Research on Automatic Verification of Finite State Systems. Tech. Rept. CS-87-105, CMU, January, 1987.

10. David Goldschlag. A Mechanically Verified Proof System for Concurrent Programs. Computational Logic, Inc. Austin Texas 78703, January, 1989. Technical Report 32.

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

12. Warren A. Hunt, Jr. "Microprocessor Design Verification". *Journal of Automated Reasoning 5*, 4 (December 1989), 429-460.

13. 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.

14. 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. ICSCA Internal Note 229.

15. 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.

16. Matt Kaufmann. DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers. Tech. Rept. 43, Computational Logic, Inc., May, 1989. Draft.

17. Leslie Lamport. . . Personal Communication.

18. Leslie Lamport. "A Simple Approach to Specifying Concurrent Systems". *Communications of the ACM 32* (1989), 32-45.

19. 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.

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

21. Jayadev Misra. Preserving Progress Under Program Composition. Tech. Rept. Notes on UNITY: 17-90, Department of Computer Sciences, The University of Texas at Austin, July, 1990.

22. Jayadev Misra. Soundess of the Substitution Axiom. Tech. Rept. Notes on UNITY: 14-90, Department of Computer Sciences, The University of Texas at Austin, March, 1990.

23. B. A. Sanders. Stepwise Refinement of Mixed Specifications of Concurrent Programs. In *Programming Concepts and Methods*, M. Broy and C. B. Jones, Eds., North Holland, Amsterdam, 1990.

24. N. Shankar. Proof Checking Metamathematics: Volumes I and II. Technical Report 9, Computational Logic, Inc., April, 1987.

25. N. Shankar. "A Mechanical Proof of the Church-Rosser Theorem". *Journal of the ACM 35* (1988), 475-522.

26. Ambuj Singh. Leads-To and Program Union. Tech. Rept. Notes on UNITY: 06-89, Department of Computer Sciences, The University of Texas at Austin, June, 1989.

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

Table of Contents

1. Introduction 1
2. The Boyer-Moore Prover 1
   2.1. The Boyer-Moore Logic 2
   2.2. Eval$ 3
   2.3. Functional Instantiation 3
   2.4. The Kaufmann Proof Checker 3
   2.5. Definitions with Quantifiers 4
3. The Operational Semantics 4
   3.1. A Concurrent Program 5
   3.2. A Computation 5
   3.3. Fairness 6
4. Specification Predicates 7
   4.1. Unless 7
   4.2. Ensures 8
   4.3. Leads-To 9
   4.4. Invariance Properties 9
   4.5. Comparison with Unity Predicates 10
5. Proof Rules 10
   5.1. Liveness 11
   5.2. Safety Theorems 13
   5.3. Program Composition 13
6. A Sample Program 14
   6.1. The Transitions 14
   6.2. The Program 21
   6.3. The Correctness Specification 24
   6.4. The Proof of Correctness 26
7. Conclusion 33

References 34