An Application of the m-EVES Verification System

Dan Craigen

Trusted Systems Group
I. P. Sharp Associates Limited
265 Carling Avenue, Suite 600
Ottawa, Ontario K1S 2E1
CANADA

Telephone: (613) 236-9942
DDN: dan@ipsa.arpa

IPSA Conference Paper CP-88-5402-29
May 1988

Abstract

The Trusted Systems Group of I.P. Sharp Associates Limited has recently released a prototype formal verification system, called m-EVES. m-EVES consists of a new language, called m-Verdi, for implementing and specifying software; a new logic (which has been proven sound); and a new theorem prover, called m-NEVER, which integrates many state-of-the-art techniques drawn from the theorem proving literature.

In this paper, after a brief overview of the m-EVES system, an application of m-EVES to a proof of a non-trivial security property (non-interference) for a pedagogical computer system (the Low Water Mark system) is discussed. An example demonstrates some of the power and novel features of m-EVES. The paper concludes with a comparison of the m-EVES solution with similar efforts using the Gypsy Verification Environment and the Boyer-Moore theorem prover.

Keywords: Automated deduction, Boyer-Moore Theorem Prover, Formal specification, Formal verification, Gypsy Verification Environment, Logic of programs, m-EVES, m-NEVER, m-Verdi, Program correctness, Program specification, Program verification systems, Theorem proving.

1 Introduction

For the purposes of this paper, it is taken as axiomatic that the application of formal methods to software development and, likewise, the application of verification systems, are of benefit. Interested readers are directed to [Cra 87a] [EVES 88] for discussions on the potential benefits of verification systems and the weaknesses inherent in many currently fielded verification systems; and to [Kem 86] [Cra 86] for overviews and commentaries on some of the leading North American verification systems.

m-EVES (an Environment for Verifying and Evaluating Software) is a prototype formal verification system. The development of m-EVES has followed two principal streams:

- the design of a new specification and implementation language (with supporting formal mathematics), called m-Verdi; and
- the implementation of a new theorem prover, called m-NEVER.

The m-EVES project focus on these two streams is a direct consequence of our observations that many existing verification systems are logically unsound and utilize primitive theorem proving technologies [Cra 87b] [EVES 88].

Prior to our summary of the application of m-EVES to the non-interference property of the Low Water Mark, some of the technical aspects of the verification system must first be reviewed.2

1.1 m-Verdi

m-Verdi [Cra 87a] [CS 87] is a language which allows for the writing of formal specifications and the implementation of imperative programs. m-Verdi is strongly typed (and uses same sameeness); has constructs for writing annotations (e.g., pre and post annotations); distinguishes between symbols that may be used solely for annotative purposes and those which may also be used within an executable context; supports information hiding and abstraction through the use of a package construct; and uses an "environment" to introduce symbols which will form a link between an m-Verdi program and the (possibly empty) set of devices it may affect. While the language consists of many linguistic features that are relatively common to the programming language community, some of these features were affected by the development of the logic and, consequently, are subtly different.

The formal semantics [Saa 87] for m-Verdi is described

1The development of m-EVES was funded by the Canadian Department of National Defence and the United States Navy. The prefix "m" in m-EVES is meant to suggest the prototypical nature of the current verification system.

2The following overview is primarily drawn from [EVES 88] and has been mildly edited.
using a form of Denotational Semantics. The logic [Saa 87] is non-standard in that it extends the many-sorted Predicate Calculus so that new symbols can be added to a theory (using m-Verdi declarations), and includes proof obligations requiring the "conservative extension" (see §1.3) of theories. The logic requires that programs be proved to terminate and that the specification of program functionality be of the well-understood Floyd-Hoare style.

1.2 m-NEVER

The m-NEVER (Not the EVES Rewriter) theorem prover [PK 87] supports the interactive development of proofs and also has powerful automatic capabilities. These capabilities include the detection of propositional tautologies; the handling of equality and integer relations; the application of conditional rewrite rules; the heuristic expansion of defined functions; the automatic instantiation of quantified variables; and the Boyer-Moore style of automatic induction.

There are further capabilities for I/O support and database management, and many user commands for selectively invoking prover capabilities.

1.3 m-EVES

The m-EVES system [PSK 87] currently resides on a Symbolics Lisp Machine and interaction with m-EVES may occur either through an editor interface or a command processor.

Interaction with m-EVES occurs using a language which we are calling ECL (the EVES Command Language). This language is essentially an extension of m-Verdi to include m-EVES commands (which, for the most part, are commands to invoke portions of the theorem prover).

Conceptually, the development of an annotated program results in the incremental development of a logical theory. A theory is determined by a set of symbols (e.g., constant symbols, function symbols), called a vocabulary, and a set of axioms which relate those symbols (e.g., in a sequence theory, an axiom might have of the form head(tack(e0,a0)) = e0).

Program development starts from an initial theory. The initial theory includes symbols dealing with Integers, Booleans, Characters and Ordinals [Fra 68] and relevant axioms.

The analyst incrementally extends a theory by introducing a declaration to m-EVES and by proving that the declaration satisfies the appropriate proof obligations [Saa 87]. A declaration increases the set of symbols and adds axioms about the newly introduced symbols. The proof obligations are such that the extension of a theory is a "conservative extension." Loosely, a conservative extension is one where the meanings assigned to the symbols existing prior to the extension are maintained in the extended theory.

3 As belies the experimental nature of this effort, experience has also led to some minor syntactic modifications to the form of some m-Verdi declarations (as they appear in ECL). In particular, the syntax for axioms and packages have changed. A tool has been written which converts ECL into m-Verdi.

2 The Low Water Mark System

The Low Water Mark System (LWM) [CGH 81] is described 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 must satisfy the simple security condition and the *-property. For simplicity the security levels are assumed to be linearly ordered.

The system incorporates a "low-water-mark" protection mechanism... 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.

In this paper, we prove that the LWM satisfies the non-interference property [Rus 84]. In essence, the non-interference property is as follows:

Suppose p and q are processes. We say that the process p does not interfere with the process q if no operation requested by p can influence any future output of q. Denote the non-interference relation by p → q.

Suppose the function VALUES returns the output to a process q relative to a sequence of operations s. Then we have:

\[ p \rightarrow q \iff VALUES(q, s) = VALUES(s \setminus p, q) \]

where \( s \setminus p \) denotes the subsequence of s resulting from the removal of all those instructions requested by p.

For the LWM, we associate with each process p a security level, slp, say, and require that p may interfere with q only if slp ≤ slo holds:

\[ -(p \rightarrow q) \Rightarrow slp \leq slo \]

To demonstrate the security of the LWM, it suffices to show that (where \( s \setminus slp \) denotes the subsequence of s resulting from the removal of all those instructions requested by processes having security levels which are strictly higher than those of process p):

\[ \forall p : VALUES(s, p) = VALUES(s \setminus slp, p) \]

We have chosen the LWM, as a non-trivial example application of m-EVES, since solutions to the problem exist for both the Gypsy Verification Environment (GVE) [GVE 84] [GVE 86] and the Boyer-Moore theorem prover [BM 79]. The commonality of the problem and the essentially similar proof structure facilitates comparison. Interested readers are directed to [KY 87a] for detailed presentations of
the GVE and Boyer-Moore solutions; [KY 87b] summarizes their experiences.

It should be noted that, unlike the GVE and Boyer-Moore solutions, the m-EVES solution also included code proofs. In the interest of brevity, these proofs (and formal specifications) are not included here. Further, only a sampling of the m-EVES output is included. Interested readers are directed to [Cra 87c] for the complete presentation.

3 Low Water Mark Proof

The prover commands in the example are summarized in the Appendix. In our description of the Low Water Mark example, the text in typewriter font is that arising from an interaction with m-EVES. Input commands, which include declarations, are terminated by semicolons. We have slightly edited the interaction to take into account formatting restrictions for this paper and have included the m-EVES prompt, the exclamation point, “!”.

As a result, all commands are bracketed by an exclamation point and a semicolon.

The development starts by defining a series of declarations which would be introduced through the environment. The environment is used to capture those concepts which are being assumed, or deliberately underspecified, during the proof effort. In general, such assumptions are of the following forms:

- The deliberate underspecification of constructs. For example, we may introduce a type but leave its definition unstated.
- The specification of constructs which are to be implemented outside of m-Verdi. The most common example of such a construct is a procedure handling low level input and output.
- The presentation of mathematical concepts, e.g., a sequence theory, which have been proven consistent elsewhere.

Our development is started by defining the security levels to be used. In this instance, and primarily for simplicity, I have decided to use an enumeration type of four elements.

The INT (integer) type is used to identify a process and a valid process identifier must be in the range L_PID to U_PID. The function PID_LEVEL is used as an extractor function that returns the security level associated with a process.

Observe that the axiom ABOUT_L_PID_AND_U_PID is a frule. In general, whenever the trigger expressions (in this instance, L_PID and U_PID) occur within a proposition that m-NEVER is attempting to prove, and the hypothesis (in this instance, TRUE) is provable, then the consequence (in this instance INT'LE(L_PID, U_PID)) is added as a hypothesis to the proposition being proved. frule is a mnemonic for “forward rule.”

The unspecified type TY_DATA defines the type of the data to be manipulated by the LWM. A sequence theory of this type is introduced so that a history of the information that is transmitted to each of the processes may be maintained. The minimal sequence theory kernel used here is an instantiated version of a part of the m-EVES library sequence theory (as it existed in October 1987).

Since the sequence theory concept is used twice (below for TY_DATA and later for KERNEL_REQUEST), a generic form of the theory is presented in Appendix B. One need only instantiate “XX” with the aforementioned type names to obtain the actual theories used in this development.6

The following series of environment declarations lead to the definition of the ABSTRACT_STATE.

The state consists of three components: memory (modelled by the array type TY_MEMORY), security tags (modelled by the array type TY_LEVELS), and the history of data transmitted to each process (modelled by the type TY_OUTPUT).

For technical reasons, a variable declaration introduces a denumerable set of symbols to the vocabulary. For the declaration of SL, the symbols SL'0, SL'7678, etc., are also introduced.

6It is intended that a form of polymorphism will be added to m-Verdi in the next phase of the EVES project.
need only instantiate "XX" with the aforementioned type names to obtain the actual theories used in this development.6

!type TY-DATA;
!var D: TY-DATA;
!const NULL-DATA: TY-DATA;

<Insert the sequence of TY-DATA here.>

The following series of environment declarations lead to the definition of the ABSTRACT-STATE.

The state consists of three components: memory (modelled by the array type TY-MEMORY), security tags (modelled by the array type TY-LEVELS), and the history of data transmitted to each process (modelled by the type TY-OUTPUT).

!const L-MEMORY: INT;
!const U-MEMORY: INT;
!axiom L-MEMORY_AND_U_MEMORY_RELATIONSHIP () =
frule triggers(L-MEMORY, U-MEMORY)
begin INT'LE (L-MEMORY, U-MEMORY)
end L-MEMORY_AND_U_MEMORY_RELATIONSHIP;

!var LOC: INT;
!function GOOD-LOCATION(LOC) :
BOOL =
begin AND(INT'LE(L-MEMORY, LOC),
INT'LE(LOC, U-MEMORY))
end GOOD-LOCATION;

!type TY-MEMORY =
array L-MEMORY .. U-MEMORY of TY-DATA;
!var M: TY-MEMORY;

!type TY-LEVELS =
array L-MEMORY .. U-MEMORY of SEC-LEVEL;
!var LS: TY-LEVELS;

!type TY-OUTPUT =
array L-PID .. U-PID of SEQUENCE-TY-DATA;
!var OT: TY-OUTPUT;
!type ABSTRACT-STATE =
record
field MEMORY : TY-MEMORY
field TAGS : TY-LEVELS
field OUTPUT : TY-OUTPUT
end ABSTRACT-STATE;
!var S: ABSTRACT-STATE;

The following enumeration lists the possible functions that may be invoked by the LWM.

!type OPERATIONS =
enum (oRESET, oWRITE, oREAD);
!var OPS: OPERATIONS;

The KERNEL-REQUEST type models the information that may be obtained from a process. As with processes and data, it is necessary to define a concept dealing with good kernel requests and a sequence comprising only good kernel requests. The sequence concept is used to model the history of kernel requests.

The heuristic information embodied in the rule keyword is described in footnote 8.

!type KERNEL-REQUEST;
!var KR: KERNEL-REQUEST;

!function PROCESS (KR) :
INT =
begin
and(GOOD_PID(PROCESS(KR)),
GOOD_LOCATION(LOCATION(KR)))
end GOOD_KERNEL_REQUEST;

!function GOOD_KERNEL_REQUEST.IMPLIES.GOOD_PID(KR) =
rule
begin
IMPLIES(GOOD_KERNEL_REQUEST(KR),
GOOD_PID(PROCESS(KR)) = TRUE)
end GOOD_KERNEL_REQUEST.IMPLIES.GOOD_PID;

!axiom GOOD_KERNEL_REQUEST.IMPLIES.GOOD_LOCATION
(KR) =
rule
begin
IMPLIES(GOOD_KERNEL_REQUEST(KR),
GOOD_LOCATION(LOCATION(KR)) = TRUE)
end GOOD_KERNEL_REQUEST.IMPLIES.GOOD_LOCATION;

<Insert the sequence of KERNEL-REQUEST here.>

!function GOOD_SEQUENCE_OF KERNEL_REQUESTS
(SO KERNEL REQUEST) :
BOOL =
measure ORDINAL VAL(LENGTH_KERNEL_REQUEST
(SO KERNEL REQUEST))
begin
if SO KERNEL REQUEST = EMPTY KERNEL REQUEST
then TRUE
else
AND(GOOD_SEQUENCE_OF KERNEL_REQUESTS(
TAIL_KERNEL_REQUEST(SO KERNEL REQUEST)),
GOOD_KERNEL_REQUEST(HEAD_KERNEL_REQUEST(SO KERNEL REQUEST))))
end if
end GOOD_SEQUENCE_OF KERNEL_REQUESTS;

6It is intended that a form of polymorphism will be added to m-Verdi in the next phase of the EVES project.
With the previous declaration, the environment is now completed. In what follows, the reader should view the declarations as being components of a package declaration. When the LWM was proved, the package command had not been implemented and so the effect was modelled with the appropriate use of “disabling.” It is important to realize that this proof uses the same levels of abstraction as if packages had been present.

A package consists of a package body and a package header. m-EVES takes a slightly different approach, to the syntactic aspects of packages, than that found in m-Verdi. In particular, with m-EVES, the model appears syntactically before the package header.

The main role of the package is to introduce a level of abstraction to the non-interference proof process and, through the heuristic content of the axioms expressed in the package header, to maximize the automatic aspects of the proof. Many good software engineering ideas, such as modularity and abstraction, have important applications in the proving process as well.

The function declarations that would occur in a package body are introduced first. The six declarations present the actual function definitions ORESET, OWRITE, OREAD, VISIBLE, LEVELS and VALUES. The first three functions specify the change of state that occurs when the LWM performs a reset, write or read operation, respectively. The final three functions extract information from the state: the contents of a specific location, the security level of a specific location, and the data transmitted to a particular process, respectively.

Subsequently, nine algebraic data type-like axioms specify the desired properties for the functions. These axioms must be provable from the function definitions and, in fact, were proved directly by m-NEVER.

Essentially, our approach results in a view that ORESET, OWRITE and OREAD are constructors of the abstract state datatype, and the remaining three functions are destructors. The nine axioms that describe the interrelationships are treated as conditional rewrite rules and are used in the subsequent proofs about non-interference.

Observe that a functional notation for updating and extracting arrays and records is being used. Since none of these functions are defined recursively, there are no associated proof obligations [Saa 87] [BM 79].

\[
\begin{align*}
\text{function ORESET (KR, S): ABSTRACT\_STATE} &= \text{pre GOOD\_KERNEL\_REQUEST(KR)} \\
\text{begin} &\quad \text{if SEC\_LEVEL'LE(PID\_LEVEL (PROCESS (KR)),}
\end{align*}
\]

\[
\begin{align*}
\text{TY\_LEVELS'WITH(ABSTRACT\_STATE'TAGS(S), LOCATION(KR)),}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'OUTPUT(S))} \\
\text{then} \\
\text{ABSTRACT\_STATE'VAL(}
\begin{align*}
\text{TY\_MEMORY'WITH(ABSTRACT\_STATE'MEMORY(S), LOCATION(KR),}\n\end{align*}
\]

\[
\begin{align*}
\text{NULL\_DATA),}
\end{align*}
\]

\[
\begin{align*}
\text{TY\_LEVELS'WITH(ABSTRACT\_STATE'TAGS(S), LOCATION(KR),}
\end{align*}
\]

\[
\begin{align*}
\text{HIGHEST\_SEC\_LEVEL),}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'OUTPUT(S))} \\
\text{else S} \\
\text{end if}
\end{align*}
\]

\[
\begin{align*}
\text{end ORESET;}
\end{align*}
\]

\[
\begin{align*}
\text{function OWRITE (KR, S): ABSTRACT\_STATE} &= \text{pre GOOD\_KERNEL\_REQUEST(KR)} \\
\text{begin} &\quad \text{if SEC\_LEVEL'LE(PID\_LEVEL (PROCESS (KR)),}
\end{align*}
\]

\[
\begin{align*}
\text{TY\_LEVELS'WITH(ABSTRACT\_STATE'TAGS(S), LOCATION(KR)),}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'OUTPUT(S))} \\
\text{then} \\
\text{ABSTRACT\_STATE'VAL(}
\begin{align*}
\text{TY\_MEMORY'WITH(}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'MEMORY(S), LOCATION(KR),}
\end{align*}
\]

\[
\begin{align*}
\text{DATA(KR)),}
\end{align*}
\]

\[
\begin{align*}
\text{TY\_LEVELS'WITH(}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'TAGS(S), LOCATION(KR),}
\end{align*}
\]

\[
\begin{align*}
\text{PID\_LEVEL(PROCESS(KR))),}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'OUTPUT(S))} \\
\text{else S} \\
\text{end if}
\end{align*}
\]

\[
\begin{align*}
\text{end OWRITE;}
\end{align*}
\]

\[
\begin{align*}
\text{function OREAD (KR, S): ABSTRACT\_STATE} &= \text{pre GOOD\_KERNEL\_REQUEST(KR)} \\
\text{begin} &\quad \text{if SEC\_LEVEL'LE(TY\_LEVELS'EXTRACT (}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'TAGS(S), LOCATION(KR))}
\end{align*}
\]

\[
\begin{align*}
\text{then} \\
\text{ABSTRACT\_STATE'VAL(}
\begin{align*}
\text{ABSTRACT\_STATE'MEMORY(S),}
\end{align*}
\]

\[
\begin{align*}
\text{LOCATION(KR),}
\end{align*}
\]

\[
\begin{align*}
\text{TY\_OUTPUT'WITH(}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'OUTPUT(S))} \\
\text{else S} \\
\text{end if}
\end{align*}
\]

\[
\begin{align*}
\text{end OREAD;}
\end{align*}
\]

\[
\begin{align*}
\text{function OREAD (KR, S): ABSTRACT\_STATE} &= \text{pre GOOD\_KERNEL\_REQUEST(KR)} \\
\text{begin} &\quad \text{if SEC\_LEVEL'LE(TY\_LEVELS'EXTRACT (}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'TAGS(S), LOCATION(KR))}
\end{align*}
\]

\[
\begin{align*}
\text{then} \\
\text{ABSTRACT\_STATE'VAL(}
\begin{align*}
\text{ABSTRACT\_STATE'MEMORY(S),}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'TAGS(S),}
\end{align*}
\]

\[
\begin{align*}
\text{TY\_OUTPUT'WITH(}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'OUTPUT(S), PROCESS(KR)),}
\end{align*}
\]

\[
\begin{align*}
\text{TACK\_TY\_DATA(}
\end{align*}
\]

\[
\begin{align*}
\text{TY\_MEMORY'EXTRACT (}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'MEMORY(S), LOCATION(KR))}
\end{align*}
\]

\[
\begin{align*}
\text{TY\_OUTPUT'EXTRACT (}
\end{align*}
\]

\[
\begin{align*}
\text{ABSTRACT\_STATE'OUTPUT(S), PROCESS(KR)))}
\end{align*}
\]
else S
end if
end oREAD;

!function VISIBLE (S, LOC): TY_DATA =
pre GOOD_LOCATION(LOC)
begin
  TY_MEMORY'EXTRACT(ABSTRACT_STATE'MEMORY(S),
          LOC)
end VISIBLE;

!function LEVELS (S, LOC): sec_level =
pre GOOD_LOCATION(LOC)
begin
  TY_LEVELS'EXTRACT(ABSTRACT_STATE'TAGS(S),
          LOC)
end LEVELS;

!function VALUES (S, PID): SEQUENCE_TY_DATA =
pre GOOD_PID(PID)
begin
  TY_OUTPUT'EXTRACT(ABSTRACT_STATE'OUTPUT(S),
          PID)
end VALUES;

Now the package specification is presented. Each axiom
must be provable from the aforementioned definitions. Only
the first proof is presented in entirety; the proofs for the re-
main ing axioms in the package specifications are of a similar
nature.

{EFFECT OF ORESET ON DESTRUCTORS.)

!axiom VISIBLE_ON_ORESET (KR, S, LOC) =
rule
begin
  IMPLIES(
    AND(GOOD_KERNEL_REQUEST(KR),
    GOOD_LOCATION(LOC)),
    VISIBLE (ORESET (KR, S), LOC)
  = if AND (LOC = LOCATION(KR),
    SEC_LEVEL'LE
    (PID_LEVEL (PROCESS (KR)),
    LEVELS (S, LOCATION(KR))))
  then NULL_DATA
  else VISIBLE (S, LOC) end if)
end VISIBLE_ON_ORESET;
!reduce;

Beginning proof of VISIBLE_ON_ORESET ...
Which simplifies
with invocation of
LEVELS, VISIBLE, ORESET, SEC_LEVEL'LE,
GOOD_KERNEL_REQUEST, GOOD_LOCATION, GOOD_PID
when rewriting with
TY_MEMORY'ACCESS OF UPDATE'AX,
SEC_LEVEL'TOP_SECRET'VALUE,
HIGHEST_SEC_LEVEL'VALUE
with the assumptions
ABSTRACT_STATE'MEMORY OF VAL'AX,
ABSTRACT_STATE'TAGS OF VAL'AX,
ABSTRACT_STATE'OUTPUT OF VAL'AX,
SEC_LEVEL'ORD OF VAL'AX,
SEC_LEVEL'VALUES'AX,
SEC_LEVEL'VAL OF ORD'AX,
ABSTRACT_STATE'VAL OF ACCESSES'AX,
L_MEMORY AND U_MEMORY RELATIONSHIP,
ABOUT L_PID AND U_PID to ...
TRUE

!axiom LEVELS_ON_ORESET (KR, S, LOC) =
rule
begin
  IMPLIES(AND(GOOD_KERNEL_REQUEST(KR),
    GOOD_LOCATION(LOC)),
    LEVELS (ORESET (KR, S), LOC)
  = if AND (LOC = LOCATION(KR),
    SEC_LEVEL'LE
    (PID_LEVEL (PROCESS (KR)),
    LEVELS (S, LOCATION(KR))))
  then HIGHEST_SEC_LEVEL
  else LEVELS (S, LOC) end if)
end LEVELS_ON_ORESET;
!reduce;

!axiom VALUES_ON_ORESET (KR, S, PID) =
rule
begin
  IMPLIES(
    AND(GOOD_KERNEL_REQUEST(KR),
    GOOD_PID(PID)),
    VALUES (ORESET (KR, S), PID)
  = VALUES (S, PID))
end VALUES_ON_ORESET;
!reduce;

The axioms describing the effect of OWRITE on the de-
structors are analogous to the axioms for ORESET. The only
difference is that a specific data item is written and the as-
sociated security level is that of the process requesting the
operation. These axioms are omitted from this paper.

The first two axioms describing the effect of OREAD on
the destructors express that neither VISIBLE nor LEVELS
is affected by the OREAD function. These axioms are omit-
ted from this paper.

!axiom VALUES_ON_OREAD (PID, KR, S) =
rule
begin
  IMPLIES(
    AND(GOOD_KERNEL_REQUEST(KR),
    GOOD_PID(PID)),
    VALUES (OREAD (KR, S), PID)
  = VALUES (S, PID))
end VALUES_ON_OREAD;
!reduce;
(LEVELS(S, LOCATION(KR)),
PID_LEVEL (PID)))
then TACK_TY_DATA
(VISIBLE(S, LOCATION(KR)),
VALUES (S, PID))
else VALUES@, PID) end if
end VALUES_ON_oREAD;
end if
end STEP_EVOLUTION;
reduce;

The above declaration ends the package declaration. The
effect of the following disable list is to hide the definitions
that would occur in the package body. As a result, the func-
tion definitions are now hidden and the prover will only use
the algebraic axioms as conditional rewrite rules in sub-
sequent proofs requiring information about the functions.
Further, since there is no need for the definitions of the var-
ious "goodness" criteria, those definitions are also hidden.

!disable oReset; !disable oWrite;
!disable oRead; !disable values;
!disable visible; !disable levels;
!disable good_pid; !disable good_location;
!disable good_kernel_request;

The declaration below formalizes the concept of perform-
ing a single kernel operation. The concept of performing a
sequence of kernel requests starting from an initial state is
required later—hence, the declaration of STEP_EVOLUTION.
The concept of purging, from a sequence of kernel requests,
all those requests which are from processes having a security
level higher than some particular level, is also introduced.

The latter two functions are recursive and, consequently,
have a proof obligation to show that each recursion will ter-
ninate [Saa 87] [BM 79]. The measure expressions, which
are of type ordinal, are used to prove these obligations. We
claim, substantiated by the proof, that the ordinal expres-
sions decrease on each recursive call and, hence, since the or-
dinals are well-founded with our definition of ordinal, the
recursions will terminate. The proof of the well-definedness
of the STEP_EVOLUTION function is documented here.

!function STEP (KR, S): ABSTRACT_STATE =
pre GOOD_KERNEL_REQUEST(KR)
begin
if COMMAND (KR) = OPERATIONS' oRESET
then oRESET (KR, S)
elseif COMMAND (KR) = OPERATIONS' oWRITE
then oWRITE (KR, S)
else oREAD (KR, S) end if
end STEP;

!function STEP_EVOLUTION (SO_KERNEL_REQUEST
, S) : ABSTRACT_STATE =
measure ORDINAL'VAL (LENGTH_KERNEL_REQUEST
(SO_KERNEL_REQUEST))
begin
if SO_KERNEL_REQUEST = EMPTY_KERNEL_REQUEST
then S
else STEP (READ_KERNEL_REQUEST
(SO_KERNEL_REQUEST),
STEP_EVOLUTION
(TAIL_KERNEL_REQUEST
(SO_KERNEL_REQUEST), S))
end if
end STEP_EVOLUTION;
reduce;

Beginning proof of STEP_EVOLUTION ...

IMPLIES (
SO_KERNEL_REQUEST <> EMPTY_KERNEL_REQUEST,
ORDINAL'LT
(ORDINAL'VAL
(LENGTH_KERNEL_REQUEST
(TAIL_KERNEL_REQUEST
(SO_KERNEL_REQUEST))),
ORDINAL'VAL
(LENGTH_KERNEL_REQUEST
(SO_KERNEL_REQUEST))))
Which simplifies
with invocation of LENGTH_KERNEL_REQUEST
when rewriting with ORDINAL'LT_6
with the assumptions
LENGTH_NON_NEGATIVE_KERNEL_REQUEST,
TACK_HEAD_TAIL_KERNEL_REQUEST to ...
TRUE

!function PURGE (SO_KERNEL_REQUEST, SL)
measure
ORDINAL'VAL (LENGTH_KERNEL_REQUEST
(SO_KERNEL_REQUEST))
begin
if SO_KERNEL_REQUEST = EMPTY_KERNEL_REQUEST
then EMPTY_KERNEL_REQUEST
elseif SEC_LEVEL'LE
(PID_LEVEL
(PROCESS (HEAD_KERNEL_REQUEST
(SO_KERNEL_REQUEST)));
SL)
then
TACK_KERNEL_REQUEST
(HEAD_KERNEL_REQUEST
(SO_KERNEL_REQUEST),
PURGE (TAIL_KERNEL_REQUEST
(SO_KERNEL_REQUEST), SL))
else
PURGE (TAIL_KERNEL_REQUEST
(SO_KERNEL_REQUEST), SL)
end if
end PURGE;
reduce;
The following two functions project the VISIBLE and LEVELS functions. The basic idea is that, if a memory location has a higher security level than the processor security level, we map the data component to null data and the security level to the highest security level. These two functions will be used in the definition of PROCESS-VIEW-IDENTICAL.

\begin{verbatim}
!function VISIBLE-PROJECTION(S, PID, LOC) : TY_DATA =
pre AND(GOOD_PID(PID),
   GOOD_LOCATION(LOC))
begin
   if SEC_LEVEL LT(PID_LEVEL(PID), LEVELS(S,LOC))
      then NULL_DATA
      else VISIBLE(S, LOC) end if
end VISIBLE-PROJECTION;

!function LEVELS-PROJECTION(S, PID, LOC) : SEC_LEVEL =
pre AND(GOOD_PID(PID),
   GOOD_LOCATION(LOC))
begin
   if SEC_LEVEL LT(PID_LEVEL(PID), LEVELS(S,LOC))
      then HIGHEST_SEC_LEVEL
      else LEVELS(S, LOC) end if
end LEVELS-PROJECTION;
\end{verbatim}

The PROCESS-VIEW-IDENTICAL predicate is now defined. PROCESS-VIEW-IDENTICAL is an equivalence relation which determines when two states will appear the same to a process. That PROCESS-VIEW-IDENTICAL is an equivalence relation was easily proven but, since it is not necessary to the current discussion, is omitted.

\begin{verbatim}
!function PROCESS-VIEW-IDENTICAL (PID, S'O, S'I) :
BOOL =
pre GOOD_PID(PID)
begin
   AND(
      VALUES(S'O, PID) = VALUES(S'I, PID),
      all LOC:
         IMPLIES(GOOD_LOCATION(LOC),
            AND(LEVELS-PROJECTION (S'O, PID, LOC) =
                LEVELS-PROJECTION (S'I, PID, LOC),
                VISIBLE-PROJECTION (S'O, PID, LOC) =
                VISIBLE-PROJECTION (S'I, PID, LOC))))
end PROCESS-VIEW-IDENTICAL;
\end{verbatim}

Using the next three theorems, it is shown that PROCESS-VIEW-IDENTICAL is maintained by instructions which are purgeable. Note that the proofs of these theorems are identical in structure (except for an extra instantiation in the third proof). Only the first proof is presented in entirety.

\begin{verbatim}
!axiom purgeable-oreset-PRESERVES-PP'   'S_VIEW (KR, S'O, S'I, PID) =
rule
begin IMPLIES(AND (GOOD_KERNEL_REQ _i(KR),
   GOOD_PID(PID),
   SEC_LEVEL LT (PID_LEVEL (PID), PID_LEVEL (PROCESS (KR))),
   PROCESS-VIEW-IDENTICAL (PID, S'O, S'I)),
   PROCESS-VIEW-IDENTICAL (PID, oRESET (KR, S'O), S'I))
end purgeable-oreset-PRESERVES_PROCESS_VIEW;
\end{verbatim}

Beginning proof of PURGEABLE-ORESET-PRESERVES_PROCESS_VIEW ...

\begin{verbatim}
!invoke PROCESS-VIEW-IDENTICAL;
\end{verbatim}
Invoking PROCESS_VIEW_IDENTICAL gives ...

IMPLIES (AND (GOOD_KERNEL_REQUEST (KR),
GOOD_PID (PID),
SEC_LEVEL_LT (PID_LEVEL (PID), PID_LEVEL (PROCESS (KR)))),
if GOOD_PID (PID)
then AND (VALUES (S'O, PID) = VALUES (S'1, PID),
all LOC:
  IMPLIES (GOOD_LOCATION (LOC),
  AND (LEVELS_PROJECTION (S'O, PID, LOC)
  = LEVELS_PROJECTION (S'1, PID, LOC),
  VISIBLE_PROJECTION (S'O, PID, LOC)
  = VISIBLE_PROJECTION (S'1, PID, LOC))))
else PROCESS_VIEW_IDENTICAL (PID, S'O, S'1) end if),
if GOOD_PID (PID)
then AND (VALUES (ORESET (KR, S'O), PID) = VALUES (S'1, PID),
all LOC'O:
  IMPLIES (GOOD_LOCATION (LOC'O),
  AND (LEVELS_PROJECTION (ORESET (KR, S'O),
  PID, LOC'O)
  = LEVELS_PROJECTION (S'1, PID, LOC'O),
  VISIBLE_PROJECTION (ORESET (KR, S'O),
  PID, LOC'O)
  = VISIBLE_PROJECTION (S'1, PID, LOC'O))))
else PROCESS_VIEW_IDENTICAL (PID, ORESET (KR, S'O), S'1) end if)

!trivial simplify;

Trivially simplifies to ...

IMPLIES (AND (GOOD_KERNEL_REQUEST (KR),
GOOD_PID (PID),
SEC_LEVEL_LT (PID_LEVEL (PID), PID_LEVEL (PROCESS (KR)))),
VALUES (S'O, PID) = VALUES (S'1, PID),
all LOC:
  IMPLIES (GOOD_LOCATION (LOC),
  AND (LEVELS_PROJECTION (S'O, PID, LOC)
  = LEVELS_PROJECTION (S'1, PID, LOC),
  VISIBLE_PROJECTION (S'O, PID, LOC)
  = VISIBLE_PROJECTION (S'1, PID, LOC))))

!instantiate LOC = LOC'O;

Instantiating LOC = LOC'O gives ...

all LOC:
  IMPLIES (AND (GOOD_KERNEL_REQUEST (KR),
  GOOD_PID (PID),
  SEC_LEVEL_LT (PID_LEVEL (PID), PID_LEVEL (PROCESS (KR)))),
VALUES (S'O, PID) = VALUES (S'1, PID),
IMPLIES (GOOD_LOCATION (LOC),
  AND (LEVELS_PROJECTION (S'O, PID, LOC))

29
= LEVELS_PROJECTION (S'1, PID, LOC),
VISIBLE_PROJECTION (S'0, PID, LOC)
= VISIBLE_PROJECTION (S'1, PID, LOC)),
all LOC'O:
  IMPLIES (GOOD_LOCATION (LOC'O),
  AND (LEVELS_PROJECTION (S'0, PID, LOC'O)
  = LEVELS_PROJECTION (S'1, PID, LOC'O),
  VISIBLE_PROJECTION (S'0, PID, LOC'O)
  = VISIBLE_PROJECTION (S'1, PID, LOC'O))),
  AND (VALUES (ORESET (KR, S'O), PID) = VALUES (S'1, PID),
  IMPLIES (GOOD_LOCATION (LOC),
  AND (LEVELS_PROJECTION (ORESET (KR, S'O), PID, LOC)
  = LEVELS_PROJECTION (S'1, PID, LOC),
  VISIBLE_PROJECTION (ORESET (KR, S'O), PID, LOC)
  = VISIBLE_PROJECTION (S'1, PID, LOC)))))

!reduce without instantiation;

Which simplifies
with invocation of SEC_LEVEL'LE, VISIBLE_PROJECTION, LEVELS_PROJECTION,
SEC_LEVEL'LT
when rewriting with VISIBLE_ON_ORESET, LEVELS_ON_ORESET, VALUES_ON_ORESET,
SEC_LEVEL'TOP_SECRET'VALUE, HIGHEST_SEC_LEVEL'VALUE
with the assumptions SEC_LEVEL'ORD_OF_VAL'AX, SEC_LEVEL'VALUES'AX,
SEC_LEVEL'LT SEC_LEVEL'TOP_SECRET'VALUE to
...
TRUE

The axioms for oWRITE and oREAD are similarly presented. For the oWRITE axiom, the proof steps are identical to
the foregoing; the oREAD axiom requires one further instantiation.

The next axiom is a main lemma expressing the property that a purgeable instruction preserves process view. The proof
trivially follows from the previous three theorems.

axiom PURGEABLE_INSTRUCTION_PRESERVES_PROCESS_VIEW (KR, S'O, S'1, PID) =
rule
begin IMPLIES (AND (GOOD_KERNEL_REQUEST(KR),
  GOOD_PID(PID),
  SEC_LEVEL'LT (PID_LEVEL (PID), PID_LEVEL (PROCESS (KR)))),
  PROCESS_VIEW_IDENTICAL (PID, S'O, S'1)),
  PROCESS_VIEW_IDENTICAL (PID, STEP (KR, S'O), S'1))
end PURGEABLE_INSTRUCTION_PRESERVES_PROCESS_VIEW;

!invoke STEP;
!rewrite;

Using the next three theorems, it is shown that PRESERVES_PROCESS_VIEW is maintained by applying an instruction
to equivalent states. The proof structure parallels that of the previous set of theorems.

!axiom oreset_PRESERVES_PROCESS_VIEW (KR, S'O, S'1, PID) =
rule
begin IMPLIES (AND (GOOD_KERNEL_REQUEST(KR),
  GOOD_PID(PID),
  PROCESS_VIEW_IDENTICAL (PID, S'O, S'1)),
  PROCESS_VIEW_IDENTICAL (PID, oreset (KR, S'O), oreset (KR, S'1)))
end oreset_PRESERVES_PROCESS_VIEW;
!invoke PROCESS_VIEW_IDENTICAL;
!trivial simplify;
The axioms for oWRITE and oREAD are similarly presented. For the oWRITE axiom the proof steps are identical to
the foregoing; the oREAD axiom requires one further instantiation.

The next axiom is a main lemma expressing the property that the application of an instruction preserves process view. The proof trivially follows from the previous three theorems.

The axioms for oWRITE and oREAD are similarly presented. For the oWRITE axiom the proof steps are identical to
the foregoing; the oREAD axiom requires one further instantiation.

The next axiom is a main lemma expressing the property that the application of an instruction preserves process view. The proof trivially follows from the previous three theorems.

The axioms for oWRITE and oREAD are similarly presented. For the oWRITE axiom the proof steps are identical to
the foregoing; the oREAD axiom requires one further instantiation.

The next axiom is a main lemma expressing the property that the application of an instruction preserves process view. The proof trivially follows from the previous three theorems.

The axioms for oWRITE and oREAD are similarly presented. For the oWRITE axiom the proof steps are identical to
the foregoing; the oREAD axiom requires one further instantiation.

The next axiom is a main lemma expressing the property that the application of an instruction preserves process view. The proof trivially follows from the previous three theorems.

The axioms for oWRITE and oREAD are similarly presented. For the oWRITE axiom the proof steps are identical to
the foregoing; the oREAD axiom requires one further instantiation.

The next axiom is a main lemma expressing the property that the application of an instruction preserves process view. The proof trivially follows from the previous three theorems.

The axioms for oWRITE and oREAD are similarly presented. For the oWRITE axiom the proof steps are identical to
the foregoing; the oREAD axiom requires one further instantiation.

The next axiom is a main lemma expressing the property that the application of an instruction preserves process view. The proof trivially follows from the previous three theorems.

The axioms for oWRITE and oREAD are similarly presented. For the oWRITE axiom the proof steps are identical to
the foregoing; the oREAD axiom requires one further instantiation.

The next axiom is a main lemma expressing the property that the application of an instruction preserves process view. The proof trivially follows from the previous three theorems.
\textbf{32}

\section*{4 Comparison with GVE and Boyer-Moore}

The top level structure of the LWM proof is modelled on Young's GVE solution \cite{KY87a}. However, due to the stronger prover embedded in m-EVES, I was able to reduce the number of lemmas, relating to \textsc{ProcessViewIdentical} and introduce the package abstraction as described above. As a consequence, the above proof is, I believe, more elegant than the GVE solution. Further, the proof appears to be more concise than the Boyer-Moore solution; there are fewer low level concerns here and the strong typing of m-Verdi nullifies the need for a number of propositions occurring in the Boyer-Moore solution.

In \cite{KY87a} Kaufmann and Young compare Boyer-Moore and the GVE in four areas: specification style, proof management, style of interaction with the prover, and soundness. Each of these areas is discussed in turn. The quotations are from \cite{KY87a}.
4.1 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."

It was also noted in [KY 87a] that an early Boyer-Moore specification of the LWM consisted of an error which would have been caught by Gypsy's type checking.

m-Verdi is similar to Gypsy; it combines linguistic constructs for expressing specifications and imperative programs, and neither language allows for a global state. m-Verdi is specification of the LWM consisted of an error which would have been caught by Gypsy's type checking.

m-Verdi, however, does have a formalism that is not as expressive as Gypsy in that there are fewer types and no handling of concurrency. m-Verdi, however, does have a solid treatment of functional recursion. As noted below, the expressiveness of m-Verdi will be significantly enhanced.

4.2 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."

m-NEVER also allows proofs to be deferred and guarantees that there will not be any circularity in proofs (see §4.4). As with Boyer-Moore, m-NEVER also supports the addition of heuristic content to propositions (e.g., whether a proposition is to be treated as a forward rule—not possible in Boyer-Moore—or as a rule).

4.3 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 considerably 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 definitions and proof commands. But as mentioned above, the Gypsy system is much more flexible about the order in which one gives proofs."

The power of m-NEVER is comparable to that of Boyer-Moore. As our proof of the LWM has indicated, fairly large proof steps are possible using m-NEVER and, I would claim, the above presentation has, on the most part, focused on the fundamental aspects of the LWM—there are few low-level concerns.

One must agree with Young and Kaufmann on the negative aspects of using heuristic provers; such provers can be difficult to control. This problem has been ameliorated, somewhat, by m-NEVER's capability to allow for user interaction during the proof process. m-NEVER synergistically combines the advantages of heuristic provers with the introduction of pertinent information from the system user. m-EVES supports the modifications of proofs and allows the replay of prover commands.

4.4 Soundness

"The Boyer-Moore logic . . . 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 . . . 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."

The m-EVES system is based upon a provably sound logic system [Saa 87]. In fact, the development of m-Verdi was strongly influenced by our principal requirement that the language have a complete formal semantic characterization and a provably sound logic. The mathematical framework of m-EVES is more complicated than that of Boyer-Moore primarily because of our inclusion of imperative constructs. m-EVES indicates which propositions are unproved and guarantees that there will be no circularity of proofs. m-EVES does not, however, have the same empirical evidence for error free performance as Boyer-Moore."
5 Conclusion

As reported in [EVES 88], the m-EVES project has resulted in the following:

- The design and implementation of a new specification and implementation language, m-Verdi, which has a complete, formal mathematical basis and supporting logic.
- The design of a sound logical system which is sufficiently powerful to handle linguistic features commonly used in the formal verification process.
- The development of a new theorem prover, m-NEVER, which incorporates significant functionality drawn from the theorem proving literature.
- The development of production quality compilers for m-Verdi.

A version of m-EVES was released in mid-November. After porting the system to Common Lisp, it is intended that packages and environments will be implemented. A compiler for m-Verdi has been written and runs on VAX computers under the VMS or UNIX operating systems [Mei 87].

With the completion of m-EVES, our research will be directed at improving the expressibility of m-Verdi (resulting in Verdi); upgrading the mathematics to mirror the changes to the language; writing a compiler for Verdi; continuing the evolution of theorem prover capability (resulting in NEVER); and continuing the evolution of the m-EVES interface.

6 Acknowledgements

My thanks to Bill Young for introducing this problem to me and to Sentot Kromodimoeljo who helped with various experimental versions of solutions to the LWM problem. The core technical people involved in the development of m-EVES were Sentot Kromodimoeljo, Irwin Meisels, Andy Neilson, Bill Pase, Mark Saaltink and myself, Dan Craigen.

A Prover Commands

The description of the commands presented here is a modified version of the on-line help available from m-EVES. Only those commands which have been used in the example are described here.

A.1 Instantiate instantiations

Performs the given INSTANTIATIONS on the current formula. To allow the instantiations to occur, the scopes of quantifiers in the formula may be extended or contracted. Logical equivalence is maintained by keeping the uninstantiated subexpression(s) as extra conjunct(s) or disjunct(s). The requested instantiations may be disallowed, in which case the command has no effect.

A.2 Invoke (expression | function_symbol)

Opens the definition of a function wherever it occurs within the formula. If the object being invoked is a function with a precondition then the invocation is a conditional form on the precondition, the expanded form, and the original form. invoke may also be applied with an expression rather than a function_symbol, in which case it works like a selective invoke in that occurrences of the expression in the formula are replaced by the expanded version.

A.3 Induct on term

Tries to apply an induction scheme to the current formula. The induction technique used is that of Boyer-Moore. Normally, the induction scheme is heuristically chosen based on calls to recursive functions within the current formula. However, the user may direct the prover by explicitly specifying the TERM on which to induct.

A.4 Reduce

Reduces the current formula. Reduction consists of simplification, rewriting, and invocation. Reduction forms the kernel of the automatic prover.

A.5 Reduce without instantiation

Reduces the current formula without performing any instantiation of quantified variables. This command performs a subset of the REDUCE command.

A.6 Rewrite

Rewrites and simplifies the current formula. Conditional rewrite rules may be applied, provided their condition can be proven using only simplification and rewriting. This command also applies any forward rules which are triggered and whose condition is provable.

A.7 Simplify

Simplifies the current formula. This may perform the substitution of equalities as well as trying to instantiate variables in order to find a proof.

The process of simplification attempts to reduce an expression to one the system considers to be simpler. In general, though not always, the resulting simplified expression is smaller than the original. However, propositional tautologies are always detected. In addition, the simplification process reasons about equalities, integers, and quantifiers.

A.8 Trivial simplify

Performs a simple tautology check and propositional simplification.
B  Generic Sequence Theory Kernel

The following version of sequence theory is the core of the theory that has been developed for m-EVES. Very careful attention has been directed at expressing the axioms in a has been developed to show the consistency of the axiom set (assuming only that the type "XX" is countable). This proof was completed using m-EVES.

type SEQUENCE_XX;
var IO_XX, II_XX, I2_XX: INT;
var EO_XX, EI_XX, E2_XX: XX;
var S0_XX, S1_XX, S2_XX: SEQUENCE_XX;
const EMPTY_XX: SEQUENCE_XX;
function TACK_XX (EO_XX, SO_XX): SEQUENCE_XX;
function HEAD_XX (SO_XX): XX;
function TAIL_XX (SO_XX): SEQUENCE_XX;

axiom HEAD_TACK_XX (SO_XX, EO_XX) =
frule
triggers (TACK_XX (EO_XX, SO_XX))
begin
  HEAD_XX (TACK_XX (EO_XX, SO_XX)) = EO_XX
end HEAD_TACK_XX;

axiom TAIL_TACK_XX (SO_XX, EO_XX) =
frule
triggers (TACK_XX (EO_XX, SO_XX))
begin
  TAIL_XX (TACK_XX (EO_XX, SO_XX)) = SO_XX
end TAIL_TACK_XX;

axiom TACK_HEAD_TAIL_XX (SO_XX) =
frule
triggers (HEAD_XX (SO_XX),
TAIL_XX (SO_XX))
begin
  IMPLIES (SO_XX <> EMPTY_XX,
TACK_XX (HEAD_XX (SO_XX),
TAIL_XX (SO_XX)) = SO_XX)
end TACK_HEAD_TAIL_XX;

axiom TACK_NOT_EMPTY_XX (EO_XX, SO_XX) =
frule
triggers (TACK_XX (EO_XX, SO_XX))
begin
  NOT (TACK_XX (EO_XX, SO_XX) = EMPTY_XX)
end TACK_NOT_EMPTY_XX;

function SIZE_XX (SO_XX): ORDINAL;

axiom SIZE_TAIL_XX (SO_XX) =
rule
begin
  IMPLIES (SO_XX <> EMPTY_XX,
ORDINAL'LT (SIZE_XX (TAIL_XX (SO_XX)),
SIZE_XX (SO_XX)))
end SIZE_TAIL_XX;

function LENGTH_XX (S0_XX): INT = measure SIZE_XX (S0_XX)
begin
  if S0_XX = EMPTY_XX
     then 0
     else PLUS (1, LENGTH_XX (TAIL_XX (S0_XX)))
end if
end LENGTH_XX;

axiom LENGTH_NON_NEGATIVE_XX (SO_XX) =
frule
triggers (LENGTH_XX (SO_XX))
begin
  INT'GE (LENGTH_XX (SO_XX), 0)
end LENGTH_NON_NEGATIVE_XX;

axiom LENGTH_TEST_XX (SO_XX, S1_XX) =
rule
begin
  IMPLIES (LENGTH_XX (SO_XX) <> LENGTH_XX (S1_XX),
NOT (SO_XX = S1_XX))
end LENGTH_TEST_XX;

References


