A Verification System for Concurrent Programs Based on the Boyer-Moore Prover

David M. Russinoff

January, 1991

Abstract

We describe a mechanical proof system for concurrent programs, based on a formalization of the temporal framework of Manna and Pnueli as an extension of the computational logic of Boyer and Moore. The system provides a natural representation of specifications of concurrent programs as temporal logic formulas, which are automatically translated into terms that are subject to verification by the Boyer-Moore prover. Several specialized derived rules of inference are introduced to the prover in order to facilitate the verification of invariance (safety) and eventuality (liveness) properties. The utility of the system is illustrated by a correctness proof for a two-process program that computes binomial coefficients.

1 Introduction

Although computer programming is potentially an exact science, few programs are actually subjected to formal verification. Proofs of specifications for even the simplest programs are often difficult to produce and unmanageably complicated. Hand-generated proofs may contain errors that are no easier to detect than programming errors. Automatic theorem proving, therefore, is crucial to the process of reasoning formally about programs. The ultimate practicality of program verification will be determined by the extent to which the programmer can be relieved of the details of generating and checking correctness proofs through the application of mechanical provers.

Effective use of today's theorem provers, however, typically requires expertise that is uncommon among software engineers. Detailed knowledge of a prover's heuristics is often necessary in order to guide it successfully through a proof. Moreover, the classical logics underlying these systems are unsuitable for modeling some aspects of real programs, such as nondeterminism and concurrency. The Boyer-Moore prover [BoM79,BoM88a], as described in Section 2, is a particularly powerful verification tool, but is nonetheless susceptible to these criticisms. Based on a computational logic designed for reasoning inductively about recursive functions, this system provides no natural representation of concurrent programs or their specifications.

Considerable recent research [ChM88,Kro87,MaP81] has been devoted to the development of alternative formalisms for program specification and verification based on temporal logic and the state-transition program model. This approach allows natural representations of nondeterministic concurrent programs and their execution properties. Verification of these properties remains a tedious process, however, since no mechanical support has been provided for these formalisms.

Our goal is to combine the expressiveness of temporal logic with the power of mechanical theorem proving in the design of a simple concurrent program verification system. Our programming language, as presented in Section 3, is a reification of the Manna-Pnueli temporal framework [MaP81,MaP84], grounded in the Boyer-Moore logic. In Section 4, we describe a scheme for formally representing programs and their execution states as terms in this logic. This ultimately allows conjectured temporal properties of programs to be encoded as static formulas, which may then be submitted to the prover.

In Section 5, we derive a set of inference rules that embody strategies for proving two important classes of these formulas, corresponding to *invariance* (or *safety*) and *eventuality* (or *liveness*) properties. An invariance property is proved simply by showing that it is preserved across all possible transitions. Proving an eventuality is a more interesting problem, solved by means of well-founded orderings of program states. Here, we exploit a unique feature of the Boyer-Moore system: a built-in theory of ordinals. As an illustration of both of these property types, and as a demonstration of the utility of our system, we describe the semi-automatic verification of the correctness of a two-process program that computes binomial coefficients.

2 The Boyer-Moore System

Here we describe the *nqthm* version of the system, which is documented in [BoM88a] and includes the extensions described in [BGK89]. It is founded on a quantifier-free first-order logic with equality and a syntax resembling that of LISP. Thus, terms are constructed from parentheses and symbols denoting variables and functions. By convention, lower-case alphabetic characters, which do not appear in symbols of the logic, are used to denote metavariables representing terms.

The basic theory includes axioms characterizing four primitive functions:

- `TRUE` and `FALSE` are functions of zero arguments. The constants `(TRUE)` and `(FALSE)`, abbreviated as `T` and `F`, respectively, serve as distinct truth values, as ensured by the axiom $\texttt{T} \neq \texttt{F}$.
- `EQUAL` is a binary function. The value of `(EQUAL l r)` is either `T` or `F`, according to whether $\texttt{l} = \texttt{r}$.
- `IF` is a ternary function. The value of `(IF t l r)` is the value of `r` if $\texttt{t} = \texttt{F}$, and the value of `l` otherwise.

In terms of these primitives, functions are defined corresponding to each of the logical connectives, e.g.,

$$\texttt{(IMPLIES P Q)} = \texttt{(IF P (IF Q T F) T)}.$$

This allows formulas to be encoded as terms, i.e., given any formula ϕ we may construct a term `t` such that

$$\phi \leftrightarrow (\texttt{t} \neq \texttt{F})$$

is a theorem. For example, the formula $\texttt{X} \neq \texttt{Y} \rightarrow \texttt{(F X Y)} = \texttt{(G X)}$ is encoded as the term

```
(IMPLIES (NOT (EQUAL X Y)) (EQUAL (F X Y) (G X))).
```

When a term `t` appears in a context where a formula is expected, it is understood to be an abbreviation for the formula $\texttt{t} \neq \texttt{F}$.

Variables occurring in axioms and theorems are understood to be universally quantified. Thus, if a term `t` is a theorem and s is substitution of terms for variables, then the result $\texttt{t}/s$ of applying s to `t` may be inferred as a theorem by the rule of *instantiation*.

The logic also includes

- a principle that generates sets of axioms specifying new types of inductively constructed objects,
- a principle for admitting axioms that define new recursive functions, and
- a principle of induction by which theorems pertaining to these objects and functions may be inferred.

The basic theory contains three types of inductively constructed objects:

- The type *number* formalizes Peano arithmetic through axioms involving the recognizer `NUMBERP`, the constant `(ZERO)`, and the successor function `ADD1`. Other arithmetic functions are defined in terms of these, including `SUB1` (the inverse of `ADD1`), `LEQ` (the standard partial order), `LESSP` (strict partial order), `ZEROP` (a predicate that fails iff its argument is a non-zero number), `PLUS`, `DIFFERENCE`, `TIMES`, and `QUOTIENT` (the basic binary operations), and `FACT` (the factorial function).
- The type *cons* formalizes ordered pairs by means of the recognizer `LISTP`, the constructor `CONS`, and the accessors `CAR` and `CDR`. Functions corresponding to other familiar list-processing functions of LISP, such as `ASSOC`, `MEMBER`, and `LIST`, are defined in terms of these primitives.
- The type *litatom* consists of an object corresponding to each symbol of the logic. The litatom corresponding to the symbol `x` is abbreviated as `'x`.

Variable-free terms that are constructed by means of CONS from numbers, litatoms, and objects belonging to user-defined data types are called *explicit values*. It may be shown [Kau86] that the set of all explicit values constitutes a model for the logic. We shall refer to this model as the *intended model*.

Various syntactic conventions similar to those of LISP allow explicit values to be expressed succinctly. For example, the explicit value

```
(CONS 4 (CONS (CONS 'Z 5) 'NIL))
```

is abbreviated as `'(4 (Z . 5))`. Under these conventions, there is a natural extension of the correspondence between symbols and literal atoms that assigns to an arbitrary term `t` an explicit value, denoted `'t`, called its *quotation*, e.g., the quotation of `(PLUS X 3)` is the explicit value denoted by `'(PLUS X 3)`.

Along with this encoding of terms, there is an obvious scheme for encoding variable substitutions as alists (i.e., lists of conses). Thus, the substitution $\{\mathtt{X} \leftarrow 2, \mathtt{Y} \leftarrow 3\}$ is represented by the alist `'((X . 2) (Y . 3))`. This correspondence motivates the definition of the function `EVAL`, which behaves as a built-in interpreter for the logic. This function takes two arguments, which are expected to be a quotation of a term `'t` and an alist `s` representing a substitution s. If the term `t` is *tame* [BoM88b], which means roughly that it involves only primitive function symbols and those with tame recursive definitions, and if each variable occurring in `t` also occurs in `s`, then `(EVAL 't s)` $= \mathtt{t}/s$. Thus, the following is a theorem:

```
(EQUAL (EVAL '(PLUS X Y) '((X . 2) (Y . 3))) 5).
```

The *ordinals* comprise another built-in data type, which includes the numbers as a subtype. The ordinals are recognized by the predicate `ORDINALP` and are ordered by the relation `ORD-LESSP`, which extends `LESSP`. Their purpose is to represent well-founded orders that are more complex than the order of the natural numbers, such as lexicographic orders. For example, we may embed the lexicographically ordered set of all pairs of numbers in the ordinals by defining an ordinal-valued binary function `LEX2` such that for any numbers `x1`, `y1`, `x2`, and `y2`,

```
(EQUAL (ORD-LESSP (LEX2 x1 y1) (LEX2 x2 y2))
       (OR (LESSP x1 x2) (AND (EQUAL x1 x2) (LESSP y1 y2))))
```

holds. Similarly, we may define the function `LEX3` to represent the lexicographically ordered set of all triples of natural numbers. (We shall make use of this in Section 5.)

When a term is presented as a conjecture to the theorem prover, various heuristics are applied in an attempt to establish it as a theorem. It is often the case that guidance from the user is required. Thus, the conjecture might be preceded by a sequence of simpler lemmas leading to the desired result. Hints may be provided to facilitate the proofs of these lemmas. These hints may include instantiations of previously proved lemmas, suggestions of induction schemes, and instructions to avoid using certain lemmas that might lead in a bad direction.

In order to avoid duplication of effort in proving a class of conjectures that conform to an obvious pattern, it may be possible to establish a *derived rule of inference* that is applicable to all members of the class. Such rules may be derived according to the following procedure, for which the system provides mechanical support (see [BGK89]):

1. Introduce as an axiom a term `(a (f1) ... (fn))`, involving some 0-ary functions `f1`, ... , `fn` not appearing in any other axiom.

2. Prove as a theorem a term `(c (f1) ... (fn))`.

Subsequently, whenever a term `(a t1 ... tn)`, resulting from the axiom by replacing the constants `(f1)`, ... , `(fn)` by any variable-free terms `t1`, ... , `tn`, is provable, the rule may be invoked to infer the corresponding theorem `(c t1 ... tn)`. We shall introduce several derived rules of inference in Section 5 in connection with certain classes of program properties. In each case, it may be assumed that the rule was derived by the procedure outlined above.

3 A Model of Concurrency

Our concurrent programming language is based on both the logic described in Section 2 and the model of concurrency presented in [MaP81]. According to this model, a *program* is composed of a finite set of *processes* and is associated with a finite set of *state variables*, including

a) *input variables*, which remain constant throughout execution and are required to satisfy some *input condition*,

b) *global variables*, which are assigned *initial values* expressed in terms of the input variables and may be manipulated by any of the processes, and

c) *program counters*, each of which is associated with some process.

Each process is represented as a graph, the nodes of which are the admissible values of the corresponding program counter. One node of each process is identified as the *start node* and is the initial value of the program counter. Each arc of a process is associated with a *precondition* for traversal and a *transition value* corresponding to each global variable of the program.

In our reification of this model, all state variables and process nodes are symbols of the Boyer-Moore logic, while the input condition and all initial values and transition values of global variables are terms of the logic. We define a *state term* of a program P to be a tame term in which only state variables of P occur as variable symbols. We shall assume that all initial and transition values and preconditions are state terms, and moreover, that only input variables occur in the initial values.

A *state* of a program P is a variable substitution that assigns explicit values to the state variables of P. The *value* of a state term `t` with respect to a state s is $\mathtt{t}/s$. `t` is *satisfied* by s if $\mathtt{t}/s$ is true in the intended model.

An *initial* state of P is one that satisfies all of the following terms:

a) the input condition of P,

b) `(EQUAL y v)`, for each global variable `v` with initial value `y`, and

c) `(EQUAL p 'n)`, for each program counter `p` with start node `n`.

In order to ensure that P has at least one initial state, we shall always assume that the input condition of a program is satisfiable.

Let Q be a process of program P corresponding to the program counter `p` and let α be an arc of Q from node `n` to node `m` with precondition `c`. Then α is *enabled* with respect to a state s if s satisfies

```
(AND (EQUAL p 'n) c).
```

In this case, we shall also say that Q is *enabled* with respect to s. Furthermore, Q *transforms* s into a second state s' with respect to P if the following additional conditions are satisfied:

a) $\mathtt{p}/s' = \mathtt{'m}$,

b) $\mathtt{y}/s' = \mathtt{t}/s$ for each global variable `y` with transition value `t`, and

c) s and s' agree on each input variable and each program counter other than `p`.

A state s' is a *successor* of a state s with respect to P if either $s' = s$ or some process of P transforms s into s'. An *execution* of P is a sequence of states $\langle s_0, s_1, \ldots \rangle$ such that s_0 is an initial state of P and s_{i+1} is a successor of s_i for each i.

As an illustration of these definitions, we shall describe a simple concurrent program BC, adapted from an example that appears in [MaP81]. This program consists of two processes, m and d, represented in Figure 1 both graphically and as linear text. Note that the arc preconditions and transition values, which are omitted from the graphical representation, may be read from the text. Thus, the arc from `D0` to `D1` has only the trivial transition values and the precondition `(NOT (EQUAL Y2 K))`.

This program is designed to compute the binomial coefficient determined by its inputs `N` and `K`, i.e., the value returned by the following function:

Definition 1

```
(BINOMIAL N K)
     =
(QUOTIENT (FACT N) (TIMES (FACT (DIFFERENCE N K)) (FACT K)))
```

Process m performs the multiplication: using `Y1` as a loop variable, it multiplies the accumulator `Y3` successively by each number exceeding `(DIFFERENCE N K)` but not `N`. Concurrently, process d performs the division: using `Y2` as a loop variable, it divides `Y3` by each positive number not exceeding `K`. Note that the precondition `(LEQ (PLUS Y1 Y2) N)` for the arc from `D2` to `D3` is designed to

Program *BC*

State variables: N, K (*inputs*); Y1, Y2, Y3 (*globals*); M, D (*pcs*)

Input condition: (AND (AND (NUMBERP K) (NUMBERP N)) (LEQ K N))
Initial values: Y1 ← N, Y2 ← 0, Y3 ← 1
Start nodes: M ← M0, D ← D0

	Process m			
M0:	*if* (EQUAL Y1 (DIFFERENCE N K))		M0	M3
	then goto M3			
M1:	Y3 ← (TIMES Y1 Y3)			
M2:	Y1 ← (SUB1 Y1)			
	goto M0	M2		M1
M3:	*halt*			

	Process d			
D0:	*if* (EQUAL Y2 K) *then goto* D4			D4
D1:	Y2 ← (ADD1 Y2)		D0	
D2:	*loop until* (LEQ (PLUS Y1 Y2) N)			
D3:	Y3 ← (QUOTIENT Y3 Y2)	D3		D1
	goto D0			
D4:	*halt*		D2	

Figure 1: A Concurrent Program

postpone the division until Y2 is guaranteed to be a factor of Y3: at D2, Y3 is a multiple of N!/(Y1!(Y2 $\Leftrightarrow$ 1)!), which is divisible by Y2 if the precondition holds.

An execution of this program may be considered to *terminate* if some state of the execution (and hence each subsequent state) satisfies the term

```
(AND (EQUAL M 'M3) (EQUAL D 'D4)).
```

As we shall see, the value of Y3 at termination is (BINOMIAL N K), that is, any execution state of BC that satisfies the above term must also satisfy

```
(EQUAL Y3 (BINOMIAL N K)).
```

However, it is not the case that every execution of BC terminates. For example, if an execution reaches a state in which the program counter D has the value D2 and the precondition

```
(NOT (LEQ (PLUS Y1 Y2) N))
```

of the loop at D2 is satisfied, then the execution may continue by repeatedly traversing that loop and never traversing any arc of process m. In fact, according to our definition, the sequence $\langle s_0, s_0, \ldots \rangle$ is a valid execution for any initial state s_0.

In order to provide an accurate model of concurrency, since it is unrealistic to imagine one process to be infinitely slow relative to another, we must impose some *fairness* condition on our execution sequences. The following version of fairness, which is commonly known as *weak fairness*, is suitable for our purpose: an execution $\langle s_0, s_1, \ldots \rangle$ of a program P is *fair* if for each process Q of P and each i, if Q is enabled with respect to s_j for all $j \geq i$, then for some $j \geq i$, Q transforms s_j into s_{j+1} with respect to P. Thus, in a fair execution, a process cannot remain enabled forever without any of its arcs being traversed.

Properties of program executions are conveniently described by formulas that are constructed from state terms by means of the logical connectives "$\rightarrow$" and "$\neg$" and the temporal operators "$\Box$" and "$\Diamond$". The semantics of such formulas are defined as follows: If $\Sigma = \langle s_0, s_1, \ldots \rangle$ is a sequence of states of a program P, then for each $i \geq 0$, let $\Sigma^{(i)} = \langle s_i, s_{i+1}, \ldots \rangle$. The sequence Σ *satisfies* a formula ϕ if

a) ϕ is a state term for P and s_0 satisfies ϕ,

b) $\phi = \neg\psi$ and Σ fails to satisfy ψ,

c) $\phi = \psi \rightarrow \theta$ and Σ satisfies either $\neg\psi$ or θ,

d) $\phi = \Box\psi$ and $\Sigma^{(i)}$ satisfies ψ for all i, or

e) $\phi = \Diamond\psi$ and $\Sigma^{(i)}$ satisfies ψ for some i.

Finally, a program P *satisfies* ϕ if for every fair execution Σ of P and every $i \geq 0$, $\Sigma^{(i)}$ satisfies ϕ.

Program properties that are of common interest generally fall into two categories. The first of these is the class of *invariance* properties, which are represented by formulas of the form

$$\mathtt{r} \rightarrow \Box \mathtt{s} \tag{1}$$

where `r` and `s` are state terms. Note that such a formula is satisfied by a program if for every fair execution in which `r` is satisfied by some state, `s` is satisfied by that and every later state. The properties described by these formulas include mutual exclusion, deadlock freedom, and partial correctness. For example, the following represents the partial correctness of our binomial coefficient program BC:

```
□(IMPLIES (AND (EQUAL M 'M3) (EQUAL D 'D4))
          (EQUAL Y3 (BINOMIAL N K)))
```

The second interesting category is the class of *eventuality* properties, which are represented by instances of the formula

$$\mathtt{r} \rightarrow \Diamond \mathtt{s} \tag{2}$$

A program satisfies this formula if every execution state that satisfies the term `r` is eventually followed by a state satisfying the term `s`. The corresponding properties include termination, accessibility, and responsiveness. For example, the termination property of program BC is represented as

```
◇(AND (EQUAL M 'M3) (EQUAL D 'D4)).
```

4 Formalization of the Model

In this section, we describe a scheme for representing programs and their executions in the Boyer-Moore logic. We then introduce axioms that characterize fair executions, thus allowing program properties to be encoded as terms in the logic. That is, given a formula ϕ representing a conjectured temporal property of a program P, we shall establish a method for automatically generating a term that is true in the intended model iff ϕ is satisfied by P. This term may then be submitted to the theorem prover for mechanical verification.

Processes and programs are encoded as list structures. A process is represented as a list of length 2, consisting of a literal atom representing its program counter and a list of objects representing its nodes. Each of these node representations is itself a list with two members: the literal atom corresponding to the node and a list whose members represent the arcs emanating from the node. An arc is encoded as a list of three objects: the literal atom corresponding to the terminal node of the arc, the quotation of the arc's precondition, and an alist associating global variables with (the quotations of) their transition values. (A global variable that is its own transition value is omitted from this alist.)

According to this scheme, the processes m and d of our example program BC are encoded as the values of the constant functions `M` and `D`, respectively:

Definition 2

```
(M) = '(M ((M0 ((M3 (EQUAL Y1 (DIFFERENCE N K)) NIL)
                (M1 (NOT (EQUAL Y1 (DIFFERENCE N K))) NIL)))
           (M1 ((M2 (TRUE) ((Y3 . (TIMES Y3 Y1))))))
           (M2 ((M0 (TRUE) ((Y1 . (SUB1 Y1))))))
           (M3 NIL)))
```

Definition 3

```
(D) = '(D ((D0 ((D4 (EQUAL Y2 K) NIL)
                (D1 (NOT (EQUAL Y2 K)) NIL)))
           (D1 ((D2 (TRUE) ((Y2 ADD1 Y2)))))
           (D2 ((D3 (NOT (LESSP N (PLUS Y1 Y2))) NIL)
                (D2 (LESSP N (PLUS Y1 Y2)) NIL)))
           (D3 ((D0 (TRUE) ((Y3 . (QUOTIENT Y3 Y2))))))
           (D4 NIL)))
```

A program is encoded as a list of length 5 whose members correspond to its input variables, global variables, program counters, input condition, and processes. The first member of this list is an alist, associating the input variables with some set of values that satisfy the input condition. The sole purpose of these values is to allow the system to verify the satisfiability of the input condition (which is required by an earlier assumption). The next two members of the list are also alists, associating global variables with their initial values and program counters with their start nodes. Thus, BC is represented by the constant `(BC)`, defined by

Definition 4

```
(BC) = (LIST '((N . 5) (K . 3))
             '((Y1 . N) (Y2 . 0) (Y3 . 1))
             '((M . M0) (D . D0))
             '(AND (AND (NUMBERP K) (NUMBERP N)) (LEQ K N))
             (LIST (M) (D)))
```

Naturally, program states are encoded as alists associating state variables with their values. For example, the initial state suggested by the above encoding of BC becomes the alist

```
'((N . 5) (K . 3) (Y1 . 5) (Y2 . 0)
  (Y3 . 1) (M . M0) (D . D0)).
```

Note that this scheme provides a convenient representation of the value of a state term with respect to a state by means of `EVAL`. In particular, a state term `t` is satisfied by a state encoded as `a` iff `(EVAL 't a)` is a theorem.

We shall require several predicates, all of which may be defined as recursive functions in the logic (as has been done in [Rus90]), and which behave as described below with respect to our encoding scheme:

- `(PROGRAMP p)` $\Leftrightarrow$ `p` is (an encoding of) a program;
- `(INITIALP s p)` $\Leftrightarrow$ `s` is an initial state of a program `p`;
- `(ENABLEDP q s)` $\Leftrightarrow$ a process `q` is enabled with respect to a state `s`;
- `(TRANSP q s0 s1 p)` $\Leftrightarrow$ `q` transforms `s0` into `s1` with respect to `p`;
- `(SUCCEEDSP s0 s1 p)` $\Leftrightarrow$ `s1` is a successor of `s0` with respect to `p`.

Once defined, these predicates may be used to construct a formal characterization of fair executions. This characterization will take the form of two axioms that constrain the behavior of a binary function `X`. Let P be a program with encoding `p`. Our intention is that the sequence

$$\langle \texttt{(X p 0)}, \texttt{(X p 1)}, \texttt{(X p 2)}, \ldots \rangle$$

represents an arbitrary fair execution of P. The first axiom ensures that this sequence is an execution of P, and also, for convenience, coerces non-numerical indices to `0`:

Axiom 1

```
(AND (IMPLIES (PROGRAMP P) (INITIALP (X P 0) P))
     (IF (ZEROP I)
         (EQUAL (X P I) (X P 0))
         (SUCCEEDSP (X P (SUB1 I)) (X P I) P)))
```

Fairness of an execution of P may be characterized as follows: for each process Q of P and for each index i, there is some index $w \geq i$ such that either Q is not enabled in the w^{th} state of the execution, or Q transforms the w^{th} state into the $(w+1)^{st}$ state. This property is formalized by our second axiom, which involves a new ternary function `W`:

Axiom 2

```
(AND (LEQ I (W Q I P))
     (IMPLIES (AND (PROGRAMP P) (MEMBER Q (PROCESSES P))
                   (ENABLEDP Q (X P (W Q I P))))
              (TRANSP Q (X P (W Q I P))
                      (X P (ADD1 (W Q I P))) P)))
```

Our definitions ensure that every program has a fair execution. It follows that Axioms 1 and 2 preserve the consistency of the logic. In fact, it is possible to verify this by explicitly defining functions `X` and `W` that may be shown mechanically to satisfy these two axioms (see [Rus90]).

In the presence of these axioms, it is a simple matter to generate a term corresponding to a given invariance property. If `r` and `s` are state terms for P, then Formula (1) is satisfied by P iff the following term is true in the intended model:

```
(IMPLIES (AND (EVAL 'r (X p I)) (LEQ I J))          (3)
         (EVAL 's (X p J)))
```

In the special case `r` = `T`, this formula is equivalent to

```
(EVAL 's (X p J))                                   (4)
```

The formulation of eventuality properties is complicated by the absence of existential quantifiers. In order to circumvent this, we introduce a function `N`, constrained by

Axiom 3

```
(AND (LEQ I (N I P R))
     (IMPLIES (AND (EVAL R (X P K)) (LEQ I K))
              (EVAL R (X P (N I P R)))))
```

Thus, if a state term `r` is satisfied by some state `(X p K)` in our fair execution of P, where `(LEQ I K)`, then `(N I p 'r)` is one such `K`. Axiom 3 clearly does not violate the consistency of our theory.

Now, if `r` and `s` are state terms for P, then the eventuality property given by Formula (2) is satisfied by P iff the following is true in the intended model:

```
(IMPLIES (EVAL 'r (X p I)) (EVAL 's (X p (N I p 's))))    (5)
```

In the special case `r` = `T`, this reduces to

```
(EVAL 's (X p (N J p 's)))                          (6)
```

5 Verification of Program properties

Proving conjectures of types (3) $\Leftrightarrow$(6) generally involves obscure induction arguments that the prover is unable to construct without considerable guidance from the user. However, since the same arguments are applicable to different instances of these terms, they need only be carried out once and established as rules of inference that may be applied repeatedly.

For example, instances of (4) may be proved by means of the following, which allows the use of a previously derived invariant `i` in the proof of an invariant `s` of a program `p`:

Derived Rule of Inference 1

```
(AND (PROGRAMP p)
     (IMPLIES (INITIALP X0 p) (EVAL 's X0))
     (IMPLIES (AND (SUCCEEDSP X1 X2 p) (EVAL 's X1) (EVAL 'i X1))
              (EVAL 's X2))
     (EVAL 'i (X p J)))
```

```
(EVAL 's (X p J))
```

In fact, instances of (4) may often be proved mechanically with no guidance other than the instruction to apply Rule of Inference 1, possibly using some specified established invariant `i`. For example, we may prove the partial correctness of program BC (formula (11) below) in this way by applying this rule to the term generated by each of five invariance properties. The first of these states that `Y1` is at least `N` $\Leftrightarrow$`K`, with equality holding at `M3` and possibly at `M0`:

```
□(IF (EQUAL M 'M3) (EQUAL Y1 (DIFFERENCE N K))                    (7)
   (OR (LESSP (DIFFERENCE N K) Y1)
       (AND (EQUAL M 'M0)
            (EQUAL Y1 (DIFFERENCE N K)))))
```

This invariant is proved without any hint, i.e., `i` = `T` in this case.

A similar invariant gives `K` as an upper bound on `Y2`; equality holds at `D4` but not at `D1`, and `Y2` $>$ `0` at `D2` and at `D3`:

```
□(AND (AND (NUMBERP Y2) (LEQ Y2 K))                               (8)
       (IF (EQUAL D 'D1) (LESSP Y2 K))
         (IF (EQUAL D 'D4) (EQUAL Y2 K)
             (OR (EQUAL D 'D0) (NOT (ZEROP Y2)))))
```

The proof of (8) involves the input condition of the program, which therefore instantiates `i` in the rule of inference.

The next invariant, for which no hint is needed, states that `Y1` + `Y2` $\Leftrightarrow$`1` $\leq$ `N` at `D2`, and `Y1` + `Y2` $\leq$ `N` elsewhere:

```
□(LEQ (PLUS Y1 (IF (EQUAL D 'D2) (SUB1 Y2) Y2)) N)                (9)
```

The following result may be paraphrased as the invariance of `Y3` = `N`!/(`Y1`!`Y2`!), except that `Y1` must be replaced by `Y1` $\Leftrightarrow$`1` at `M2`, and `Y2` must be replaced by `Y2` $\Leftrightarrow$`1` at `D2` and at `D3`:

```
□(EQUAL Y3                                                        (10)
        (QUOTIENT (FACT N)
                  (TIMES (FACT (IF (EQUAL M 'M2)
                                   (SUB1 Y1)
                                   Y1))
                         (FACT (IF (MEMBER D '(D2 D3))
                                   (SUB1 Y2)
                                   Y2)))))
```

The proof of (10) depends on (7), (8), and (9), hence, in applying Derived Rule of Inference 1, `i` is given as the conjunction of these three terms. This proof also depends on a number of properties of natural numbers, including the theorem that $c!$ is divisible by the product $a!b!$ under the assumption that $a + b \leq c$. Once these properties are encoded and proved as rewrite rules, (10) may be proved automatically.

Finally, the partial correctness result is derived as a consequence of (7), (8), and (10):

```
□(IMPLIES (AND (EQUAL M 'M3) (EQUAL D 'D4))                    (11)
          (EQUAL Y3 (BINOMIAL N K)))
```

Our method for verifying eventuality properties is based on well-founded orderings of states. We identify a well-founded *measure* with respect to which the state of P decreases as execution proceeds and such that the conjectured property corresponds to a minimal value of the measure. This measure is represented as a term `m`, the value of which with respect to each state is an ordinal, i.e.,

```
(ORDINALP (EVAL 'm (X p I)))
```

is a theorem.

While in general there is no measure that decreases at each execution step, we may find a measure that decreases under action by one process and remains constant under action by all other processes. This process may vary from one state to another, and is given by the value of some term `h`. If we can show that this value remains constant and that the process is enabled as long as the value of `m` is constant, then fairness may be invoked to show that the value of `m` eventually decreases. The following rule formalizes this strategy for formulas of type (6):

Derived Rule of Inference 2

```
(AND (PROGRAMP p)
     (ORDINALP (EVAL 'm X))
     (MEMBER (EVAL 'h X) (PROCESSES p))
     (IMPLIES (EQUAL (EVAL 'm X1) (EVAL 'm X2))
              (EQUAL (EVAL 'h X1) (EVAL 'h X2)))
     (EVAL 'i (X p J))
     (IMPLIES (AND (NOT (EVAL 's X)) (EVAL 'i X))
              (OR (ENABLEDP (EVAL 'h X) X)
                  (NOT (MEMBER (CDR (ASSOC (PC (EVAL 'h X)) X))
                               (NODES (EVAL 'h X))))))
     (IMPLIES (AND (MEMBER Q (PROCESSES p)) (TRANSP Q X1 X2 p)
                   (EVAL 'i X1)
                   (NOT (EVAL 's X1)) (NOT (EVAL 's X2)))
              (IF (EQUAL Q (EVAL 'h X1))
```

```
        (ORD-LESSP (EVAL 'm X2) (EVAL 'm X1))
      (NOT (ORD-LESSP (EVAL 'm X1) (EVAL 'm X2))))))
```

```
(EVAL 's (X p (N J p 's)))
```

The functions PC, NODES, and PROCESSES, which appear in this rule, return the program counter and a list of the node labels of a process, and a list of the processes of a program, respectively. Thus, according to the first four hypotheses, m is an ordinal-valued measure and the value of h is a process of a program p, which is the same for two states with the same measure.

The fifth hypothesis simply states that i is an invariant of p. The sixth requires that with respect to any state that satisfies i but not s, the process determined by h is enabled, assuming that the value of its program counter is a valid node label. Finally, according to the seventh hypothesis, under action by a process Q of p, as long as i is satisfied and s is not, the value of m decreases if Q is the value of h and otherwise remains constant.

As an illustration of the utility of this rule, we shall describe a derivation of the termination property

```
◇(AND (EQUAL M 'M3) (EQUAL D 'D4))
```

of program BC, which completes a proof of its total correctness. In this case, p is (BC) and 's is the quotation of the above term. The construction of h is based on the observation that prior to the halting of process m, it is possible to traverse an arc of process d (namely the loop at D2) without progressing toward termination. (It follows from (9) and (10) that the loop is never enabled once process m has halted.) Thus, h is taken to be the term

```
(IF (EQUAL M 'M3) (D) (M))
```

Next, we construct a term m representing a well-founded measure. This measure will be a lexicographic order based on three components. Since Y1 decreases to 0 and Y2 increases to K as an execution approaches termination, the primary component of our measure is the quantity (PLUS Y1 (DIFFERENCE K Y2)). For fixed values of Y1 and Y2, the program counter M assumes the successive values M0, M1, and M2, while D assumes the values D2, D3, D0, and D1. Thus, the secondary and ternary components are

```
(LOC M '(M3 M2 M1 M0))
```

and

```
(LOC D '(D4 D1 D0 D3 D2))
```

respectively, where (LOC X L) is defined to be the number of entries in the list L that precede the first occurrence of X. We make use of the function LEX3 (discussed in Section 2), defining m to be

```
(LEX3 (PLUS Y1 (DIFFERENCE K Y2))
      (LOC M '(M3 M2 M1 M0))
      (LOC D '(D4 D1 D0 D3 D2)))
```

The termination of BC may now be established simply by invoking Rule 2.

6 Conclusion

Although the Boyer-Moore system was originally intended for the verification of properties of recursive functions, it has two important features that have allowed us to model and verify properties of concurrent programs. One of these is a built-in interpreter for the logic, which is essential to our scheme for encoding program properties as terms. The other is a representation of the ordinals, which allows proofs of eventuality properties based on arbitrarily complex well-founded relations.

The theory outlined here has been implemented as a verification system that has been used to produce mechanical proofs of correctness for a variety of programs, including an incremental garbage collector [Rus91], several programs that achieve mutual exclusion, and the example discussed herein. This system is composed of a set of axioms that extend the basic theory of the Boyer-Moore logic, along with a functional interface to the theorem prover that generates terms corresponding to conjectured program properties.

Our extension to the basic theory, consisting of the axioms and definitions of the functions introduced in Section 4, has been observed to be both provably consistent and adequate for representing invariance and eventuality properties of concurrent programs. Of course, the proofs of these properties may be quite complicated. Generating or understanding them may require intimate knowledge of the logic and theorem prover, especially the induction mechanism. Fortunately, however, these proofs generally conform to several patterns, which have been captured in the derived rules of inference presented in Section 5. These rules have all been verified mechanically to produce only logical consequences of our theory. While this process was somewhat tedious, requiring the proof of several hundred auxilliary lemmas, the results have been found to be quite useful, providing fairly quick derivations of conjectures that would otherwise admit only very complicated proofs.

Our interface to the Boyer-Moore system includes a parser that allows a concurrent program to be entered in the form of linear text (as in Figure 1), automatically encoded as a constant in the logic, and mechanically verified to satisfy our definitions. The generation of a term corresponding to a given conjectured property of a program, expressed as an instance of either (6) or (7), is then also performed automatically. Finally, this term is submitted to the theorem prover, along with a rule of inference selected according to its form. A hint facility allows the user to identify previously established properties to be used in a proof, as well as (in the case of eventuality properties) a relevant well-founded measure.

The value of this proof system lies in its utilization of the power of the Boyer-Moore prover through a scheme that shields its user from the details of its implementation. Thus, once the requisite arithmetic rewrite rules were established, the correctness of our example binomial coefficient program could have been verified by a user with no knowledge of either the internal representation of the program and its properties or the operation of the prover. It should be noted, however, that in order to guide our system effectively, one must begin

with a general notion of the structure of the proof of a given conjecture. The system can only be expected to relieve the user of the more tedious aspects of the proof process.

References

[BoM79] Boyer, R. S. and Moore, J S., A Computational Logic, Academic Press, New York, 1979.

[BoM88a] Boyer, R. S. and Moore, J S., A Computational Logic Handbook, Academic Press, Boston, 1988.

[BoM88b] Boyer, R. S. and Moore, J, *The Addition of Bounded Quantification and Partial Functions to A Computational Logic and its Theorem Prover*, Tech. Report ICSCA-CMP-52, Institute for Computing Science, U. of Texas at Austin, 1988.

[BGK89] Boyer, R. S., Goldschlag, D. M., Kaufmann, M., and Moore, J, *Functional Instantiation in First Order Logic*, Tech. Report 44, Computational Logic, Inc., Austin, TX, 1989.

[ChM88] Chandy, K. M. and Misra, J., Parallel Program Design: a Foundation, Addison-Wesley, Reading, MA, 1988.

[Gol89] Goldschlag, D. M., *A Mechanically Verified Proof System for Concurrent Programs*, Tech. Report 32, Computational Logic, Inc., Austin, TX, 1989.

[Kau86] Kaufmann, M., *A Formal Semantics and Proof of Soundness for the logic of the NQTHM Version of the Boyer-Moore Theorem Prover*, Internal Note 229, Institute for Computing Science, U. of Texas at Austin, 1986.

[Kro87] Kroger, F., Temporal Logic of Programs, Springer-Verlag, Berlin, 1987.

[MaP81] Manna, Z. and Pnueli, A., *Verification of Concurrent Programs: the Temporal Framework*, in The Correctness Problem in Computer Science, edited by Boyer, R. S. and Moore, J, Academic Press, London, 1981.

[MaP84] Manna, Z. and Pnueli, A., *Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs*, Science of Computer Programming 4 (1984), North-Holland.

[Rus90] Russinoff, D. M., *Verifying Concurrent Programs with the Boyer-Moore Prover*, Tech. Report STP/ACT-218-90, MCC, Austin, TX, 1990.

[Rus91] Russinoff, D. M., *A Mechanically Verified Incremental Garbage Collector*, Tech. Report STP-FT-148-91, MCC, Austin, TX, 1991.