Comparing Gypsy and the Boyer-Moore Logic for Specifying Secure Systems

Matt Kaufmann
William D. Young

Technical Report #59 May 1987

Also appears without appendices (and with inessential differences) in:
Proceedings of the 10th National Computer Security Conference

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

The Gypsy Verification Environment (GVE) [1, 2] is one of two systems endorsed by the National Computer Security Center for use in meeting the verification requirements for an A1 level evaluation as outlined in the *Trusted Computer Systems Evaluation Criteria* [3].* Gypsy has been used extensively in secure systems specification and verification projects including the Encrypted Packet Interface [4], Message Flow Modulator [5], Honeywell SCOMP [6], Honeywell SAT [7], and ACCAT Guard [8]. The Boyer-Moore theorem prover has also seen extensive use in the security arena. It has been used as a component of the HDM verification system [9] on KSOS [10], SCOMP, and SACDIN [11].

Yet the ways in which these two systems are currently used in secure system development efforts are quite different. The GVE is utilized as a fully integrated verification environment. The Gypsy language is used for constructing code and specification; verification conditions are generated and proved in the GVE proof checker; and, in some cases, the Gypsy code is compiled and run. The Boyer-Moore system, on the other hand, is used only as the proof checker for verification conditions generated from specifications written in some high level language such as Special. This is true despite the fact that the Boyer-Moore logic contains a fully executable functional programming language. The Boyer-Moore system has been used not only to state and prove theorems in traditional mathematical domains such as number theory and recursive function theory, but also to specify and prove the correctness of a microprocessor [12], and is being used to prove the correctness of an operating system and a compiler [13]. A main contention of this paper is that the Boyer-Moore logic can also be used effectively as a specification language for secure systems, particularly at the model level.

This paper investigates the viability of the Boyer-Moore logic as a specification language for secure system modeling efforts by comparing it to Gypsy on a significant example. The example we chose was the Low Water Mark problem, a simple secure system which has been used in two different studies [14, 15] for comparing verification systems. At least three different Gypsy specifications for this problem have been published [14, 16, 17]; our specification differs from each of these. Using a non-interference style characterization of security, we specified the Low Water Mark system in Gypsy and in the Boyer-Moore logic. The key security theorem was proved in each system using the associated proof-checker. We compare the specifications and proofs in the two languages, point out the advantages and disadvantages of each system, and investigate the possibility of defining a hybrid language which combines most of the advantages of each.

---

*The other is FDM, which will not be discussed further in this paper.

THE TWO LANGUAGES

Gypsy

Gypsy is a descendent of Pascal. It is a unified programming and specification language with facilities for exception handling, data abstraction, and concurrency. The specification component of the language contains the full expressive power of the predicate calculus. Specifications may be written as Floyd-Hoare style program annotations, algebraic-style axioms, or state machine descriptions.

Gypsy is a procedural language but contains a sizable functional component. The specification described in this paper is written entirely within the functional subset of the language. This is typical for abstract specifications. Implementations are usually given in a procedural style.

The following is a fully specified Gypsy implementation of a factorial routine. The *entry* and *exit* assertions in the definition of **F** give its specification. Notice that the function **fact** is a specification function against which the code is verified.

```
scope factorial_example =
begin

   function F (n: integer): integer =
   begin
      entry N ge 0;
      exit result = fact (N);
      var i: integer := 1;
      result := 1;
      loop
         result := result * i;
         if i = n
            then leave
            else i := i + 1
         end {if}
      end {loop}
   end; {function F}

   function fact (n: integer): integer =
   begin
      exit result =
              if n le 0
                 then 1
                 else n * fact (n-1)
              fi;
   end; {function fact}

end; {scope}
```

Gypsy is fully described in [1] and a methodology for using the language effectively is documented in [2].

The Boyer-Moore Logic

The Boyer-Moore logic is a quantifier-free constructive first-order logic with equality and rules for defining recursive functions. The language is a minor variant of Pure Lisp [18] and consists of variables and function names combined in a prefix notation. Predicates are represented as boolean-valued functions. Though untyped, the logic supports a restricted version of user-defined recursive data types (the "shell principle"). Despite the absence of quantifiers in the logic, the system allows one to prove lemmas that are, in effect, treated as universally quantified statements.

A Boyer-Moore specification of the factorial function has the form

Definition
```
(ZEROP N)
    =
(OR (EQUAL N 0)
    (NOT (NUMBERP N)))
```

Definition
```
(FACTORIAL N)
    =
(IF (ZEROP N)
    1
    (TIMES N (FACTORIAL (SUB1 N)))).
```

The Boyer-Moore *definition principle* guarantees that functions accepted by the Boyer-Moore system are total. Thus, `(FACTORIAL N)` is defined even if `N` is not a numeric argument. This contributes to a specification style which is quite different than that used in Gypsy. The logic is fully described in [19, 20].

The Main Differences

The primary differences in the two languages are summarized below.

1. Syntactically the two languages are quite different. Gypsy syntax is Pascal-like mixed infix/prefix notation; the Boyer-Moore logic uses a LISP-like prefix syntax.

2. Gypsy provides procedures, including concurrent procedures. The Boyer-Moore logic is purely functional.

3. Because of the procedural aspects of the language, the semantics of Gypsy [21] is significantly more complex than that of the Boyer-Moore logic. The Boyer-Moore logic has a simple applicative semantics; in a certain sense, an interpreter for the logic can be defined fairly easily within the logic.

4. Gypsy encourages a top-down development style by allowing the programmer to leave implementations pending. This permits references to and proofs about routines which are not fully elaborated. In the proof domain, there is no enforced constraint that lemmas be proved before they are used. The Boyer-Moore logic encourages a bottom-up development style since functions must be accepted before they can be referenced. Top-down development of proofs can be accomplished in the Boyer-Moore framework by adding lemmas as axioms and then later redoing the proof. However, this is counter to the paradigm of proof development in the Boyer-Moore system; it is assumed that lemmas will be proved before they are used.

5. Gypsy data typing restricts syntactically the passing of arguments to routines. However, there is at present no guarantee that routines will be defined even for arguments of permissible type. The Gypsy Verification

Environment supports *partial correctness* proofs; that is, proofs of termination must be performed outside the system. The Boyer-Moore definition principle guarantees that all functions are total. Arguments which are not of the expected *type* are usually treated as if they were. Thus the Boyer-Moore function **`FACTORIAL`** above treats a non-numeric argument as if it were zero.

6. The Gypsy specification language contains the universal and existential quantifiers. The Boyer-Moore logic is constructive and quantifier free. Lemmas involving variables are regarded as implicitly universally quantified. To obtain the effect of an existential quantifier it is necessary to provide a "witness function" which computes the required value.

THE LOW WATER MARK PROBLEM

The Low Water Mark problem was introduced in [14] and used there for a comparison of four verification systems. It was recently revived as one benchmark problem for a more extensive comparison of verification systems reported in [15]. In the context of that study, two distinct solutions to the problem were coded in Gypsy [16, 17]. We describe a third solution, coded in Gypsy and in the Boyer-Moore logic and fully verified in each of the two systems. The method of specification follows the *non-interference* approach described in the following section.

Cheheyl, *et al.* describe the Low Water Mark problem as follows:

> *The example system has at least one data object and three operations: READ, WRITE, and RESET. The operations are used by several processes having various fixed security levels. The system is required to satisfy the simple security and *-property. For simplicity the security levels are assumed to be linearly ordered.*
>
> *...The low water mark idea is that the data object has a security level that can decrease but not increase except via RESET. A decrease in level occurs when the object is (totally) rewritten by a lower level process. The new level of the object is the level of the calling process.*

The Bell and LaPadula simple security and *-properties [22] are following requirements: for a process to read an object the process level must dominate that of the object; to write, the object level must dominate that of the process. A read involves no change of levels for the object; a write causes the object level to drop to that of the process. Reset causes the object security level to become *system high* and the value of the object to become *undefined*.

Cheheyl, *et al.* require that the *dominates* relation on levels be a total ordering "for simplicity." However, Rushby [23] has shown that the system described is insecure if the levels are only partially ordered.

NON-INTERFERENCE

The security model for our solution to the low water mark problem is a non-interference model. The notion of non-interference assertions was developed by Goguen and Meseguer [24, 25] and elaborated upon by Rushby [26]. It provides a powerful and quite general mechanism for describing security policies. Non-interference has been applied successfully in the proof of certain security properties of the Honeywell Secure Ada Target machine [27].

Process p_1 is said to be *non-interfering* with process p_2 (denoted $p_1 \not\mapsto p_2$) if no instruction issued by p_1 can influence the future output of the system to p_2. A non-interference security policy is simply a set of assertions which characterize which interferences between processes in a system are prohibited (alternatively a binary relation on processs). This notion can be rendered more amenable to formal treatment by the following observation: for any sequence of operations *seq*,

$$p_1 \not\mapsto p_2 \equiv \text{View}_{p_2}(\text{seq}) = \text{View}_{p_2}(\text{seq}/p_1)$$

where $View_p(seq)$ is the complete picture that process p has of the state of the system, and *seq/p* is the subsequence of *seq* obtained by deleting those instructions executed by *p*.

Our non-interference policy is a simplification of the DoD MLS policy. Processes have associated levels which are related by a total order. Process p_1 may interfere with process p_2 only if $level(p_1) \leq level(p_2)$. That is, the security policy is characterized by the following set of non-interference assertions:

$$\{p_1 \not\mapsto p_2 : \neg\ \text{level}(p_1) \leq \text{level}(p_2)\}.$$

We notice that it is only the *levels* of individual processes that are relevant to security. Consequently, to demonstrate that security is maintained in the system it suffices to show, for an arbitrary process *p*, that no instruction executed on behalf of any process at a higher level can affect *p*'s view. That is,

$$\text{View}_p(\text{seq}) = \text{View}_p(\text{seq/level(p)}),$$

where *seq/level(p)* is the instruction sequence purged of all instructions executed on behalf of processes at levels not dominated by *level(p)*. It is proved in [27] that this policy implies both simple security and the *-property. Thus, it satisfies the constraints of the low water mark problem.

THE SPECIFICATIONS

In this section we give the Gypsy and Boyer-Moore specifications of the non-interference policy described in the preceding paragraph. For readability we write all Gypsy code in lower case and all Boyer-Moore code in upper case. Enough elaboration is given here (we hope) to give a clear idea of the nature and details of each of the two specifications. The reader interested in the complete set of definitions and lemmas is referred to the appendices. The proof logs for the Gypsy version appear in [28].

The Main Data Types

The notions (data types) of process, object, state, and instruction appear in each specification. To avoid entanglement in syntactic details, we give only an outline of the main data types here, namely state, level, and instruction sequence (and relevant auxiliary types). We refer the reader to the appendices if more details are desired.

Security States. Let us begin with Gypsy. In the Gypsy spec, an *object* is arbitrary; the *object* type is declared to be *pending*. Similarly, the *process* type is *pending*. However, the types *object_level_map*, *object_value_map*, and *process_values_map* are declared in order to specify a security state with the following (Pascal-like) type declaration:

```
type security_state =
   record (values_read: process_values_map;
           object_level: object_level_map;
           object_value: object_value_map);
```

The three types referred to in this record are declared as mappings: for example, one such declaration is

```
type process_values_map =
   mapping from process to value_sequence;
```

where one declares

```
type value_sequence = sequence of value_type;
```

Thus, the values read by process **`p`** are obtained by taking the **`values_read`** component of the state and then applying the resulting mapping to the process **`p`**:

```
state.values_read[p]
```

The Boyer-Moore type declarations are quite similar, except that there are no "mapping types". Let us begin with *processes*. Thus, for example, the values read by a process are a component of the process rather than the result of applying a function to the process. The following syntax is simply a declaration of a process as a record type with two fields. Thus if **`(PROCESSP X)`** is true, then **`(PROC-NAME X)`** equals the value of the first field and **`(VALUES-READ X)`** equals the value of the second field. Conversely, if **`ProcName`** and **`ValRead`** are these two respective values, then **`X`** = **`(PROCESS ProcName ValRead)`**, i.e. **`PROCESS`** is actually a function which constructs a process from a name and some values-read.

Shell Definition
```
Add the shell PROCESS of two arguments
with recognizer PROCESSP
and accessors PROC-NAME and VALUES-READ
```

(**Remark** for readers familiar with the Boyer-Moore logic. The reader may notice that we have omitted declarations of the type restrictions, default values, and bottom object from the following shell definition. In fact these are *none*, **`ZERO`**, and *none*, respectively. Similar simplifications will be made in the other shell definitions presented below. The actual events appear in the second appendix below.)

In the Boyer-Moore specification, it was convenient to choose to access the values-read as a function of the *name* of a process rather than of the process itself. An advantage to this approach is that it is absolutely clear that the level of a process does not change during the execution of instructions; however, this is also a disadvantage since the model is less general. At any rate, in the Boyer-Moore case, one reads the values from a process name as follows:

```
(VALUES-READ (GET-PROCESS P-NAME STATE))
```

where **VALUES-READ** (defined above) is an accessor for processes and **GET-PROCESS** is a recursively defined function:

Definition

```
(GET-PROCESS P-NAME PROCESSES)
      =
(IF (NOT (LISTP PROCESSES))
    F
    (IF (EQUAL P-NAME (PROC-NAME (CAR PROCESSES)))
        (CAR PROCESSES)
        (GET-PROCESS P-NAME (CDR PROCESSES))))
```

Notice that this function is necessary in the Boyer-Moore version because the Boyer-Moore logic does not support mapping types. That is, functions must be defined in the logic rather than being data objects (such as the object *state.values_read* in Gypsy).

So, we have defined the notion of process for the Boyer-Moore version. The notion of *object* is similar, and we omit details:

Shell Definition

```
Add the shell OBJECT of three arguments
with recognizer OBJECTP
and accessors OBJ-NAME, VALUE, and OLEVEL
```

We may now define a state to be simply a record consisting of a list of processes and a list of objects.

Shell Definition

```
Add the shell STATE of two arguments
with recognizer STATEP
and accessors PROCESSES and OBJECTS
```

Actually no restriction is made on the components of a state, because of the weakness of the Boyer-Moore type (shell) mechanism. Instead, a predicate **PROPER-STATEP** is defined which restricts the class of "states" for the theorem ultimately proved. Thus the following function (predicate) returns **T** (true) exactly when its argument is a state whose components are respectivly a list of processes and a list of states. (Thus the functions **PROCESS-LISTP** and **OBJECT-LISTP** are defined first; however, we omit their straightforward definitions here.)

Definition

```
(PROPER-STATEP STATE)
        =
  (AND (STATEP STATE)
       (PROCESS-LISTP (PROCESSES STATE))
       (OBJECT-LISTP (OBJECTS STATE)))
```

This kind of treatment of states is necessary because the **sequence of** type constructor of Gypsy (used above in the declaration of *value_sequence*) has no real analogue in the Boyer-Moore logic.

Levels. The notion of *level* is specified in each language as a function (which is left undefined). However, as mentioned above, we chose to have the function depend on the process in the Gypsy version but on the process *name* in the Boyer-Moore version.

```
function process_level (p: process): level_type = pending;
```

Declaration
(PLEVEL PROC-NAME) = *[unspecified]*

There is one other difference in the handling of levels by the two versions. The Gypsy version (which was done first) used the integer ordering functions in order to order the levels, thus treating **level_type** (see above) as though it were the type of integers. This had the advantage of allowing the Gypsy simplifier to contribute its built-in knowledge of integers (and their ordering) to the proof. However, the Boyer-Moore version was undertaken with the goal of allowing an arbitrary total order, **DOMINATES**, with the axioms for a total order included. This was indeed accomplished, and no other axioms were needed except that the "system high" level is the greatest level: **(DOMINATES (SYSTEM-HIGH) L)**. (A similar axiom was added for the Gypsy proof as well.)

Instruction Sequences. The notion of instruction is defined as a record in each language, with fields corresponding roughly to the type (i.e. read, write, or reset), process, object, and value. (Of course, the value field is not necessary for a *read* instruction; it is simply ignored.)

```
type instruction_class = (rd, wrt, rst);

type instruction = record (class: instruction_class;
                           p: process;
                           o: object;
                           v: value_type)
```

Shell Definition
Add the shell MAKE-INSTRUCTION of four arguments
with recognizer INSTRUCTIONP
and accessors TYPE, I-PROC-NAME, I-OBJ-NAME, and I-VALUE

The notion of instruction sequence is declared as a type in Gypsy but is defined by a recursive function in the Boyer-Moore logic (again, because there is no sequence type constructor in that logic):

```
type instruction_sequence = sequence of instruction;
```

Definition

```
(INSTRUCTION-LISTP LST)
      =
(IF (NOT (LISTP LST))
    T
    (AND (INSTRUCTIONP (CAR LST))
         (INSTRUCTION-LISTP (CDR LST))))
```

The Main Theorem

Following the simple security and *-properties mentioned earlier, we imagine executing instructions which request reads, writes, and resets, where "illegal" requests are ignored. Thus a process may not read an object at a strictly higher level or write to (or reset) an object at a strictly lower level. Let us begin by stating the main security theorem in each of the two languages. We will then give definitions of functions used in these statements, including *interpret*, which runs the given instruction sequence on the given state (to return a new state), and *purge*, which removes all instructions whose level exceeds the level of the given process. In the Gypsy version, one considers the equality of the values read by a given process in the following two states: the state obtained after running the original instructions, and the state obtained after running the purged instructions. In the Boyer-Moore version, we have chosen to consider these two processes rather than just the values that they have read, since a process is merely a name together with those values. Either way, the definitions of the *purge* function guarantee security of the system because the purged instructions have no effect on the process's view of the system.

```
lemma system_is_secure (inseq: instruction_sequence;
                        state: security_state;
                        p: process) =
     interpret (inseq, state).values_read[p]
   = interpret (purge (inseq, process_level (p)),
                state).values_read[p];
```

Lemma *(SYSTEM-IS-SECURE)*.

```
(IMPLIES
  (AND (PROPER-STATEP ST)
       (INSTRUCTION-LISTP INSTLIST))
  (EQUAL
    (GET-PROCESS P-NAME
                 (PROCESSES (INTERPRET INSTLIST ST)))
    (GET-PROCESS P-NAME
                 (PROCESSES (INTERPRET (PURGE INSTLIST
                                              (PLEVEL P-NAME))
                                       ST)))))
```

The function *interpret* takes a sequence of instructions and an initial state and returns a new state. It is in turn defined in terms of a "single stepper" which interprets a single instruction. Notice that the Gypsy definition of *interpret* recursively decomposes the instruction sequence from the right while the Boyer-Moore definition works from the left. Gypsy syntax supports accessing sequences from either end; Boyer-Moore's LISP-like style strongly favors recursively decomposing lists from the left. The Boyer-Moore version of *interpret* "runs" the given instructions in the reverse of the order in which the Gypsy version "runs" the instructions.

```
function single_step (i: instruction;
                      state: security_state): security_state =
begin
   exit result
         = if i.class = rd
              then read (i.p, i.o, state)
              else if i.class = wrt
                      then write (i.p, i.o, i.v, state)
                      else reset (i.p, i.o, state)
                   fi
           fi;
    pending
end;

function interpret (inseq: instruction_sequence;
                    state: security_state): security_state =
begin
   exit result
         = if inseq = null (instruction_sequence)
              then state
              else single_step (last (inseq),
                                interpret (nonlast (inseq), state))
           fi;
    pending
end;
```

Definition

```
(SINGLE-STEP INST ST)
        =
(IF (GET-OBJECT (I-OBJ-NAME INST) (OBJECTS ST))
    (IF (EQUAL (TYPE INST) 'READ)
        (READ (I-PROC-NAME INST) (I-OBJ INST ST) ST)
        (IF (EQUAL (TYPE INST) 'WRITE)
            (WRITE (PLEVEL (I-PROC-NAME INST))
                   (I-OBJ INST ST)
                   (I-VALUE INST)
                   ST)
            (RESET (PLEVEL (I-PROC-NAME INST))
                   (I-OBJ INST ST) ST)))
    ST)
```

Definition

```
(INTERPRET INSTLIST ST)
       =
(IF (NOT (LISTP INSTLIST))
    ST
    (SINGLE-STEP (CAR INSTLIST)
                 (INTERPRET (CDR INSTLIST) ST)))
```

The auxiliary *read*, *write*, and *reset* functions take a security state together with other appropriate arguments (such as process or its level, object, and value), and return a new state. However, the state is unchanged if the relevant levels are inappropriate. For example, here are the two definitions of *write*. Notice that the Gypsy syntax is richer in that its **with** construct allows a convenient notation for updating specified fields of a record.

```
function write (p: process; o: object;
                v: value_type;
                state: security_state): security_state =
begin
   exit result
         = if process_level (p) le state.object_level[o]
              then state with (.object_value[o] := v;
                               .object_level[o] :=
                                       process_level (p))
              else state
           fi;
    pending
end; {write}
```

Recall below that **PROCESSES** picks out the **PROCESSES** field of the given state, and similarly for **OBJECTS**.

Definition

```
(WRITE LEVEL O V ST)
       =
(IF (DOMINATES (OLEVEL O) LEVEL)
    (STATE
      (PROCESSES ST)
      (REWRITE-OBJECT (OBJ-NAME O) V LEVEL (OBJECTS ST)))
    ST)
```

where **(REWRITE-OBJECT O-NAME V L OBJECTS)** returns the result of replacing the value of the object named **O-NAME** with **V** and the level with **L**, in the given list of **OBJECTS**. (We omit the recursive definition of **REWRITE-OBJECT**.)

It remains to define *purge*. The "values read" component of the new state contains, for each process **P**, the sequence of values received by **P** as a result of the **READ** instructions executed on its behalf. In our model, this is the only information that a process can obtain about the system. Thus, the following are the definitions of *purge*. Notice that the Boyer-Moore function is defined for a much broader collection of arguments. The type-free nature of the logic and the requirement that all functions be total requires that **PURGE** be defined on arguments which are intuitively quite different than the intended "argument types."

```
function purge (inseq: instruction_sequence;
                l: level_type): instruction_sequence =
begin
   exit (assume result =
           if inseq = null (instruction_sequence)
              then null (instruction_sequence)
              else if l ge process_level (last (inseq).p)
                      then      purge (nonlast (inseq), l)
                             <: last (inseq)
                      else purge (nonlast (inseq), l)
                   fi
           fi);
end; {purge}
```

Definition

```
(PURGE INSTLIST LEVEL)
       =
(IF (NOT (LISTP INSTLIST))
    INSTLIST
    (IF (DOMINATES LEVEL
                   (PLEVEL (I-PROC-NAME (CAR INSTLIST))))
        (CONS (CAR INSTLIST)
              (PURGE (CDR INSTLIST) LEVEL))
        (PURGE (CDR INSTLIST) LEVEL)))
```

OUTLINES OF THE PROOFS

The main lemmas for the proofs are similar. In each case, the idea is to proceed by some kind of induction on the length of the instruction list. However, in order to obtain a sufficiently strong inductive hypothesis it is desirable to prove a somewhat stronger result than the main theorem (which was called "System Is Secure" above), from which the main theorem follows immediately. The stronger result says that the given process has the same view of the two relevant states, namely the ones obtained with and without running the purged instructions.

```
function process_view_identical
    (p: process;
     state1, state2: security_state)
  : boolean =
begin
   exit (result =
           (  state1.values_read[p] = state2.values_read[p]
            & (all o1: object,
                      o1 in could_read (p, state1)
                 iff o1 in could_read (p, state2))
            & (all o2: object,
                      o2 in could_read (p, state1)
                ->      state1.object_level[o2]
                      = state2.object_level[o2]
                   &    state1.object_value[o2]
                      = state2.object_value[o2])));
end; {process_view_identical}
```

where the function `could_read` returns the set of objects that can be read by the given process:

```
function could_read
   (p: process;
    state: security_state): object_set =
begin
   exit (all o: object,
                o in result
            iff process_level (p) ge state.object_level[o]);
end; {could_read}


lemma purge_preserves_process_view
   (inseq: instruction_sequence;
    state: security_state;
    p: process) =
  process_view_identical
    (p, interpret (inseq, state),
        interpret (purge (inseq, process_level (p)),
                   state));
```

And now the Boyer-Moore version:

Definition

```
(PROCESS-VIEW-IDENTICAL P-NAME ST1 ST2)
       =
(AND (EQUAL (GET-PROCESS P-NAME (PROCESSES ST1))
            (GET-PROCESS P-NAME (PROCESSES ST2)))
     (OBJECT-NAMES-AGREE (OBJECTS ST1) (OBJECTS ST2))
     (OBJECTS-MATCH-BELOW-LEVEL (PLEVEL P-NAME)
                                (OBJECTS ST1)
                                (OBJECTS ST2)))
```

where **OBJECT-NAMES-AGREE** returns **T** (true) when the two object lists have the same names (in the same order) and **OBJECTS-MATCH-BELOW-LEVEL** implies that the given object lists agree when restricted to objects below the given level:

Definition

```
(OBJECTS-MATCH-BELOW-LEVEL LEVEL OBJS1 OBJS2)
      =
(IF (AND (LISTP OBJS1) (LISTP OBJS2))
    (IF (OR (DOMINATES LEVEL (OLEVEL (CAR OBJS1)))
            (DOMINATES LEVEL (OLEVEL (CAR OBJS2))))
        (AND (EQUAL (CAR OBJS1) (CAR OBJS2))
             (OBJECTS-MATCH-BELOW-LEVEL
               LEVEL (CDR OBJS1) (CDR OBJS2)))
        (OBJECTS-MATCH-BELOW-LEVEL
          LEVEL (CDR OBJS1) (CDR OBJS2)))
    (AND (NOT (LISTP OBJS1))
         (NOT (LISTP OBJS2))))
```

Thus we are brought to the Boyer-Moore version of the main lemma:

Lemma *(PURGE-PRESERVES-PROCESS-VIEW)*.

```
(IMPLIES
  (AND (PROPER-STATEP ST)
       (INSTRUCTION-LISTP INSTLIST))
  (PROCESS-VIEW-IDENTICAL
    P-NAME
    (INTERPRET INSTLIST ST)
    (INTERPRET (PURGE INSTLIST (PLEVEL P-NAME))
               ST)))
```

In both versions, there are two main sublemmas used for proving the inductive step of this lemma, i.e. for proving (roughly) that the lemma remains true when one considers one more instruction on the given list of instructions. The first lemma treats the case that the added instruction has a level which is higher than that of the given process (and may therefore be purged):

```
lemma
  purgeable_instruction_preserves_process_view
     (a: instruction;
      p: process;
      state1, state2: security_state) =
      not process_level (a.p) le process_level (p)
    & process_view_identical (p, state1, state2)
  ->  process_view_identical
        (p, single_step (a, state1), state2);
```

Lemma
(PURGEABLE-INSTR-PRESERVES-PROCESS-VIEW).

```
(IMPLIES
  (AND (PROPER-STATEP ST1)
       (PROPER-STATEP ST2)
       (INSTRUCTIONP INST)
       (PROCESS-VIEW-IDENTICAL P-NAME ST1 ST2)
       (NOT (DOMINATES (PLEVEL P-NAME)
                       (PLEVEL (I-PROC-NAME INST)))))
  (PROCESS-VIEW-IDENTICAL P-NAME
                          (SINGLE-STEP INST ST1)
                          ST2))
```

The other lemma treats the other case, i.e. where the added instruction is not purged:

```
lemma
 nonpurgeable_single_step_preserves_process_view
          (p: process;
           a: instruction;
           state1, state2: security_state) =
      process_level (a.p) le process_level (p)
    & process_view_identical (p, state1, state2)
 -> process_view_identical
      (p, single_step (a, state1), single_step (a, state2));
```

Lemma
(NONPURGEABLE-INSTRUCTION-PRESERVES-PROCESS-VIEW)

```
(IMPLIES
  (AND (PROPER-STATEP ST1)
       (PROPER-STATEP ST2)
       (PROCESS-VIEW-IDENTICAL P-NAME ST1 ST2)
       (DOMINATES (PLEVEL P-NAME)
                  (PLEVEL (I-PROC-NAME INST))))
  (PROCESS-VIEW-IDENTICAL P-NAME
                          (SINGLE-STEP INST ST1)
                          (SINGLE-STEP INST ST2)))
```

Both proofs use a number of additional subsidiary definitions and lemmas. However, the Boyer-Moore system certainly provides a much more powerful level of automatic support.

COMPARING THE TWO METHODOLOGIES

We compare the following aspects of the Gypsy and Boyer-Moore methodologies:

- **Specification style**
- **Proof management**
- **Style of interaction with the prover**
- **Soundness**

Specification Style

Gypsy is a rich language in that it has sets and functions as first-class data objects, first-order quantifiers, and an expressive typing discipline for user-defined types. The Boyer-Moore logic does not have these features, but its solid treatment of recursion and lists, along with its capability for introducing data types with the so-called shell principle, allows one sufficient specification power. Gypsy also is richer in that it has procedural constructs, though for high-level specs (such as the Low Water Mark example) users of Gypsy have found it advantageous to use a functional style. In spite of the different notions of data type and the greater expressive power of the Gypsy language, the specs are clearly quite similar for the two versions of the Low Water Mark example that are presented here.

The issue of types deserves further comment. Consider the following (incorrect) statement of the main theorem in the Boyer-Moore logic. Sadly, an earlier version of our specification contained this misstatement. The reader is invited to find the error before reading further.

Lemma *(SYSTEM-IS-SECURE)*.

```
(IMPLIES
  (AND (PROPER-STATEP ST)
       (INSTRUCTION-LISTP INSTLIST))
  (EQUAL
    (GET-PROCESS P-NAME
                 (INTERPRET INSTLIST ST))
    (GET-PROCESS P-NAME
                 (INTERPRET (PURGE INSTLIST (PLEVEL P-NAME))
                            ST))))
```

The problem with this statement is that **GET-PROCESS** is defined so that its second argument is (expected to be) a list of processes, not a state. In fact, the two sides of the equality above are actually both provably equal to **F** (false), under the given hypotheses! An analogous error in the Gypsy text would have been caught by the type-checker. However, the problem of matching formal specifications to intuitive requirements remains a central issue in program verification research.

Proof Management

The Gypsy system allows one to defer the proofs of lemmas during a proof session. This capability is amenable to a top-down proof style which is quite natural. The Boyer-Moore system allows one to add axioms, which enables one to have the same top-down capability to some extent; one simply assumes seemingly necessary lemmas before proving the main result, and then one goes back and proves those supporting facts. However, that strategy is awkward with the Boyer-Moore system since event histories are totally ordered.

Style of Interaction with the Prover

The Boyer-Moore prover is much more powerful than is the Gypsy prover, and thus allows much larger proof steps and is significantly less tedious to operate. Even though the two verifications discussed here each contain about thirty lemmas, the Boyer-Moore prover proved each of those lemmas automatically (occasionally with some simple hints supplied with the statements of the lemmas), while the Gypsy prover required considerable tedious interaction in order to prove many of the lemmas. However, the powerful heuristics and rule-based rewriting capabilities of the Boyer-Moore prover also make its behavior somewhat unpredictable and also quite difficult and frustrating to control at times, though it prints out useful information to help discover what additional lemmas are needed. The level of interaction is not the only significant difference in the style of interaction. It is much easier with the Boyer-Moore system to modify an existing proof and replay the resulting definition and proof commands. But as mentioned above, the Gypsy system is much more flexible about the order in which one gives proofs.

Soundness

The Boyer-Moore logic, as described completely in [20] and Chapter 3 of [19], has simple and well-understood operational and denotational semantics in which every proved theorem is in fact true. Unfortunately, the same cannot be said of Gypsy. Moreover, the Gypsy system does not have a mechanism for ensuring that all lemmas have been proved, nor does it guarantee that circular arguments (in which lemmas use each other for their proofs) are not constructed. Finally, there is empirical evidence over 15 years for virtually bug-free performance of the Boyer-Moore implementation that has not been matched by the Gypsy implementation.

CONCLUSIONS

Despite certain shortcomings, we believe that the Boyer-Moore logic provides a reasonable specification alternative for secure systems, particularly at the model level. Soundness of the logic and the care with which it is implemented in the theorem prover are strong advantages of the Boyer-Moore system over Gypsy or other currently available verification systems. Gypsy, on the other hand, is an expressive and versatile language which provides the benefits of data types, data abstraction, concurrency, conditional handling, procedural semantics, etc.

We believe that the relative strengths of the Gypsy and Boyer-Moore systems are in fact quite compatible. Work is already underway to remedy some of the soundness deficiencies of Gypsy, and a preliminary system for the Boyer-Moore logic has been constructed [29] that uses the Boyer-Moore prover as a component but also allows more user control. Moreover, a form of quantification has already been added to the Boyer-Moore logic and prover [30] and plans are under way to add sets, full first-order quantification, and more flexible structuring of proofs. Preliminary investigation has also begun into developing a Gypsy-like syntax for the Boyer-Moore logic. This "merger" of Gypsy and the Boyer-Moore systems is the subject of an active research effort at Computational Logic, Inc., with the intended result to be called *Rose*. The primary component omitted from Rose is expected to be the procedural part of Gypsy. The hope is that technology will continue to be developed toward obtaining efficient implementations of functional languages, particularly with respect to seeming opportunities for concurrent evaluation.

Our experiences to date suggest that a fundamental requirement for any successful verification is that the person(s) doing the verification understand the theorems to be proved. A system with Gypsy's flexibility of language and use and with the Boyer-Moore system's proof power and clear semantics would be a great aid in this respect.

ACKNOWLEDGEMENTS

We thank Dave Plummer of the University of Texas for helpful comments. We also thank Tom Haigh of Honeywell Secure Computing Technology Center for co-development of some of the ideas on non-interference (cf. [27]). Finally, we thank our colleagues at the Institute for Computing Science at UT and Computational Logic Inc. for a stimulating environment.

Appendix A

Gypsy Text of the Low Water Mark Example, from [31]

```
scope secure_system =
 begin

   type process = pending;

   function process_level (p: process): level_type = pending;

   type level_type = pending;

   const system_high: level_type = pending;

   lemma system_high_dominates =
      all l: level_type, l le system_high;

   type value_type = pending;

   type value_sequence = sequence of value_type;

   type process_values_map
                = mapping from process to value_sequence;

   const undefined_value: value_type = pending;

   type object = pending;

   type object_level_map = mapping from object to level_type;

   type object_value_map = mapping from object to value_type;

   type security_state = record (values_read: process_values_map;
                                 object_level: object_level_map;
                                 object_value: object_value_map);

   function read (p: process; o: object;
                  state: security_state): security_state =
   begin
      exit result
          = if process_level (p) ge state.object_level[o]
               then state with (.values_read[p] := state.values_read[p]
                                             <: state.object_value[o])
               else state
            fi;
       pending
   end; {read}
```

```
function write (p: process; o: object;
                v: value_type;
                state: security_state): security_state =
begin
   exit result
         = if process_level (p) le state.object_level[o]
              then state with (.object_value[o] := v;
                               .object_level[o] := process_level (p))
              else state
           fi;
    pending
end; {write}

function reset (p: process; o: object;
                state: security_state): security_state =
begin
   exit result
         = if process_level (p) le state.object_level[o]
              then state with (.object_value[o] := undefined_value;
                               .object_level[o] := system_high)
              else state
           fi;
    pending
end; {reset}

type instruction_class = (rd, wrt, rst);

type instruction = record (class: instruction_class;
                           p: process;
                           o: object;
                           v: value_type);

type instruction_sequence = sequence of instruction;

function single_step (i: instruction;
                      state: security_state)
                : security_state =
begin
   exit (assume result =
            if i.class = rd
               then read (i.p, i.o, state)
               else if i.class = wrt
                       then write (i.p, i.o, i.v, state)
                       else reset (i.p, i.o, state)
                    fi
            fi);
end; {single_step}
```

```
function interpret (inseq: instruction_sequence;
                    state: security_state)
               : security_state =
begin
   exit (assume result =
            if inseq = null (instruction_sequence)
               then state
               else single_step (last (inseq),
                                 interpret (nonlast (inseq), state))
            fi);
end; {interpret}

function purge (inseq: instruction_sequence;
                l: level_type): instruction_sequence =
begin
   exit (assume result =
           if inseq = null (instruction_sequence)
              then null (instruction_sequence)
              else if l ge process_level (last (inseq).p)
                        then      purge (nonlast (inseq), l)
                               <: last (inseq)
                        else purge (nonlast (inseq), l)
                   fi
            fi);
end; {purge}

type object_set = set of object;

function could_read (p: process;
                     state: security_state): object_set =
begin
   exit (assume all o: object,
             o in result
         iff process_level (p) ge state.object_level[o]);
end; {could_read}

function process_view_identical (p: process;
                                 state1, state2: security_state)
                     : boolean =
begin
   exit (assume result =
           (  state1.values_read[p] = state2.values_read[p]
            & (all o1: object,
                     o1 in could_read (p, state1)
                 iff o1 in could_read (p, state2))
            & (all o2: object,
                   o2 in could_read (p, state1)
                ->     state1.object_level[o2]
                     = state2.object_level[o2]
                   &   state1.object_value[o2]
                     = state2.object_value[o2])));
end; {process_view_identical}
```

```
lemma system_is_secure (inseq: instruction_sequence;
                        state: security_state;
                        p: process) =
     interpret (inseq, state).values_read[p]
   = interpret (purge (inseq, process_level (p)),
                state).values_read[p];

lemma purge_preserves_process_view (inseq: instruction_sequence;
                                    state: security_state;
                                    p: process) =
   process_view_identical
         (p, interpret (inseq, state),
             interpret (purge (inseq, process_level (p)), state));

lemma purgeable_instruction_preserves_process_view
                                (a: instruction;
                                 p: process;
                                 state1, state2: security_state) =
        not process_level (a.p) le process_level (p)
       & process_view_identical (p, state1, state2)
   -> process_view_identical (p, single_step (a, state1), state2);

lemma single_step_preserves_process_view
                                (p: process;
                                 a: instruction;
                                 state1, state2: security_state) =
      process_view_identical (p, state1, state2)
   -> process_view_identical (p, single_step (a, state1),
                               single_step (a, state2));

lemma reflexivity_of_process_view_identical (p: process;
                                             state: security_state) =
   process_view_identical (p, state, state);

lemma symmetry_of_process_view_identical
                                 (p: process;
                                  state1, state2: security_state) =
      process_view_identical (p, state1, state2)
   -> process_view_identical (p, state2, state1);

lemma transitivity_of_process_view_identical
                          (p: process;
                           state1, state2, state3: security_state) =
        process_view_identical (p, state1, state2)
       & process_view_identical (p, state2, state3)
   -> process_view_identical (p, state1, state3);

lemma purgeable_single_step_preserves_process_view
                                  (p: process;
                                   a: instruction;
                                   state1, state2: security_state) =
        not process_level (a.p) le process_level (p)
       & process_view_identical (p, state1, state2)
   -> process_view_identical (p, single_step (a, state1),
                               single_step (a, state2));
```

```
lemma nonpurgeable_single_step_preserves_process_view
                                  (p: process;
                                   a: instruction;
                                   state1, state2: security_state) =
        process_level (a.p) le process_level (p)
       & process_view_identical (p, state1, state2)
   -> process_view_identical (p, single_step (a, state1),
                               single_step (a, state2));

lemma nonpurgeable_read_preserves_process_view
                                   (p: process;
                                    a: instruction;
                                    state1, state2: security_state) =
        process_level (a.p) le process_level (p)
       & process_view_identical (p, state1, state2)
   -> process_view_identical (p, read (a.p, a.o, state1),
                               read (a.p, a.o, state2));

lemma read_preserves_process_view (p: process;
                                    o: object;
                                    state1, state2: security_state) =
      process_view_identical (p, state1, state2)
   -> process_view_identical (p, read (p, o, state1),
                                 read (p, o, state2));

lemma nonpurgeable_write_preserves_process_view
                                    (p: process;
                                     a: instruction;
                                     state1, state2: security_state) =
        process_level (a.p) le process_level (p)
       & process_view_identical (p, state1, state2)
   -> process_view_identical (p, write (a.p, a.o, a.v, state1),
                               write (a.p, a.o, a.v, state2));

lemma nonpurgeable_reset_preserves_process_view
                               (p: process;
                                a: instruction;
                                state1, state2: security_state) =
        process_level (a.p) le process_level (p)
       & process_view_identical (p, state1, state2)
   -> process_view_identical (p, reset (a.p, a.o, state1),
                               reset (a.p, a.o, state2));

lemma others_reads_dont_affect_process_view (p, p2: process;
                                             o: object;
                                             state: security_state) =
   p ne p2 -> process_view_identical (p, state, read (p2, o, state));

lemma others_resets_dont_affect_process_view
                                (p, p2: process;
                                 o: object;
                                 state: security_state) =
      not process_level (p2) le process_level (p)
   -> process_view_identical (p, state, reset (p2, o, state));
```

```
lemma others_writes_dont_affect_process_view
                                (p, p2: process;
                                 o: object;
                                 v: value_type;
                                 state: security_state) =
      not process_level (p2) le process_level (p)
   -> process_view_identical (p, state, write (p2, o, v, state));

lemma dominated_objects_in_could_read (o: object;
                                      p: process;
                                      state: security_state) =
      state.object_level[o] le process_level (p)
   -> o in could_read (p, state);

lemma process_view_preserves_dominated_values
                               (o: object;
                                p: process;
                                state1, state2: security_state) =
        state1.object_level[o] le process_level (p)
       & process_view_identical (p, state1, state2)
   ->   state1.object_level[o] = state2.object_level[o]
       & state1.object_value[o] = state2.object_value[o];

lemma process_view_partitions_levels
                            (o: object;
                             p: process;
                             state1, state2: security_state) =
      process_view_identical (p, state1, state2)
   -> (    state1.object_level[o] le process_level (p)
        iff state2.object_level[o] le process_level (p));

lemma changing_object_preserves_could_read
                               (o, o2: object;
                                p: process;
                                v: value_type;
                                l: level_type;
                                state1, state2: security_state) =
      (o in could_read (p, state1) iff o in could_read (p, state2))
   -> (       o
            in could_read (p,
                           state1 with (.object_value[o2] := v;
                                        .object_level[o2] := l))
        iff    o
             in could_read (p,
                            state2 with (.object_value[o2] := v;
                                         .object_level[o2] := l)));

lemma changing_values_read_preserves_view
                           (p, p2: process;
                            v: value_type;
                            state1, state2: security_state) =
      process_view_identical (p, state1, state2)
   -> process_view_identical (p, state1 with (.values_read[p2]
                                  := state1.values_read[p2] <: v),
                                 state2 with (.values_read[p2]
                                  := state2.values_read[p2] <: v));
```

```
lemma changing_object_preserves_view (p: process;
                                   o: object;
                                   v: value_type;
                                   l: level_type;
                                   state1, state2: security_state) =
   process_view_identical (p, state1, state2)
->
   process_view_identical (p,
                           state1 with (.object_value[o] := v;
                                        .object_level[o] := l),
                           state2 with (.object_value[o] := v;
                                        .object_level[o] := l));

lemma dominated_levels_preserved  (p: process;
                                   o: object;
                                   l: level_type;
                                   state1, state2: security_state) =
      process_view_identical (p, state1, state2)
    & l le process_level (p)
    & l le state1.object_level[o]
->
      l le state2.object_level[o];

lemma dominated_levels_preserved2  (p: process;
                                    o: object;
                                    l: level_type;
                                    state1, state2: security_state) =
      process_view_identical (p, state1, state2)
    & l le process_level (p)
    & l > state1.object_level[o]
->
      l > state2.object_level[o];

lemma dominated_levels_preserved3  (p: process;
                                   o: object;
                                   l: level_type;
                                   state1, state2: security_state) =
      process_view_identical (p, state1, state2)
    & l le process_level (p)
    & l < state1.object_level[o]
->
      l < state2.object_level[o];

{ The following are basis lemmas required because the theorem prover
  does not handle relational operations on pending types completely. }

lemma levels_transitive2 (l1, l2, l3: level_type) =
   l1 le l2 & l2 < l3 -> l1 < l3;

lemma levels_transitive3 (l1, l2, l3: level_type) =
   l1 le l2 & l2 le l3 -> l1 le l3;

lemma levels_le_lemma (l1, l2: level_type) =
   l1 < l2 -> l1 le l2;
```

```
  lemma less_not_equal_lemma (l1, l2: level_type) =
     l1 < l2 -> l1 ne l2;

end;
```

Appendix B

Boyer-Moore Text of the Low Water Mark Example

```
(ADD-SHELL PROCESS NIL
           PROCESSP
           ((PROC-NAME (NONE-OF) ZERO)
            (VALUES-READ (NONE-OF) ZERO)))

; I want the instruction to be purely syntactic and also for PURGE
; to be state-independent.  Thus I have the level depending purely
; on the name of the process:

(DCL PLEVEL (PROC-NAME))

(DEFN GET-PROCESS
      (P-NAME PROCESSES)
      (IF (NOT (LISTP PROCESSES))
          F
          (IF (EQUAL P-NAME
                     (PROC-NAME (CAR PROCESSES)))
              (CAR PROCESSES)
              (GET-PROCESS P-NAME
                           (CDR PROCESSES)))))

(DEFN PROCESS-LISTP
      (PL)
      (IF (NOT (LISTP PL))
          T
          (AND (PROCESSP (CAR PL))
               (PROCESS-LISTP (CDR PL)))))

(ADD-SHELL OBJECT NIL
           OBJECTP
           ((OBJ-NAME (NONE-OF) ZERO)
            (VALUE (NONE-OF) ZERO)
            (OLEVEL (NONE-OF) ZERO)))

(DEFN GET-OBJECT
      (O-NAME OBJECTS)
      (IF (NOT (LISTP OBJECTS))
          F
          (IF (EQUAL O-NAME
                     (OBJ-NAME (CAR OBJECTS)))
              (CAR OBJECTS)
              (GET-OBJECT O-NAME (CDR OBJECTS)))))

(DEFN OBJECT-LISTP
      (OL)
      (IF (NOT (LISTP OL))
          T
          (AND (OBJECTP (CAR OL))
               (OBJECT-LISTP (CDR OL)))))
```

```
(ADD-SHELL STATE NIL
           STATEP
           ((PROCESSES (NONE-OF) ZERO)
            (OBJECTS (NONE-OF) ZERO)))

(DEFN PROPER-STATEP
      (STATE)
      (AND (STATEP STATE)
           (PROCESS-LISTP (PROCESSES STATE))
           (OBJECT-LISTP (OBJECTS STATE))))

(DCL DOMINATES (L1 L2))

(ADD-AXIOM DOMINATES-IS-REFLEXIVE
           (REWRITE)
           (DOMINATES L L))

(ADD-AXIOM DOMINATES-IS-TRANSITIVE
           (REWRITE)
           (IMPLIES (AND (DOMINATES L1 L2)
                         (DOMINATES L2 L3))
                    (DOMINATES L1 L3)))

(ADD-AXIOM DOMINATES-IS-ANTISYMMETRIC
           (REWRITE)
           (IMPLIES (AND (DOMINATES L1 L2)
                         (DOMINATES L2 L1))
                    (EQUAL (EQUAL L1 L2) T)))

(ADD-AXIOM DOMINATES-IS-TOTAL
           (REWRITE)
           (IMPLIES (NOT (DOMINATES L1 L2))
                    (DOMINATES L2 L1)))

(DCL SYSTEM-HIGH NIL)

(ADD-AXIOM SYSTEM-HIGH-DOMINATES
           (REWRITE)
           (DOMINATES (SYSTEM-HIGH) L))

(DCL UNDEFINED NIL)
```

```
(DEFN
   ADD-VALUE-READ
   (V P-NAME PROCESSES)
; adds value v to the values read (in reverse order) by the
; first process in PROCESSES with name P-NAME
   (IF (NOT (LISTP PROCESSES))
       PROCESSES
       (IF (EQUAL P-NAME
                  (PROC-NAME (CAR PROCESSES)))
           (CONS (PROCESS P-NAME
                          (CONS V
                                (VALUES-READ (CAR PROCESSES))))
                 (CDR PROCESSES))
           (CONS (CAR PROCESSES)
                 (ADD-VALUE-READ V P-NAME
                                 (CDR PROCESSES))))))

(DEFN REWRITE-OBJECT
      (O-NAME V L OBJECTS)
; rewrites the value of object O-NAME to V and the level to L,
; in the given list of OBJECTS
      (IF (NOT (LISTP OBJECTS))
          OBJECTS
          (IF (EQUAL O-NAME
                     (OBJ-NAME (CAR OBJECTS)))
              (CONS (OBJECT O-NAME V L)
                    (CDR OBJECTS))
              (CONS (CAR OBJECTS)
                    (REWRITE-OBJECT O-NAME V L
                                    (CDR OBJECTS))))))

(DEFN READ
      (P-NAME O ST)
      (IF (DOMINATES (PLEVEL P-NAME) (OLEVEL O))
          (STATE (ADD-VALUE-READ (VALUE O)
                                 P-NAME
                                 (PROCESSES ST))
                 (OBJECTS ST))
          ST))

(DEFN WRITE
      (LEVEL O V ST)
      (IF (DOMINATES (OLEVEL O) LEVEL)
          (STATE (PROCESSES ST)
                 (REWRITE-OBJECT (OBJ-NAME O)
                                 V LEVEL
                                 (OBJECTS ST)))
          ST))
```

```
(DEFN RESET
      (LEVEL O ST)
      (IF (DOMINATES (OLEVEL O) LEVEL)
          (STATE (PROCESSES ST)
                 (REWRITE-OBJECT (OBJ-NAME O)
                                 (UNDEFINED)
                                 (SYSTEM-HIGH)
                                 (OBJECTS ST)))
          ST))

(ADD-SHELL MAKE-INSTRUCTION NIL INSTRUCTIONP
           ((TYPE (NONE-OF) ZERO)
            (I-PROC-NAME (NONE-OF) ZERO)
            (I-OBJ-NAME (NONE-OF) ZERO)
            (I-VALUE (NONE-OF) ZERO)))

(DEFN I-OBJ
      (INST ST)
      (GET-OBJECT (I-OBJ-NAME INST)
                  (OBJECTS ST)))

(DEFN SINGLE-STEP
      (INST ST)
      (IF (GET-OBJECT (I-OBJ-NAME INST) ;could have used I-OBJ
                      (OBJECTS ST))
          (IF (EQUAL (TYPE INST) 'READ)
              (READ (I-PROC-NAME INST)
                    (I-OBJ INST ST)
                    ST)
              (IF (EQUAL (TYPE INST) 'WRITE)
                  (WRITE (PLEVEL (I-PROC-NAME INST))
                         (I-OBJ INST ST)
                         (I-VALUE INST)
                         ST)
                  (RESET (PLEVEL (I-PROC-NAME INST))
                         (I-OBJ INST ST)
                         ST)))
          ST))

; Notice that the following definition assumes that the
; instruction stream is given in reverse order.  This is
; because of the way that we wish to recur. Gypsy makes
; this choice irrelevant because of the symmetry between
; first and last.

(DEFN INTERPRET
      (INSTLIST ST)
      (IF (NOT (LISTP INSTLIST))
          ST
          (SINGLE-STEP (CAR INSTLIST)
                       (INTERPRET (CDR INSTLIST) ST))))

; I found an error in an earlier version of the following
; definition by looking at theorem-prover output:  namely,
; I left off I-PROC-NAME. The output contained:
;    (PLEVEL (MAKE-INSTRUCTION V1 C Z1 X1)).
```

```
(DEFN PURGE
      (INSTLIST LEVEL)
      (IF (NOT (LISTP INSTLIST))
          INSTLIST
          (IF (DOMINATES LEVEL
                         (PLEVEL (I-PROC-NAME (CAR INSTLIST))))
              (CONS (CAR INSTLIST)
                    (PURGE (CDR INSTLIST) LEVEL))
              (PURGE (CDR INSTLIST) LEVEL))))

(DEFN OBJECTS-MATCH-BELOW-LEVEL
      (LEVEL OBJS1 OBJS2)
      (IF (AND (LISTP OBJS1) (LISTP OBJS2))
          (IF (OR (DOMINATES LEVEL (OLEVEL (CAR OBJS1)))
                  (DOMINATES LEVEL
                             (OLEVEL (CAR OBJS2))))
              (AND (EQUAL (CAR OBJS1) (CAR OBJS2))
                   (OBJECTS-MATCH-BELOW-LEVEL LEVEL
                                              (CDR OBJS1)
                                              (CDR OBJS2)))
              (OBJECTS-MATCH-BELOW-LEVEL LEVEL
                                         (CDR OBJS1)
                                         (CDR OBJS2)))
          (AND (NOT (LISTP OBJS1))
               (NOT (LISTP OBJS2)))))

(DEFN OBJECT-NAMES-AGREE
      (OBJS1 OBJS2)
      (IF (AND (LISTP OBJS1) (LISTP OBJS2))
          (AND (EQUAL (OBJ-NAME (CAR OBJS1))
                      (OBJ-NAME (CAR OBJS2)))
               (OBJECT-NAMES-AGREE (CDR OBJS1)
                                   (CDR OBJS2)))
          (AND (NOT (LISTP OBJS1))
               (NOT (LISTP OBJS2)))))

(PROVE-LEMMA OBJECT-NAMES-AGREE-PROPERTY
             (REWRITE)
             (IMPLIES (AND (OBJECT-NAMES-AGREE OBJS1 OBJS2)
                           (GET-OBJECT NAME OBJS1)
                           (OBJECT-LISTP OBJS1)
                           (OBJECT-LISTP OBJS2))
                      (GET-OBJECT NAME OBJS2)))

(DEFN PROCESS-VIEW-IDENTICAL
      (P-NAME ST1 ST2)
      (AND (EQUAL (GET-PROCESS P-NAME (PROCESSES ST1))
                  (GET-PROCESS P-NAME (PROCESSES ST2)))
           (OBJECT-NAMES-AGREE (OBJECTS ST1)
                               (OBJECTS ST2))
           (OBJECTS-MATCH-BELOW-LEVEL (PLEVEL P-NAME)
                                      (OBJECTS ST1)
                                      (OBJECTS ST2))))
```

```
; Our goal is to prove that purging preserves the process
; view.  We'll induct according to the definition of INTERPRET.

; The base case follows easily, so let's concentrate on the
; inductive step.  Our first goal is to show that the
; interpreter and single stepper preserve the notion of state.

(PROVE-LEMMA
        ADD-VALUE-READ-PRESERVES-PROCESS-LISTP
        (REWRITE)
        (IMPLIES (PROCESS-LISTP PL)
                 (PROCESS-LISTP (ADD-VALUE-READ V P-NAME PL))))

(PROVE-LEMMA
    REWRITE-OBJECT-PRESERVES-OBJECT-LISTP
    (REWRITE)
    (IMPLIES (OBJECT-LISTP OL)
             (OBJECT-LISTP (REWRITE-OBJECT ONAME VAL LEV OL))))

(PROVE-LEMMA SINGLE-STEP-PRESERVES-PROPER-STATEP
             (REWRITE)
             (IMPLIES (PROPER-STATEP S)
                      (PROPER-STATEP (SINGLE-STEP INST S))))

(PROVE-LEMMA INTERPRET-PRESERVES-PROPER-STATEP
             (REWRITE)
             (IMPLIES (PROPER-STATEP S)
                      (PROPER-STATEP (INTERPRET INSTLIST S)))
             ((DISABLE SINGLE-STEP)))

(PROVE-LEMMA NOT-DOMINATES-IMPLIES-NOT-DOMINATES-SYSTEM-HIGH
             (REWRITE)
             (IMPLIES (NOT (DOMINATES LEVEL1 LEVEL2))
                      (NOT (DOMINATES LEVEL1 (SYSTEM-HIGH)))))

(PROVE-LEMMA
           NAME-OF-GET-OBJECT
           (REWRITE)
           (IMPLIES (GET-OBJECT NAME OBJECTS)
                    (EQUAL (OBJ-NAME (GET-OBJECT NAME OBJECTS))
                           NAME)))

(PROVE-LEMMA DOMINATES-IS-TRANSITIVE-AGAIN
             (REWRITE)
             (IMPLIES (AND (DOMINATES LEV1 LEV2)
                           (NOT (DOMINATES LEV3 LEV2)))
                      (NOT (DOMINATES LEV3 LEV1))))
```

```
(PROVE-LEMMA
 REWRITE-OBJECT-PRESERVES-OBJECTS-BELOW-LEVEL
 (REWRITE)
 (IMPLIES
  (AND (NOT (DOMINATES LEVEL
                       (OLEVEL (GET-OBJECT NAME OBJECTS1))))
       (OBJECTS-MATCH-BELOW-LEVEL LEVEL OBJECTS1 OBJECTS2)
       (GET-OBJECT NAME OBJECTS1)
       (OBJECT-LISTP OBJECTS1)
       (NOT (DOMINATES LEVEL HIGH-LEVEL)))
  (OBJECTS-MATCH-BELOW-LEVEL LEVEL
          (REWRITE-OBJECT (OBJ-NAME (GET-OBJECT NAME OBJECTS1))
                          VALUE HIGH-LEVEL OBJECTS1)
          OBJECTS2)))

(DISABLE NAME-OF-GET-OBJECT)

(PROVE-LEMMA OBJECTS-MATCH-BELOW-LEVEL-SYMMETRY
             NIL
             (EQUAL (OBJECTS-MATCH-BELOW-LEVEL LEVEL OBJS1
                                               OBJS2)
                    (OBJECTS-MATCH-BELOW-LEVEL LEVEL OBJS2
                                               OBJS1)))

(PROVE-LEMMA
 REWRITE-OBJECT-PRESERVES-OBJECT-NAMES-AGREE
 (REWRITE)
 (IMPLIES
         (OBJECT-NAMES-AGREE OBJECTS1 OBJECTS2)
         (AND (OBJECT-NAMES-AGREE (REWRITE-OBJECT NAME VALUE
                                                  LEVEL
                                                  OBJECTS1)
                                  OBJECTS2)
              (OBJECT-NAMES-AGREE OBJECTS2
                                  (REWRITE-OBJECT NAME VALUE
                                                  LEVEL
                                                  OBJECTS1))
              (OBJECT-NAMES-AGREE (REWRITE-OBJECT NAME VALUE
                                                  LEVEL
                                                  OBJECTS2)
                                  OBJECTS1)
              (OBJECT-NAMES-AGREE OBJECTS1
                                  (REWRITE-OBJECT NAME
                                                  VALUE
                                                  LEVEL
                                                  OBJECTS2)))))
```

```
(PROVE-LEMMA
       HIGHER-READ-PRESERVES-VALUES-READ
       (REWRITE)
       (IMPLIES (NOT (DOMINATES (PLEVEL LOW-NAME)
                                (PLEVEL HIGH-NAME)))
                (EQUAL (GET-PROCESS LOW-NAME
                                    (ADD-VALUE-READ VAL
                                                    HIGH-NAME
                                                    PROCESSES))
                       (GET-PROCESS LOW-NAME PROCESSES))))

(PROVE-LEMMA
   PURGEABLE-INSTR-PRESERVES-PROCESS-VIEW
   (REWRITE)
   (IMPLIES (AND (PROPER-STATEP ST1)
                 (PROPER-STATEP ST2)
                 (INSTRUCTIONP INST)
                 (PROCESS-VIEW-IDENTICAL P-NAME ST1 ST2)
                 (NOT (DOMINATES (PLEVEL P-NAME)
                                 (PLEVEL (I-PROC-NAME INST)))))
            (PROCESS-VIEW-IDENTICAL P-NAME
                                    (SINGLE-STEP INST ST1)
                                    ST2)))

(PROVE-LEMMA
  PROCESS-VIEW-IDENTICAL-GIVES-DOMINATION-OF-SAME-LOW-LEVELS
  (REWRITE)
  (IMPLIES (AND (OBJECT-LISTP OBJECTS1)
                (OBJECT-LISTP OBJECTS2)
                (OBJECTS-MATCH-BELOW-LEVEL LEVEL OBJECTS1
                                           OBJECTS2)
                (OBJECT-NAMES-AGREE OBJECTS1 OBJECTS2)
                (GET-OBJECT ONAME OBJECTS1)
                (DOMINATES (OLEVEL (GET-OBJECT ONAME OBJECTS1))
                           LOW-LEVEL)
                (DOMINATES LEVEL LOW-LEVEL))
           (DOMINATES (OLEVEL (GET-OBJECT ONAME OBJECTS2))
                      LOW-LEVEL)))

; The following is stated so as to keep the number of free
; variables as small as possible.
```

```
(PROVE-LEMMA CASE-13
 (REWRITE)
 (IMPLIES
     (AND (OBJECT-LISTP (OBJECTS ST1))
          (OBJECT-LISTP (OBJECTS ST2))
          (NOT (DOMINATES (OLEVEL (GET-OBJECT (I-OBJ-NAME INST)
                                              (OBJECTS ST2)))
                          (PLEVEL (I-PROC-NAME INST))))
          (OBJECT-NAMES-AGREE (OBJECTS ST1)
                              (OBJECTS ST2))
          (GET-OBJECT (I-OBJ-NAME INST)
                      (OBJECTS ST1))
          (DOMINATES (OLEVEL (GET-OBJECT (I-OBJ-NAME INST)
                                         (OBJECTS ST1)))
                     (PLEVEL (I-PROC-NAME INST)))
          (DOMINATES (PLEVEL P-NAME)
                     (PLEVEL (I-PROC-NAME INST))))
     (NOT (OBJECTS-MATCH-BELOW-LEVEL (PLEVEL P-NAME)
                                     (OBJECTS ST1)
                                     (OBJECTS ST2))))
 ((USE
   (PROCESS-VIEW-IDENTICAL-GIVES-DOMINATION-OF-SAME-LOW-LEVELS
                        (OBJECTS1 (OBJECTS ST1))
                        (OBJECTS2 (OBJECTS ST2))
                        (ONAME (I-OBJ-NAME INST))
                        (LOW-LEVEL (PLEVEL (I-PROC-NAME INST)))
                        (LEVEL (PLEVEL P-NAME))))
  (DISABLE
  PROCESS-VIEW-IDENTICAL-GIVES-DOMINATION-OF-SAME-LOW-LEVELS)))

(PROVE-LEMMA OBJECT-NAMES-AGREE-SYMMETRY
             (REWRITE)
             (EQUAL (OBJECT-NAMES-AGREE OBJS1 OBJS2)
                    (OBJECT-NAMES-AGREE OBJS2 OBJS1)))
```

```
(PROVE-LEMMA
 REST-OF-CASE-6
 (REWRITE)
 (IMPLIES
     (AND (OBJECT-LISTP (OBJECTS ST2))
          (OBJECT-LISTP (OBJECTS ST1))
          (NOT (DOMINATES (OLEVEL (GET-OBJECT (I-OBJ-NAME INST)
                                              (OBJECTS ST1)))
                          (PLEVEL (I-PROC-NAME INST))))
          (OBJECT-NAMES-AGREE (OBJECTS ST1)
                              (OBJECTS ST2))
          (GET-OBJECT (I-OBJ-NAME INST)
                      (OBJECTS ST2))
          (DOMINATES (OLEVEL (GET-OBJECT (I-OBJ-NAME INST)
                                         (OBJECTS ST2)))
                     (PLEVEL (I-PROC-NAME INST)))
          (DOMINATES (PLEVEL P-NAME)
                     (PLEVEL (I-PROC-NAME INST))))
     (NOT (OBJECTS-MATCH-BELOW-LEVEL (PLEVEL P-NAME)
                                     (OBJECTS ST1)
                                     (OBJECTS ST2))))
 ((USE
   (CASE-13 (ST1 ST2) (ST2 ST1))
   (OBJECTS-MATCH-BELOW-LEVEL-SYMMETRY
                                      (LEVEL (PLEVEL P-NAME))
                                      (OBJS1 (OBJECTS ST1))
                                      (OBJS2 (OBJECTS ST2))))))

(PROVE-LEMMA OBJECT-NAMES-AGREE-PROPERTY-AGAIN
             (REWRITE)
             (IMPLIES (AND (OBJECT-NAMES-AGREE OBJS2 OBJS1)
                           (GET-OBJECT NAME OBJS1)
                           (OBJECT-LISTP OBJS1)
                           (OBJECT-LISTP OBJS2))
                      (GET-OBJECT NAME OBJS2)))

(DISABLE OBJECT-NAMES-AGREE-SYMMETRY)

; The following was IMPORTANT for the proof below it!

(ENABLE NAME-OF-GET-OBJECT)
```

```
(PROVE-LEMMA
 REWRITE-OBJECT-PRESERVES-OBJECTS-BELOW-LEVEL-AGAIN
 (REWRITE)
 (IMPLIES
  (AND (OBJECT-NAMES-AGREE OBJECTS1 OBJECTS2)
       (OBJECTS-MATCH-BELOW-LEVEL PLEVEL OBJECTS1 OBJECTS2)
       (GET-OBJECT NAME OBJECTS1)
       (OBJECT-LISTP OBJECTS1)
       (OBJECT-LISTP OBJECTS2)
       (DOMINATES HIGH-LEVEL PLEVEL))
  (OBJECTS-MATCH-BELOW-LEVEL PLEVEL
          (REWRITE-OBJECT (OBJ-NAME (GET-OBJECT NAME OBJECTS1))
                          VALUE HIGH-LEVEL OBJECTS1)
          (REWRITE-OBJECT (OBJ-NAME (GET-OBJECT NAME OBJECTS2))
                          VALUE HIGH-LEVEL OBJECTS2))))

(PROVE-LEMMA
 REWRITE-OBJECTS-THE-SAME-PRESERVES-OBJECTS-BELOW-LEVEL
 (REWRITE)
 (IMPLIES
  (AND (OBJECT-NAMES-AGREE OBJECTS1 OBJECTS2)
       (OBJECTS-MATCH-BELOW-LEVEL PLEVEL OBJECTS1 OBJECTS2)
       (GET-OBJECT NAME OBJECTS1)
       (OBJECT-LISTP OBJECTS1)
       (OBJECT-LISTP OBJECTS2))
  (OBJECTS-MATCH-BELOW-LEVEL PLEVEL
          (REWRITE-OBJECT (OBJ-NAME (GET-OBJECT NAME OBJECTS1))
                          VALUE LEVEL OBJECTS1)
          (REWRITE-OBJECT (OBJ-NAME (GET-OBJECT NAME OBJECTS2))
                          VALUE LEVEL OBJECTS2))))

(DISABLE NAME-OF-GET-OBJECT)

(PROVE-LEMMA
 CASE-19
 (REWRITE)
 (IMPLIES
  (AND (PROCESS-LISTP X)
       (OBJECT-LISTP Z)
       (PROCESS-LISTP V)
       (OBJECT-LISTP W)
       (EQUAL (GET-PROCESS P-NAME X)
              (GET-PROCESS P-NAME V))
       (OBJECT-NAMES-AGREE Z W)
       (OBJECTS-MATCH-BELOW-LEVEL (PLEVEL P-NAME)
                                  Z W)
       (DOMINATES (PLEVEL P-NAME) (PLEVEL D))
       (NOT (DOMINATES (PLEVEL D)
                       (OLEVEL (GET-OBJECT X1 W))))
       (GET-OBJECT X1 Z)
       (DOMINATES (PLEVEL D)
                  (OLEVEL (GET-OBJECT X1 Z))))
  (EQUAL (GET-PROCESS P-NAME
                      (ADD-VALUE-READ (VALUE (GET-OBJECT X1 Z))
                                      D X))
         (GET-PROCESS P-NAME V))))
```

```
(PROVE-LEMMA CASE-9
          (REWRITE)
          (IMPLIES (AND (OBJECT-LISTP Z)
                        (OBJECT-LISTP W)
                        (OBJECT-NAMES-AGREE Z W)
                        (OBJECTS-MATCH-BELOW-LEVEL LEVEL Z W)
                        (DOMINATES LEVEL (PLEVEL D))
                        (DOMINATES (PLEVEL D)
                                   (OLEVEL (GET-OBJECT X1 W))))
                   (DOMINATES (PLEVEL D)
                              (OLEVEL (GET-OBJECT X1 Z)))))

(PROVE-LEMMA
 SAME-ADD-VALUE-READ
 (REWRITE)
 (IMPLIES
  (PROCESS-LISTP X)
  (EQUAL
   (GET-PROCESS NAME1
                (ADD-VALUE-READ VAL NAME2 X))
   (IF (EQUAL NAME1 NAME2)
       (IF (GET-PROCESS NAME1 X)
           (PROCESS NAME1
                    (CONS VAL
                          (VALUES-READ (GET-PROCESS NAME1 X))))
           F)
       (GET-PROCESS NAME1 X)))))

(PROVE-LEMMA
  OBJECTS-AGREE-BELOW-LEVEL NIL
  (IMPLIES (AND (OBJECT-LISTP OBJS1)
                (OBJECT-LISTP OBJS2)
                (OBJECT-NAMES-AGREE OBJS1 OBJS2)
                (OBJECTS-MATCH-BELOW-LEVEL HIGH-LEVEL OBJS1
                                           OBJS2)
                (DOMINATES HIGH-LEVEL LOW-LEVEL)
                (DOMINATES LOW-LEVEL
                           (OLEVEL (GET-OBJECT O-NAME OBJS2))))
           (EQUAL (VALUE (GET-OBJECT O-NAME OBJS1))
                  (VALUE (GET-OBJECT O-NAME OBJS2)))))
```

```
(PROVE-LEMMA
 CASE-7
 (REWRITE)
 (IMPLIES
  (AND (PROCESS-LISTP X)
       (OBJECT-LISTP Z)
       (PROCESS-LISTP V)
       (OBJECT-LISTP W)
       (EQUAL (GET-PROCESS P-NAME X)
              (GET-PROCESS P-NAME V))
       (OBJECT-NAMES-AGREE Z W)
       (OBJECTS-MATCH-BELOW-LEVEL (PLEVEL P-NAME)
                                  Z W)
       (DOMINATES (PLEVEL P-NAME) (PLEVEL D))
       (DOMINATES (PLEVEL D)
                  (OLEVEL (GET-OBJECT X1 W)))
       (GET-OBJECT X1 Z))
  (EQUAL
   (EQUAL
         (GET-PROCESS P-NAME
                      (ADD-VALUE-READ (VALUE (GET-OBJECT X1 Z))
                                      D X))
         (GET-PROCESS P-NAME
                      (ADD-VALUE-READ (VALUE (GET-OBJECT X1 W))
                                      D V)))
   T))
 ((USE (OBJECTS-AGREE-BELOW-LEVEL (OBJS1 Z)
                                  (OBJS2 W)
                                  (LOW-LEVEL (PLEVEL D))
                                  (HIGH-LEVEL (PLEVEL P-NAME))
                                  (O-NAME X1)))))

(DISABLE SAME-ADD-VALUE-READ)

(PROVE-LEMMA
     NONPURGEABLE-INSTRUCTION-PRESERVES-PROCESS-VIEW
     (REWRITE)
     (IMPLIES (AND (PROPER-STATEP ST1)
                   (PROPER-STATEP ST2)
                   (PROCESS-VIEW-IDENTICAL P-NAME ST1 ST2)
                   (DOMINATES (PLEVEL P-NAME)
                              (PLEVEL (I-PROC-NAME INST))))
              (PROCESS-VIEW-IDENTICAL P-NAME
                                      (SINGLE-STEP INST ST1)
                                      (SINGLE-STEP INST ST2))))

(PROVE-LEMMA OBJECT-NAMES-AGREE-IS-REFLEXIVE
             (REWRITE)
             (OBJECT-NAMES-AGREE X X))

(PROVE-LEMMA OBJECTS-MATCH-BELOW-LEVEL-IS-REFLEXIVE
             (REWRITE)
             (OBJECTS-MATCH-BELOW-LEVEL LEVEL X X))
```

```
(PROVE-LEMMA PROCESS-VIEW-IDENTICAL-IS-REFLEXIVE
             (REWRITE)
             (PROCESS-VIEW-IDENTICAL P ST ST))

(DISABLE SINGLE-STEP)

(DISABLE PROCESS-VIEW-IDENTICAL)

(DISABLE PROPER-STATEP)

(DEFN INSTRUCTION-LISTP
      (LST)
      (IF (NOT (LISTP LST))
          T
          (AND (INSTRUCTIONP (CAR LST))
               (INSTRUCTION-LISTP (CDR LST)))))

; The following lemma was originally proved with (and the
; main lemmas were developed with an eye toward) the hint:
; ((INDUCT (INTERPRET INSTLIST ST))) .

(PROVE-LEMMA
 PURGE-PRESERVES-PROCESS-VIEW NIL
 (IMPLIES
  (AND (PROPER-STATEP ST)
       (INSTRUCTION-LISTP INSTLIST))
  (PROCESS-VIEW-IDENTICAL P-NAME
                    (INTERPRET INSTLIST ST)
                    (INTERPRET (PURGE INSTLIST (PLEVEL P-NAME))
                               ST))))

(PROVE-LEMMA
 SYSTEM-IS-SECURE
 (REWRITE)
 (IMPLIES
  (AND (PROPER-STATEP ST)
       (INSTRUCTION-LISTP INSTLIST))
  (EQUAL
   (GET-PROCESS P-NAME
                (PROCESSES (INTERPRET INSTLIST ST)))
   (GET-PROCESS P-NAME
         (PROCESSES (INTERPRET (PURGE INSTLIST (PLEVEL P-NAME))
                               ST)))))
 ((USE (PURGE-PRESERVES-PROCESS-VIEW))
  (ENABLE PROCESS-VIEW-IDENTICAL)))
```

References

1. D.I. Good, R.L. Akers, L.M. Smith, ‘‘Report on Gypsy 2.05’’, Tech. report ICSCA-CMP-48, Institute for Computer Science and Computing Applications, The University of Texas at Austin, February 1986.

2. D.I. Good, B.L. Divito, M.K. Smith, ‘‘Using The Gypsy Methodology’’, Tech. report, Institute for Computing Science, University of Texas at Austin, June 1984.

3. Department of Defense, ‘‘*Trusted Computer Systems Evaluation Criteria*’’, DOD 5200.28-STD, December, 1985.

4. M.K. Smith, A. Siebert, B. Divito, and D. Good, ‘‘A Verified Encrypted Packet Interface’’, *Software Engineering Notes,* Vol. 6, No. 3, July 1981.

5. D.I. Good, A.E. Siebert, L.M. Smith, ‘‘Message Flow Modulator Final Report’’, Tech. report ICSCA-CMP-34, Institute for Computing Science, University of Texas at Austin, December 1982.

6. D.I. Good, ‘‘SCOMP Trusted Processes’’, ICSCA Internal Note 138, The University of Texas at Austin.

7. W.E. Boebert, W.D. Young, R.Y. Kain, S.A. Hansohn, ‘‘Secure ADA Target: Issues, System Design, and Verification’’, *Proc. Symposium on Security and Privacy,* IEEE, 1985.

8. J. Keeton-Williams, S.R. Ames, B.A. Hartman, and R.C. Tyler, ‘‘Verification of the ACCAT-Guard Downgrade Trusted Process’’, Tech. report NTR-8463, The Mitre Corporation, 1982.

9. K.N. Levitt, L. Robinson, B.A. Silverberg, ‘‘The HDM Handbook," vols. 1-3’’, Tech. report, SRI International, June 1979.

10. E.J. McCauley and P.J. Drongowski, ‘‘KSOS: The Design of a Secure Operating System’’, *Proceedings of the AFIPS Conf., Vol. 48,* AFIPS Press, Arlington, VA., 1979.

11. ‘‘System Specification for SAC Digital Network’’, ESD-MCV-1A, ITT Defense Communications Division, Nutley, N.J..

12. Warren A. Hunt, ‘‘FM8501: A Verified Microprocessor’’, Tech. report ICSCA-CMP-47, Institute for Computing Science, University of Texas at Austin, December 1985.

13. W.R. Bevier, W.A. Hunt, W.D. Young, ‘‘Toward Verified Execution Environments’’, *Proceedings of the 1987 Symposium on Security and Privacy,* IEEE, 1987.

14. M. Cheheyl, M. Gasser, G. Huff, J. Millen, ‘‘Verifying Security’’, *ACM Computing Surveys,* Vol. 13, No. 3, September 1981, pp. 279-340.

15. Richard Kemmerer, ‘‘Verification Assessment Study Final Report’’, In 5 volumes, unpublished.

16. Michael K. Smith, ‘‘Low-Water-Mark, Gypsy Style’’, Internal Note 159, Institute for Computing Science, The University of Texas at Austin, February, 1985.

17. Michael K. Smith, ‘‘Low-Water-Mark Using Abstract Types’’, Internal Note 158, Institute for Computing Science, The University of Texas at Austin, February, 1985.

18. J. McCarthy, *et. al., LISP 1.5 Programmer’s Manual,* MIT Press, Cambridge, MA., 1965.

19. Robert S. Boyer and J Strother Moore, *A Computational Logic,* Academic Press, New York, 1979.

20. R.S. Boyer and J S. Moore, *Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures,* Academic Press, 1981, pp. 103-185.

21. Richard Cohen, ‘‘Proving Gypsy Programs’’, Tech. report ICSCA-CMP-51, Institute for Computing Science, University of Texas at Austin, 1986.

22. D.E. Bell and L.J. LaPadula, ‘‘"Secure Computer System: Unified Exposition and Multics Interpretation"’’, Tech. report MTR-2997, MITRE Corp., July 1975.

23. John Rushby, ‘‘"The Low-Water Mark Example"’’, in Richard Kemmerer, editor, *Verification Assessment Study Final Report,* Vol. 5, unpublished.

24. J.A. Goguen and J. Meseguer, ''Security Policy and Security Models'', *Proc. Symposium on Security and Privacy*, IEEE, 1982, pp. 11-20.

25. J.A. Goguen and J. Meseguer, ''Unwinding and Inference Control'', *Proc. Symposium on Security and Privacy*, IEEE, 1984, pp. 75-86.

26. John Rushby, ''Mathematical Foundations of the MLS Tool for Revised Special'', Draft internal note, Computer Science Laboratory, SRI International, Menlo Park, California.

27. J.T. Haigh, W.D. Young, ''Extending the Non-Interference Version of MLS for SAT'', *Proceedings of the 1986 Symposium on Security and Privacy*, IEEE, 1986, pp. 232-239.

28. William D. Young, ''Proof Logs for Low-Water-Mark Problem Using Non-Interference'', Internal note 213, Institute for Computer Science and Computing Applications, The University of Texas at Austin.

29. Matt Kaufmann, ''A Primitive User's Manual for an Interactive Version of the Boyer-Moore Theorem-Prover (Draft)'', ICSCA Internal Notes 234 (Part 1) and 235 (Part 2), The University of Texas at Austin.

30. R.S. Boyer and J S. Moore, ''The Addition of Bounded Quantification and Partial Functions to the Boyer-Moore Logic and Theorem Prover'', Tech. report ICSCA-CMP-52, Institute for Computer Science and Computing Applications, The University of Texas at Austin, January 1987.

31. William D. Young, ''The Low-Water-Mark Problem Using Non-Interference'', Internal note 212, Institute for Computer Science and Computing Applications, The University of Texas at Austin.

Table of Contents