A Mechanical Formalization of Several Fairness Notions

David M. Goldschlag

Technical Report 65 March, 1991

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 Order 7406, and ONR Contract N00014-88-C-0454. 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, the Office of Naval Research, or the U.S. Government.

Abstract

Fairness abstractions are useful for reasoning about computations of non-deterministic programs. This paper presents proof rules for reasoning about three fairness notions and one safety assumption with an automated theorem prover. These proof rules have been integrated into a mechanization of the Unity logic [13, 14] and are suitable for the mechanical verification of concurrent programs. Mechanical verification provides greater trust in the correctness of a proof.

The three fairness notions presented here are unconditional, weak, and strong fairness [11]. The safety assumption is deadlock freedom which guarantees that no deadlock occurs during the computation. These abstractions are demonstrated by the mechanically verified proof of a dining philosopher's program, also discussed here.

1. Introduction

This paper presents a mechanical formalization of three fairness notions and one safety assumption. This formalization extends the mechanization of Chandy and Misra's Unity logic [10] described in [13, 14] and permits the mechanical verification of concurrent programs under assumptions of unconditional, weak, and strong [11], and the safety assumption of deadlock freedom. Deadlock freedom guarantees that no deadlock occurs during the computation.

Fairness abstractions are useful for reasoning about computations of non-deterministic programs. Assuming a fairness notion may permit delaying consideration of certain implementation issues at early stages of program design. This paper demonstrates these assumptions by the mechanically verified proof of a solution to the dining philosopher's problem.

The fairness notions are formalized as proof rules. These proof rules are either theorems of the operational semantics of concurrency presented here or are consistent with a restriction of that operational semantics. In the first case, the proof system is (relatively) complete, since all properties may ultimately be derived directly from the operational semantics. In the second case, completeness depends upon whether the proof rules in the literature are sufficient. However, the proof system is sound, since the restricted operational semantics justifies the new proof rules.

The mechanization presented here is an encoding of the Unity logic on the Boyer-Moore prover [5, 6]. Proofs in this system resemble Unity hand proofs, but are longer, since all concepts are defined from first principles. This proof system is semi-automatic since complex proofs are guided by the user. Mechanical verification provides greater trust in the correctness of a proof.

This paper is organized in the following way: Since the mechanization here has been done on the Boyer-Moore prover, section 2 presents a brief introduction to the Boyer-Moore logic and its prover. Section 3 presents an operational semantics of concurrency that justifies the proof rules presented in section 5. That section also presents the intuition behind each of the fairness notions. Section 7 presents the specification and proof of a solution to the dining philosopher's problem which illustrates the use of the proof rules. Section 8 summarizes related work and offers some concluding remarks.

2. The Boyer-Moore Prover

2.1 The Boyer-Moore Logic

This proof system is specified in the Nqthm version of the Boyer-Moore logic [6, 7]. 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**.
- **(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**.

2.2 Functional Instantiation

The theorem prover is directed by *events* submitted by the user. Definitions and theorems introduce new defined function and theorems, respectively. Partially constrained function symbols are defined by the *constrain* event which introduces new function symbols and their constraints. To ensure the consistency of the constraints, one must demonstrate that they are satisfiable. Therefore, the constrain event also requires the presentation of one old function symbol as a model for each new function symbol; the constraints, with each new symbol substituted by its model, must be provable [8]. There is no logical connection between the new symbols and their models, however; providing the models is simply a soundness guarantee.

All extensions to the Boyer-Moore logic presented in this paper were added using either the definitional principle or the constrain mechanism. Furthermore, the admissibility of these definitions and constraints was mechanically checked using the Boyer-Moore prover. This guarantees that the resulting logic is a conservative extension of the Boyer-Moore logic, and is therefore sound. All theorems presented here were mechanically verified by the Boyer-Moore prover enhanced with Kaufmann's proof checker [15].

---

[1] Actually, this quote mechanism is a facility of the Lisp reader [24].

[2] This simple definition is only true for total functions but is sufficient for this paper [7].

2.3 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].

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.

Notice that the quantifiers **FORALL** and **EXISTS** may occur only in non-recursive definitions and are automatically skolemized away by the theorem prover. For notational convenience, other quantified formulas are occasionally used in this paper (and were translated manually). In these cases, the quantifier symbols ∀ and ∃ are used instead.

3. The Operational Semantics

The operational semantics of concurrency used here is based on the transition system model [19, 20, 10]. 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. *Fairness* notions are restrictions of the scheduling of statements in the computation. For example, if every program statement is a total function, then *unconditional fairness* requires that each statement be responsible for an infinite number of transitions in the computation (every statement is scheduled infinitely often). Other fairness notions introduce the concept of *enabled* transitions, where a statement can only effect a transition if it is enabled (the statement can produce a successor state). These notions will be formalized in Section 5. Stronger fairness notions restrict the set of computations that a program may generate; hence a program's behavior may be correct under one fairness notion and not under another.

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

3.1 A Concurrent Program

A program is a list of statements. Each statement is a relation from previous states to next states [17]. We define the function **N** so the term **(N OLD NEW E)** is true if and only if **NEW** is a possible successor state to **OLD** under the transition specified by statement **E**. The actual definition of **N** is not important until one considers a particular program. For completeness, however, the definition of **N** is:

Definition: N

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

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

A statement **E** is *enabled* in state **OLD** if there exists some state **NEW** such that **(N OLD NEW E)** is true. That is, a statement is enabled if it can produce a successor state. We call such transitions *effective*. If a statement cannot effect any effective transitions from state **OLD** then it is *disabled* for that state. The

enabling condition for a statement is the weakest precondition guaranteeing an effective transition. A statement's effective transition may be the identity transition, however (e.g., the **SKIP** statement).

3.2 A Computation

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

Introducing extra skip states can be considered stuttering and is legitimate since repeated states do not interfere with either the safety or liveness properties discussed in section 4.) Fairness notions presented later will guarantee that a statement eventually executes an effective transition.

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

Constraint: S-Effective-Transition[3]

```
(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)**. Additional constraints about **CHOOSE** will be presented later.

The two assumptions are:

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

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

---

[3]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.

Constrain: S-Idle-Transition[4]

```
(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 The Scheduler

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

Constraint: Choose-Chooses

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

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

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

Scheduling every statement infinitely often is equivalent to always scheduling each statement again. This property is specified by the function **NEXT** and its relationship to **CHOOSE**:

Constraint: Next-Is-At-Or-After

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

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

Constraint: Choose-Next

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

This completes the definition of the operational semantics of concurrency.[5] Since **S**, **CHOOSE**, and **NEXT** are characterized only by the constraints listed above, **S** defines an arbitrary computation of a concurrent program. **S** guarantees that every statement will be scheduled infinitely often; transitions need not be effective. Statements proved about **S** are true for any computation.[6] So theorems in which **PRG** is a free variable are really proof rules, and this is the focus of the next sections.

---

[4]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)))
```

[5]There are several other constraints that coerce non-numeric index arguments to zero and identify **NEXT**'s type as numeric.

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

4. Specification Predicates

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

The interesting properties of concurrent programs are safety and liveness (progress). Safety properties are those that state that something bad will never happen [2]; examples are invariant properties such as mutual exclusion and freedom from deadlock. Liveness properties state that something good will eventually happen [1]; examples are termination and freedom from starvation. We borrow Unity's predicates for safety (**UNLESS**) and liveness (**LEADS-TO**) and present the definitions of these predicates in the context of this proof system.

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

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

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

Theorem: Eval-Or

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

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

Definition: Unless

```
(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 but **Q** does not 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. Unlike Unity's **UNLESS**, this **UNLESS** is not restricted to reachable states in the computation. This difference simplifies program composition and is key to the definition of the fairness

notion of deadlock freedom.

4.2 Leads-To

LEADS-TO is the general progress predicate. It is defined as follows:
Definition: Leads-To

```
(LEADS-TO P Q PRG IN)
   ⇔
(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 some later point in the computation.

Unity's theorems about **LEADS-TO** are theorems in this proof system as well. For example, **LEADS-TO** is transitive (this theorem may be applied repeatedly, by induction):
Theorem: Leads-To-Transitive

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

Also, Unity's PSP theorem, combining a progress and safety property to yield another progress property, is a theorem here as well:
Theorem: PSP

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

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

Curiously, **LEADS-TO** can be used to specify invariance properties. **(LEADS-TO P '(FALSE) PRG)** implies that the negation of **P** is invariant. This is deduced by contradiction: if **P** does hold at some point in the computation, then **'(FALSE)** would have to hold subsequently, which is impossible. Notice, that **(LEADS-TO P '(FALSE) PRG)** is only concerned with reachable states in the computation: it does not imply **(UNLESS (LIST 'NOT P) '(FALSE) PRG)**, even though **(LIST 'NOT P)** evaluates to the negation of **P**. (See section 4.1.)

5. Fairness

5.1 Unconditional Fairness

Fairness notions place restrictions on the scheduler, yielding more useful computations. The weakest fairness notion, *unconditional fairness*, requires that program states be always enabled (statements are total functions). Consequently, no restrictions are placed on the scheduler, other than that every statement be scheduled infinitely often (in no particular order). It is easy to imagine scenarios where starvation or deadlock occur under unconditional fairness.

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

Definition: Total

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

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

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

Definition: Ensures

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

Therefore, if a program is **TOTAL**, and **P** persists until **Q** and some statement transforms all **P** states to **Q** states, unconditional fairness implies that **(LEADS-TO P Q PRG)** holds as well. This argument is formalized in the following theorem, which is the proof rule for unconditional fairness:

Theorem: Unconditional-Fairness

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

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

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

5.2 Weak Fairness

Weak fairness is an extension of unconditional fairness for programs that are not **TOTAL**. Weak fairness excludes from consideration computations in which a statement is enabled continuously but is not scheduled effectively. That is, in order to guarantee that a statement will effect an effective transition under weak fairness, one must ensure that once it is enabled, it remains enabled (at least) until it is scheduled. Notice, however, that this is a simple property of the computation **S**: since, by fairness, every statement is

scheduled again, if some statement is continuously enabled from some point in the computation, it will execute effectively the next time it is scheduled. Hence, the proof rule describing weak fairness is a theorem that requires some knowledge about statements' enabling conditions.

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

Definition: Enabling-Condition

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

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

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

Definition: E-Ensures

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

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

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

Theorem: Weak-Fairness

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

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

In this theorem, the hypothesis requires that the key statement be enabled (or **Q** holds) even at that point. If that is the case, then we may deduce that the key statement is enabled continuously until the state before **Q** holds.

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

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

5.3 Strong Fairness

Strong fairness guarantees freedom from starvation. Starvation occurs when a process needs a resource, which is available infinitely often (but not necessarily continuously), yet only requests the resource when it is unavailable. Strong fairness precludes computations where a statement that is enabled infinitely often is never scheduled effectively. Equivalently, strong fairness requires that if a statement is enabled infinitely often, it is scheduled effectively infinitely often.

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

The proof rule formalizing this argument is:

Constraint: Strong-Fairness

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

The term **(STRONGLY-FAIR PRG)** introduces a new (undefined) function symbol that, essentially, tags uses of this proof rule. Since **(STRONGLY-FAIR PRG)** cannot be proved, it must be a hypothesis to any **LEADS-TO** property deduced using this proof rule. Furthermore, since reasoning directly about the operational semantics **S** yields nothing more than weak fairness, it is impossible to deduce stronger results about **S** without appealing to this proof rule. This proof rule is consistent with the rest of this theory because there exists a model for the function **STRONGLY-FAIR** satisfying this constraint: any unary function that is always false. Completeness and appropriateness is justified by the correctness of the supporting literature. [19, 18]

It may appear that this proof rule requires circular reasoning: it proves one **LEADS-TO** property by appealing to another. A clever answer is found in [19]: We can disregard the key statement when proving the **LEADS-TO** property in the hypothesis, since if it is ultimately scheduled effectively, we can ignore the **LEADS-TO** property in the hypothesis (since **Q** is then reached), and if it not scheduled effectively, we can ignore it when deducing that hypothesis (since it is equivalent to a skip statement). Other researchers

emphasize this point by requiring that the **LEADS-TO** property in the hypothesis be proved with respect to a smaller program: the original less the key statement. [18]

5.4 Deadlock Freedom

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

The proof rule formalizing this notion is:

Constraint: Deadlock-Freedom

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

This proof rules states that if **INV** is stable and **C** is statement **E**'s enabling condition, if **INV** is a stronger predicate than the negation of **C** then **INV** is false of every state in the computation. That is, the negation of **INV** is an invariant of the computation (section 4.2, page 7). Stating that **INV** is stronger than the negation of **C** with respect to computation states is more powerful than stating it with respect to all states (since computation states is a smaller set); the function **SOME-INDEX** is simply an arbitrary index. As with strong fairness, the new function **DEADLOCK-FREE** is an undefined function symbol which serves as a tag for uses of this proof rule. This proof rule is consistent with the rest of this theory because any unary function whose value is always false serves as a model for **DEADLOCK-FREE** satisfying this constraint.

In the next sections, the proof rules for weak fairness, strong fairness, and deadlock freedom are illustrated by the proof of a sample program.

6. More Specification Predicates

Before presenting the example program, it is helpful to introduce several additional specification predicates and proof rules, which will simplify both the statement and proof of the correctness theorems. The first predicate places assumptions on the initial state:

Definition: Initial-Condition

```
(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. Invariants are specified using the predicate **INVARIANT**:

Definition: Invariant

```
(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 (and that invariant's necessary initial conditions are satisfied), then **(INVARIANT INV PRG)** is true as well.

The next predicate specifies properties that are eventually stable. Such properties are important, as they represent generalizations of the notion of fixed points in the computation. (Unity specifies such properties using **UNLESS** and **LEADS-TO** and adding auxiliary variables to the program [21].) In our example, we will show that deadlock, which is a stable property, is, under certain hypotheses, eventually reached in the computation. Since deadlock freedom guarantees that deadlock is avoided, the hypotheses must be false. This is a key step in the correctness proof.

Definition: Eventually-Stable

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

(EVENTUALLY-STABLE R PRG) states that there exists a point in the computation, after which **R** holds continuously. It is interesting to note that such properties are the negation of **LEADS-TO** properties, in the sense presented in the following theorem:

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

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

This theorem states that if predicate **NOT-Q** is not eventually stable, and the negation of **NOT-Q** implies **Q**, then **Q** states occur infinitely often in the computation (i.e., **Q** states eventually follow any **P** state). The term **(S PRG (JES (ILEADS P PRG Q) PRG NOT-Q))** represents that arbitrary state at which **NOT-Q** must imply **Q**. (It is obtained from the skolemization of the definition of **LEADS-TO**.)

The usual way to prove eventually stable properties is by the following theorem:

Theorem: Stable-Occurs-Proves-Eventually-Stable

```
(IMPLIES (AND (LISTP PRG)
              (UNLESS P '(FALSE) PRG)
              (LEADS-TO '(TRUE) P PRG))
         (EVENTUALLY-STABLE P PRG))
```

This theorem states that if the predicate **P** is a stable property, and is certainly reached during the computation, then it is eventually stable as well.

In addition, there are several other theorems useful for weakening and strengthening the predicate argument to **EVENTUALLY-STABLE**.

7. An Example Program

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

We first present the statements for each philosopher:

Definition: Thinking-To

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

This function represents the generic transition between thinking and hungry states for a philosopher with index **INDEX**. It states that a philosopher may take a transition between a thinking state and either another thinking state, or a hungry state. The function **UNCHANGED** states that every variable in the state (which is an association list matching variable names to values) except for the variable **(CONS 'S INDEX)** representing the state of philosopher **INDEX**, remains unchanged. Notice that this transition is always enabled: if it is executed when the philosopher is not thinking, then no values change.

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

Definition: Hungry-Left

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

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

The analogous function for picking up free right forks is:

Definition: Hungry-Right

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

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

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

Definition: Hungry-Both

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

The final statement represents the transition between eating and thinking, with the simultaneous release of

both forks:

Definition: Eating-To

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

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

Definition: Phil

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

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

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

7.1 The Correctness Specification

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

Theorem: Correctness

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

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

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

Definition: Proper-Phil

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

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

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

Theorem: Phil-Prg-Invariant

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

This theorem states that if the initial state is legal, then both **PROPER-PHILS** and **PROPER-FORKS** are invariant.

7.2 The Correctness Proof

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

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

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

Theorem: Owns-Both-Leads-To-Eating

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

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

Theorem: Eating-Leads-To-Free

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

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

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

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

The negation of the **EVENTUALLY-STABLE** term in the hypotheses implies that the philosopher's right fork becomes free infinitely often, or is owned by the philosopher's right neighbor. More succinctly, this is equivalent to:

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

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

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

Theorem: Never-All-Rights

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

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

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

Theorem: Hungry-Leads-To-Owns-Left

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

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

Theorem: Correctness

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

This proof has been mechanically verified on the Boyer-Moore prover, with the help of many intermediate

lemmas.

8. Conclusion

This paper presents a formalization, in the Boyer-Moore logic, of four fairness notions: unconditional, weak, and strong fairness, and deadlock freedom. This formalization has been implemented on the Boyer-Moore prover and is suitable for the mechanical verification of concurrent programs. This was demonstrated by the mechanically verified proof of a solution to the dining philosophers program whose correctness depended upon strong fairness and deadlock freedom. Mechanical verification increases the trust one may place in a proof.

[3] presents three criteria for judging new fairness notions. They are:

- Feasibility: For every program, some fair computation does exist. Furthermore, it should be possible to extend every partial computation to a fair one (so the scheduler is implementable).
- Equivalence robustness: computations that are equivalent up to the ordering of independent transitions are all either fair or unfair.
- Liveness enhancement: The fairness notion make liveness properties hold that would otherwise be false.

Three of the fairness notions presented here, unconditional, weak, and strong fairness satisfy all three of these criteria. Deadlock freedom does not satisfy the feasibility criteria since, in general, it is not possible for a practical scheduler to always prevent deadlock. However, deadlock freedom may prove to be a useful abstraction for modeling those systems that do incorporate deadlock prevention schemes. At the most abstract level, deadlock situations would be identified, and then assumed to be avoided.

Other researchers have embedded one logic within another mechanized logic in order to prove soundness and provide mechanized support for the new logic. Hoare logic was embedded in LCF in [23], Dijsktra's weakest preconditions were embedded in HOL in [4], while CSP was embedded in HOL in [9]. These mechanized theories have not been used to mechanically prove the correctness of other programs. Manna's and Pnueli's framework for proving both invariance and eventuality properties, under weak fairness, were formalized on the Boyer-Moore prover in [22]; this system was used to verify an example program computing binomial coefficients and several other programs. Lamport has encoded the proof rules from his Temporal Logic of Actions [18] for weak and strong fairness on LP [12] and has used the system to verify a program.

This operational semantics of concurrency presented here provides justification for both unconditional and weak fairness and for a subset of the Unity logic, by providing a model for an arbitrary weakly fair trace. The proof rules for strong fairness and deadlock freedom are consistent with this theory. Since this theory is a conservative extension of a sound logic, it is sound as well.

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. K.R. Apt, N. Francez, and S. Katz. "Appraising Fairness in Distributed Languages". *Distributed Computing 2* (August 1988), 226-241.

4. R.J.R. Back and J. von Wright. Refinement Concepts Formalized in Higher Order Logic. In *Programming Concepts and Methods*, M. Broy and C. B. Jones, Eds., North Holland, Amsterdam, 1990.

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

6. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

7. 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..

8. R.S. Boyer, D. Goldschlag, M. Kaufmann, J S. Moore. Functional Instantiation in First Order Logic. Tech. Rept. 44, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, May, 1989. Published in proceedings of the 1989 Workshop on Programming Logic, Programming Methodology Group, University of Goteborg, West Germany.

9. Albert Camilleri. "Reasoning in CSP via the HOL Theorem Prover". *IEEE Transactions on Software Engineering SE-16* (September 1990).

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

11. Nissim Francez. *Fairness.* Springer-Verlag, New York, 1986.

12. S.J. Garland, J.V. Guttag, J.J. Horning. "Debugging Larch Shared Language Specifications". *IEEE Transactions on Software Engineering SE-16*, 9 (September 1990).

13. David M. Goldschlag. Mechanizing Unity. In *Programming Concepts and Methods*, M. Broy and C. B. Jones, Eds., North Holland, Amsterdam, 1990.

14. David M. Goldschlag. "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover". *IEEE Transactions on Software Engineering SE-16*, 9 (September 1990).

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

18. Leslie Lamport. A Temporal Logic of Actions. Tech. Rept. Research Report 57, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, April, 1990.

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

20. 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.

21. Jayadev Misra. Auxiliary Variables. Tech. Rept. Notes on UNITY: 15-90, Department of Computer Sciences, The University of Texas at Austin, July, 1990.

22. David M. Russinoff. Verifying Concurrent Programs with the Boyer-Moore Prover. Tech. Rept. Forthcoming, MCC, Austin, Texas, 1990.

23. S. Sokolowski. "Soundness of Hoare's Logic: an Automatic Proof Using LCF". *TOPLAS 9* (1987), 100-120.

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

Table of Contents