The SDOS project – Verifying Hook-up Security

D.G. Weber
Bob Lubarsky

Odyssey Research Associates, Inc.
301A Harris B. Dates Drive
Ithaca, NY 14850-1313
607 277 2020

Abstract

Verification of multi-level security for a distributed, object-oriented system is made easier if the system security policy is a hook-up *property: the property must be true for a system if it is true for each communicating component of that system. The work of formal verification for the entire system may then be divided into independent subtasks, each task being the verification of the hook-up property for one system component.*

In this paper, work toward design and verification of a multi-level secure distributed operating system is described. A technique is presented for verifying a particular hook-up security property using the Gypsy Verification Environment.

1 Introduction

This work was undertaken during the Secure Distributed Operating System Project (SDOS), now ongoing at Odyssey Research Associates (ORA), and at BBN Laboratories. The SDOS project has several goals:

1. the exploration of multi-level security (MLS) for a distributed operating system (DOS);
2. the development of a top-level design for a DOS based on the object-oriented principles used in the design of the Cronus DOS [8] developed at BBN Labs;
3. verification of the top-level design against properties given in the SDOS security policy.

The top-level design for SDOS is being expressed in the formal specification language of the Gypsy Verification Environment [4]. That language, also called Gypsy, is a Pascal-like programming language in which program correctness is expressed by embedded assertions.

The SDOS security policy contains an MLS component which is taken to be the hook-up security property of McCullough [7]. This security policy guarantees that proof of multi-level security for the entire system can be had by proving hook-up security for its communicating components. This guarantee is particularly appropriate for security verification of an object-oriented system. We have found that this property is not directly expressible by Gypsy embedded assertions. However, we have also found that certain designs, expressible in Gypsy, may be transformed so that the verification conditions produced by the Gypsy Verification Environment imply the hook-up security property.

In section 2 we will review briefly some aspects of the object-oriented DOS design chosen for SDOS. Section 3 will outline the SDOS security policy, dividing the constraints of that policy into three categories: mandatory, discretionary, and configuration constraints. Section 4 forms the bulk of the paper; in it is discussed a technique for verifying a design expressed in Gypsy against the constraints of the mandatory policy.

2 An Object-Oriented DOS Design

A primary goal of a distributed operating system is to provide global system resource management over a network of communicating computers. This means that system resources, such as operating system services and devices,

[0] This work was supported by the Air Force Systems Command at Rome Air Development Center under Contract No. F30602-85-C-0056. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Air Force or the U.S. Government.

should be controlled through a single, uniform facility for the entire DOS.

The designs of the Cronus DOS and SDOS both have emphasized object-oriented methodology, in particular the *object model* [6]. In this model, objects are instances of abstract data types and correspond to logically addressable resources, such as data and physical devices. The type of each object defines the set of operations on the object; these operations are the only means of accessing the object. Object operations are implemented by object managers, which hide the representation of the object from its accessors.

Each processor in the underlying SDOS network will be called a *host*.

Each service performed by SDOS consists of a client process invoking an (abstract) operation on an object of a particular abstract data type. Two kinds of system process participate in providing this service.

1. Message-switching processes facilitate the routing of message traffic between network hosts and between processes on a single host.

2. At least one manager process for the abstract data type is involved in performing the operation.

The basic structure of the system has one message switch process and several object manager processes per host.

Where must the enforcement for multi-level security reside? Clearly, if there are objects of several security levels on one host, then the message-switch must be trusted. The object manager processes need not be trusted if their only communication is via the message-switch. However, if all of the managers are untrusted, then when several operations are invoked on the same host on objects at different security levels, different manager processes must handle them, even if each manager process is functionally equivalent. This can result in inefficiency due to the overhead of process creation, and from swapping between a large number of object managers servicing different security levels. It may therefore be preferable to build some trusted, multi-level secure object managers. For SDOS, where "trust" is to be guaranteed by formal verification, this means that the message-switch and some subset of the object managers must be verified.

It is important to note that the system we have described is *extensible*. This means that the set of abstract data types and the set of operations defined on them are not fixed, and may be extended by adding new object managers. If a new manager is to handle invocations at more than one security level, its functionality must be verified with the same degree of rigor as the message-switch and other trusted managers.

Herein lies the advantage offered by verification of a hook-up security property: new trusted system components, either new hosts with trusted message-switches, or new trusted object managers, may be included in the system without reverification of already-existing components. This advantage becomes particularly great in an object-oriented distributed system.

3 Security Policy

The SDOS security policy [9] divides policy requirements into three types:

1. Discretionary requirements – These are access controls placed by users on the use of the abstract operations defined by SDOS. Because the set of abstract data types is extensible, and the set of abstract operations cannot be given in advance, the discretionary policy rules must be stated generically. The generic rules may then be applied to any data type.

2. Mandatory requirements – These are controls on information flow among the system's untrusted components which can occur through processing by the system's trusted components. The controls are defined, not as access controls, but as constraints on the possible extrinsic behavior of the system, that is, in terms of possible histories of message-passing interactions between the trusted and untrusted parts of the system. These constraints are dependent on the security levels of the untrusted components. They are designed to limit what users at one security level can deduce about actions taken by users at greater (or incomparable) security levels.

3. Configuration requirements – These are rules which constrain the roles of the system security administrators, and which govern how discretionary and mandatory controls apply in case of changes to the SDOS network configuration.

We have viewed the mandatory and discretionary policies as applying to two quite distinct kinds of interaction between the trusted and untrusted parts of the system. The discretionary policy applies to the use of the system-defined abstract operations, while the mandatory applies to the lower-level operations of sending and receiving messages. Even though these lower-level message-passing operations are the actions out of which the higher-level abstract operations are implemented, the mandatory policy is not simply a special case of the discretionary policy. They are policies on different levels of abstraction.

In what follows, we will focus on exclusively on the mandatory (MLS) policy requirements.

3.1 Traces

In order to define the mandatory MLS policy in terms of message-passing interactions, we have turned to a description of process behavior based on traces [2]. A trace is a history of a process' interaction with its environment. Associated with each process is a set of events in which it may engage. For our purposes, these events will be message-passing events between processes. Each trace is then a sequence of message-passing events. Process behavior is described as a set of possible traces.

Defining security in terms of possible traces of message-passing events will cause us to adopt a policy different in form from the widely-used policy of Bell and LaPadula [1]. The Bell-LaPadula policy is commonly given as requirements on the internal states of an abstract state machine. A policy on traces leaves the development of state machines to the system designer.

Notation for Traces We will need a few notations concerning sequences. Let E and F be sets of events, e a particular event, α and β sequences of events, and $f : E \mapsto F$ a mapping from E into F. Let l be an arbitrary security level, and suppose that each event is associated with some security level. We define:

- $\overline{E}$ is the set of events not in E;
- E^* is the set of all possible sequences formed from the events in E;
- $\alpha {}^\wedge \beta$ is the concatenation of α followed by β;
- $\alpha \sqsupseteq \beta$ iff β is an initial subsequence of α, while $\alpha \sqsupset \beta$ iff β is a proper initial subsequence;
- $last(\alpha)$ is the last event in α, while $nonlast(\alpha)$ is the trace with the last event removed;
- $\alpha \upharpoonright E$, the projection of α with respect to set E, is the sequence obtained from α by deleting all events not in E and preserving the order of the events that are left;
- $\alpha \upharpoonright l$ is the projection of α with respect to the set of events associated with security levels less than or equal to level l;
- f^* is the mapping from E^* into F^* of application of f componentwise;
- $\langle e \rangle$ is the trace with the single event e, and $\langle\rangle$ is the empty trace.

A process will be defined as the following structure of four sets, $\langle E, I, O, T \rangle$, where E is a set of events, $I \subseteq E$ a set of input events, $O \subseteq E$ a set of output events, and $T \subseteq E^*$ a set of traces. For any possible trace of the process, all initial subsequences are also traces:

$$\forall \alpha, \gamma \in E^*,\ \alpha {}^\wedge \gamma \in T \ \rightarrow\ \alpha \in T.$$

3.2 Hook-up Security

Definition 1 *A process $\langle E, I, O, T \rangle$ is* **input-total***, or simply* **total***, iff any trace may always be extended with any input. That is,*

$$\forall \alpha \in T, \forall x \in I,\ \alpha {}^\wedge \langle x \rangle \in T.$$

Definition 2 *A process $\langle E, I, O, T \rangle$ is* **restrictive with respect to level l** *iff it is input-total, and additionally, for any trace, if any block of inputs to that trace is modified without altering the sequence of inputs visible at level l, then there is another trace for which future behavior is modified, which contains no future inputs at higher levels, and which exhibits the same behavior as the original trace at levels below l. Formally,*

$$\begin{aligned} &\forall \alpha, \gamma \in E^*, \forall \beta, \beta' \in I^*,\ \alpha {}^\wedge \beta {}^\wedge \gamma \in T \ and \\ &\qquad\qquad \beta \upharpoonright l = \beta' \upharpoonright l \ \rightarrow \\ &\exists \gamma' \in E^*,\ \alpha {}^\wedge \beta' {}^\wedge \gamma' \in T \ and \\ &\qquad \gamma' \upharpoonright l = \gamma \upharpoonright l \ and \\ &\qquad \gamma' \upharpoonright I \upharpoonright \bar{l} = \langle\rangle. \end{aligned}$$

This is the basic property defined by McCullough. Its intent is roughly as follows: even if one pooled all information based on behavior of the process at level l or below, one's ability to deduce the existence or non-existence of inputs at higher (or incomparable) levels is limited. Given a particular set of inputs at higher levels, there is at least one other trace with identical behavior at level l and below which masks the pattern of the higher-level inputs. Note that this definition cannot rule out deducibility based on knowledge of the relative probability of traces, or on the times at which events in a trace occur.

Two processes, viewed as executing in parallel, may in some cases be hooked together to form another process. If an input of one process is also an output event of the other, then this mutual event is neither an input nor an output of the hooked-up process but is an internal communication event. We must rule out the possibility that events shared by two processes are not communication events of this form.

Definition 3 *Processes $P_1 = \langle E_1, I_1, O_1, T_1 \rangle$ and $P_2 = \langle E_2, I_2, O_2, T_2 \rangle$ are* **coherent** *if*

$$E_1 \cap E_2 = (I_1 \cap O_2) \cup (I_2 \cap O_1).$$

Definition 4 *The* **hook-up** *of two processes, $P_1 = \langle E_1, I_1, O_1, T_1 \rangle$ and $P_2 = \langle E_2, I_2, O_2, T_2 \rangle$, is defined if P_1 and P_2 are coherent and yields a new process $P_1 \| P_2 = \langle E, I, O, T \rangle$ with*

$$\begin{aligned} &E = E_1 \cup E_2 \\ &I = I_1 \cup I_2 \ - I_1 \cap O_2 \ - I_2 \cap O_1 \\ &O = O_1 \cup O_2 \ - O_1 \cap I_2 \ - O_2 \cap I_1 \\ &\forall t \in E^*, t \in T \ \leftrightarrow\ (t \upharpoonright E_1) \in T_1 \ and \ (t \upharpoonright E_2) \in T_2 \end{aligned}$$

The paper by McCullough shows that the hookup of two processes, each restrictive with respect to level l, is another process also restrictive with respect to level l. This is the justification for calling restrictiveness a hookup property.

We will take a multi-level secure process to be one which is restrictive with respect to every level l. Multi-level security is then also a hook-up property.

4 Verification Using Gypsy

Consider what it might mean to require the property of restrictiveness of a Gypsy procedure. We will not attempt to relate formally the semantics of processes described above to the semantics of Gypsy procedures. Instead, we will argue informally that there is a connection. We have modeled the design of SDOS in Gypsy as a collection of cobegun procedures communicating via buffers. We want to associate the input and output events described above with the (Gypsy) actions of sending and receiving from a buffer. Assertions about traces will then become associated with assertions about (Gypsy) buffer histories.

How might one verify in Gypsy the hook-up property of restrictiveness with respect to an arbitrary level l? There are two basic problems:

1. The Gypsy embedded-assertion approach to stating correctness conditions will only allow assertions about the properties of single traces in isolation. More precisely, Gypsy embedded assertions are all of the form:

$$\forall \alpha \in T, P(\alpha)$$

   where P is some predicate over event sequences which contains no quantifier over event sequences. Note that each program variable is a function of the past sequence of events, and so relations among program variables fall into this form. The property of restrictiveness is more complicated, since it requires one to show the *existence* of a trace, given the existence of another trace. The embedded-assertion method is not designed to do this.

2. The property of restrictiveness requires a process to be input-total, i.e., always ready to accept another input. This is never true of procedures described in Gypsy, which accept inputs only at "receive" statements.

The solution to the second problem is to associate one or more unbounded buffers with each Gypsy procedure of the design. The combination of a Gypsy procedure and its associated buffers forms an input-total process.

The solution we give here to the first problem has two parts. First, we define several simpler requirements, which, when placed on a process, imply that the property of restrictiveness holds for the combination of that process and its associated buffers. The most important of these will be called "weak non-interference" (WNI). This property is very similar to the Goguen-Meseguer security policy [3], in that it prevents one from deducing that unseen higher-level inputs have occurred. It differs from Goguen-Meseguer in that: determinism is not assumed; it is defined purely in terms of the set of traces of the process rather than in terms of internal states; responses to inputs are not required to happen immediately after the inputs which caused them. We show that weak non-interference, plus determinism, plus the property that a trace must exist in response any set of inputs, plus the coupling of a process with unbounded buffers, is sufficient to imply the property of restrictiveness.

Second, we show how to prove weak non-interference in certain particular cases. The technique presented is not general; it is easy to find examples of Gypsy procedures satisfying WNI which cannot be handled in this way. However, we have been able to apply it successfully to several examples of trusted object managers in the SDOS design.

4.1 Inferring restrictiveness from WNI

Definition 5 *A process $\langle E, I, O, T\rangle$ satisfies the property of* **weak non-interference (WNI) with respect to level l** *iff for every trace, there is another trace all of whose inputs are visible below level l, and which exhibits the same visible behavior.*

$$\forall \alpha \in T,\, \exists \alpha' \in T,\, \alpha' \uparrow l = \alpha \uparrow l \text{ and } \alpha' \uparrow I \uparrow \bar{l} = \langle\rangle.$$

The property of weak non-interference is weaker than restrictiveness, and it is not a hook-up property.

Definition 6 *A process $\langle E, I, O, T\rangle$ is* **input-live** *iff the set I is nonempty and any trace may be properly extended into another trace ending in an input. That is,*

$$\forall \alpha \in T,\, \exists \alpha_x \in T,\, \alpha_x \sqsupset \alpha \text{ and } last(\alpha_x) \in I.$$

Definition 7 *A process $\langle E, I, O, T\rangle$ is* **input-universal** *iff, when it is possible to accept some input, any input may be accepted. That is,*

$$\forall \alpha \in T,\, \forall e_1, e_2 \in I,\, \alpha^\wedge\langle e_1\rangle \in T \;\rightarrow\; \alpha^\wedge\langle e_2\rangle \in T.$$

Definition 8 *A process $\langle E, I, O, T\rangle$ is* **deterministic** *iff*

$$\forall \alpha \in E^*,\, \forall e_1, e_2 \in E,\, \alpha^\wedge\langle e_1\rangle \in T \text{ and } \alpha^\wedge\langle e_2\rangle \in T \;\rightarrow\; (e1, e2 \in I) \text{ or } (e1, e2 \in \bar{I} \text{ and } e_1 = e_2).$$

Note that this form of determinism is too strong to be applied to input-total processes. An input-total and deterministic process would never output anything.

Why is WNI weaker than restrictiveness? WNI allows one to remove all high-level inputs from a trace, without disturbing low-level behavior. Restrictiveness, on the other hand, allows one either to remove or insert such inputs. The following theorem shows that adding auxiliary conditions to WNI will strengthen it sufficiently to allow arbitrary high-level inputs to be inserted in a trace. The theorem says that if a process satisfies all the previously-stated properties of this section, then it is possible to alter its sequence of high-security-level inputs arbitrarily without altering its low-security-level inputs and outputs (although the output sequence may need to be extended).

Theorem 1 *If the process $\langle E, I, O, T\rangle$ is input-live, and -universal, and is deterministic and satisfies weak non-interference with respect to level l, then*

$$\begin{array}{l} \forall \alpha, \gamma \in E^*,\, \forall \beta \in I^*,\; \alpha {}^\wedge \gamma \in T \; and \\ \qquad\qquad \beta \uparrow l = \gamma \uparrow I \uparrow l \;\rightarrow \\ \exists \gamma' \in E^*,\; \alpha {}^\wedge \gamma' \in T \; and \\ \qquad \gamma' \uparrow I = \beta \; and \\ \qquad \gamma' \uparrow l \sqsupseteq \gamma \uparrow l \end{array}$$

This and the following theorem will be given without proof. However, the proof of theorem 1 proceeds roughly as follows. We are given a trace $\alpha {}^\wedge \gamma$, and wish to alter some of the high-security-level inputs to γ. Input-liveness and input-universality imply that there must exist a new trace, $\alpha {}^\wedge \gamma'$, in response to these new inputs. WNI applied to $\alpha {}^\wedge \gamma$ and to $\alpha {}^\wedge \gamma'$, gives new traces with the same low-security-level behavior and no high-security-level inputs. Therefore these traces have exactly the same inputs. Determinism implies that they have the same behavior, and therefore that γ and γ' exhibit the same behavior at low security-levels.

Definition 9 *The process $\langle E, I, O, T\rangle$ is an* **infinite buffer** *iff there is a one-to-one and onto mapping $map : I \mapsto O$, and*

$$\forall \alpha \in E^*,\, \alpha \in T \;\leftrightarrow\; \forall \alpha' \sqsubseteq \alpha,\; map^*(\alpha' \uparrow I) \sqsupseteq \alpha' \uparrow O$$

In addition, the buffer **preserves level l** *iff*

$$\forall x \in I,\; x \in l \;\leftrightarrow\; map(x) \in l$$

Note that every infinite buffer is input-total, and if it preserves level l, then it is also restrictive.

In the following theorem, these objects will appear:

- an arbitrary level l;
- a nonempty array of n infinite buffers which preserve l, $B_1, \ldots, B_n$, with buffer $B_i = \langle E_i, I_i, O_i, T_i\rangle$, $I_i \cap O_i = \{\}$ and $i \neq j \;\rightarrow\; E_i \cap E_j = \{\}$;
- process $A = \langle E, I, O, T\rangle$, with $I = \bigcup_{(i=1,\ldots,n)} O_i$ and $E - I \cap E_i = \{\}$;
- process $W = A \| B_1 \| \ldots \| B_n$, the concurrent hookup of all the processes. $W = \langle EW, IW, OW, TW\rangle$ where
  - $EW = E \cup \bigcup_{(i=1,\ldots,n)} E_i$;
  - $IW = \bigcup_{(i=1,\ldots,n)} I_i$;
  - $OW = O$;
  - $t \in TW$ iff $t \uparrow E \in T$ and $t \uparrow E_i \in T_i$ for all i.

Theorem 2 *If A is input-live, -universal, deterministic, and satisfies WNI with respect to l, then W is restrictive with respect to l.*

This theorem is the result which allows us to convert a proof of restrictiveness, (and hence of hook-up security), into simpler proofs of WNI, determinism, input-liveness, and -universality. The process W, which is the hook-up of A with a set of infinite buffers, will be input-total since each buffer is. The process A need not be input-total, and when described in Gypsy, it will never be. As in theorem 1, the auxiliary assumptions on A allow the inputs to W to be modified without altering its visible behavior. The proof of this theorem can be found elsewhere [10].

4.2 Relating the Theorem to Gypsy

Suppose a Gypsy program is composed of a cobegin of procedures and a collection of infinite Gypsy buffers visible to those procedures. Each Gypsy buffer will be associated with exactly one Gypsy procedure; a procedure will only receive messages from buffers with which it is associated. Further suppose that each procedure is of the form:

```
loop
  await
    each i:integer,
      on receive mesg from buffer(i)
      then
        body(mesg);
  end;
end;
```

where each buffer[i] is an unbounded Gypsy buffer associated with this procedure. The code ‘body’ contains no “receives”, and it may contain “sends” to buffers associated with other procedures.

Then we may (informally) apply theorem 2 to this situation, treating each buffer as a process, and treating the Gypsy events of “sending” and “receiving” from buffers as the events referred to in the theorem. The communication between the procedure and its associated input buffers is assumed to obey the rules described in the definition of hook-up. We can then claim that this procedure and the collection of buffers associated with it are restrictive with respect to level l if we show:

- that each buffer preserves l: this holds trivially if we give levels to messages in each buffer. Let a predicate $dominated - by - l$ be defined on the set of messages, and let a buffer "send" or "receive" be an event at a level less than or equal to l if and only if the message sent or received satisfies $dominated - by - l$;
- input-universality: this is guaranteed since all input events occur via the 'await' statement;
- input-liveness: this will be guaranteed if can show that 'body' may terminate for each possible set of preconditions;
- determinism: this is guaranteed in any Gypsy procedure which makes no cobegins, either directly or indirectly, through calls to other procedures. This can be checked purely syntactically;
- weak non-interference with respect to level l for the procedure: demonstrating this will be the goal of the following section.

We have previously noted that WNI is similar in flavor to the Goguen-Meseguer security policy. A paper by Haigh and Young [5] used an unwinding theorem to reduce the Goguen-Meseguer policy to assertions on Gypsy variables. The variables represented the abstract state of the Secure Ada Target (SAT). We chose not adapt this approach to WNI, but rather attempted to verify the WNI property directly.

4.3 Verifying WNI Using Gypsy

Weak non-interference states that for every trace α there is another trace α' which bears a certain relation to α. Showing that α' exists will require, in effect, running two copies of the procedure simultaneously – the real one with the complete history of inputs, and another one with no high-security-level inputs. The latter will be referred to as the purged history, and it will be generated from the input sequence $\alpha \uparrow I \uparrow l$. The verification is then a demonstration that the complete and purged histories have identical behavior at low security levels, i.e., that $\alpha \uparrow l = \alpha' \uparrow l$.

We will need to transform a design expressed in Gypsy into another Gypsy program with the appropriate assertions. The reason for the program transformation is to make two copies of every variable in the original design. Not only internal program variables, but also buffer variables must be duplicated. In this way we can generate simultaneously both the actual run of the design, and the purged history. The assertions given in the transformed program will include at least the statement that the contents of buffer histories, taking only events less than or equal to level l, will be equal.

We assume that the procedure is of the form

```
loop
  await
    each i:integer,
      on receive message from buffer(i)
      then
        body;
  end;
end;
```

where 'body' has no receives. In practice this seems to be a simple but useful form in Gypsy for modeling a non-terminating procedure communicating with other entities. We consider Gypsy program fragments body composed of the commands :=, if then else, send, case, loop, and subprocedure calls. Each procedure will be a branch of a cobegin, but cobegin statements will not be allowed within 'body', since such a cobegin will introduce non-determinism and thus invalidate one of the assumptions of theorem 2.

Given such a program, the way to generate the purged history is simply to ignore those inputs of high- (or not comparable-) security level:

```
loop
  await
    each i:integer,
      on receive message from buffer(i)
      then
        if message.level le l then
          body;
        end;
  end;
end;
```

Since we want to compare histories, we must run these programs simultaneously:

```
loop
  await
    each i:integer,
      on receive message from buffer(i)
      then
        body;
        if message.level le l then
          MESSAGE' := message;
          INSEQ' := INSEQ' :< MESSAGE';
          BODY';
        end;
  end;
end;
```

where body' is the same as body but with a disjoint, primed, set of variables, whose values simulate the state under the purged history. Inseq' simulates the input history associated with the purged history.

We must show that the two communication histories for this procedure, purged for level l, are the same. If there is only one output buffer, and if **purge(b,l)** is a Gypsy function which represents the projection of event sequence b with respect to l, i.e., $b \uparrow l$, then it is enough to verify the loop assertion:

```
assert
  purge(outto(outbuf,myid), l) =
  purge(outto(outbuf',myid), l);
```

Because this assertion must hold at each input in the history, the purged input sequence must be interleaved with the purged output sequence in the same manner in both histories. It is therefore sufficient even though no explicit mention of inputs is made.

If there is more than one output buffer, it is not sufficient to prove that the low-level outputs to each are the same in the two histories, since relative order of outputs is important. Gypsy implicitly provides for merging of buffer histories; however, its VC generation facilities treat this merging as though it were based on order of arrival at the buffer processes, not on the order of departure from the procedure itself. To handle more than one output buffer, new sequence variables must be introduced to record the relative order of outputs at the sending procedure. (Note: delays between the time a message is sent by a process and the time it arrives at a buffer violate the assumptions given in the definition of hook-up. However, new infinite buffer processes preserving l may be introduced to account for the delay while retaining the semantics of hook-up. Since these new infinite buffer processes are restrictive, their presence will not change the security properties of the entire system.)

In principle, proving the assertions described above is all that is required. However, in practice auxiliary assertions on internal variables are useful. It will be sufficient to show that, at all comparable places in the programs body and body', certain variables from each (e.g. those $\leq l$) are related (e.g. equal). So not only must the loops be run simultaneously (body and body' within one loop), the bodies themselves must be run simultaneously. Since they refer to disjoint sets of variables, their steps can be intertwined in any manner, so long as the order within each program is unchanged. We will give some practical methods for intertwining body and body' and generating useful auxiliary assertions. While this intertwining is not strictly necessary for proof, it will almost always reduce the number of execution paths to be handled by the Gypsy VC generator.

First we must move body inside the scope of the if statement:

```
loop
  await
    each i:integer,
      on receive message from buffer(i)
      then
        if message.level < l then
          MESSAGE' := message;
          INSEQ' := INSEQ' :< MESSAGE';
          body;
          BODY';
        else body;
        end;
  end;
end;
```

We now describe one inductive method for unifying the program fragments body and body' into a new program fragment mergedbody, which has the same effect on all program variables but contains auxiliary assertions relating the primed and unprimed variables. This method produces assertions which may be sufficient, but are not necessary, for the verification of the assertions on buffer histories. The description of the method is phrased as an algorithm that inputs body and outputs mergedbody.

To Merge The following algorithm for merging program bodies is defined inductively on the statements in body:

If the next step in body is **a:=b**, do in mergedbody: **a:=b; a':=b'**.

If the next statement in body is **send x to y**, then do in mergedbody **send x to y; send x' to y'**.

If the next statement is an if-then-else, let it be of the form:

```
if x then y;
     else z;
```

Do in mergedbody

```
assert x iff x';
if x then y; y';
     else z; z';
```

If the next statement is a loop, let it be of the form:

```
loop
  if x then y;
  else z; leave;
  end;
end;
```

Do in mergedbody

```
loop
  assert x iff x';
  if x then y; y';
  else z; z'; leave;
  end;
end;
```

If the next statement is a case statement:

```
case x
...
is yi then zi;
...
```

do

```
assert x=yi iff x'=yi';
case x
...
is yi then zi; zi';
...
```

Finally, if the next statement in body is subprocedure call SP(var), then do

```
SP(var);
SP(var');
```

To Finish The program bodies are merged to the extent that the new assertions generated can be verified. More sophisticated steps can be added to the algorithm for special cases. The final program which results is

```
loop
  await
    each i:integer,
      on receive message from buffer(i)
      then
        if message.level le l then
          mergedbody;
        else
          body;
        end;
  end;
end;
```

5 Conclusions

We have presented a method for verifying an MLS hook-up security property using Gypsy. The MLS property will form the basis for mandatory security in a Secure Distributed Operating System (SDOS). The verification method reduces a proof of the McCullough hook-up security property to proof of a non-interference property (WNI), plus auxiliary assumptions; a technique for demonstrating WNI in Gypsy is given for certain cases.

Using this method, we have been able to verify WNI, and thus hook-up security, for Gypsy descriptions of some SDOS object managers. The primary drawback of the approach is the need to transform a design expressed in Gypsy into an alternate form in which assertions can be given. This is tedious and error-prone; presumably a VC generator which automated the step of program transformation would eliminate this drawback.

Future work along the lines given in this paper will involve generalizing the notions and tools mentioned. In particular, the MLS policies used, restrictiveness and WNI, rule out all covert channels not based on timing information, and therefore include no means for describing limited amounts of information downgrading. However, some limited covert channels are usually tolerated in any detailed system design. The policies must be generalized to include this possibility.

6 Acknowledgement

The authors would like to thank Daryl McCullough of ORA for insight into the meaning of hook-up security, and Steve Vinter of BBN for many useful discussions on distributed system design.

References

[1] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, Revision 2, MITRE Corp., Bedford, MA, Mar. 1976.

[2] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3), 1984.

[3] J. Goguen and J. Meseguer. Security policy and security models. In *IEEE Symp. Security and Privacy*, 1982.

[4] D. Good et al. Report on Gypsy 2.05. Technical report, Computational Logic, Inc., Austin, TX, Oct. 1986.

[5] J. Haigh and W. Young. Extending the non-interference version of MLS for SAT. In *IEEE Symp. Security and Privacy*, 1986.

[6] A. Jones. The object model: A conceptual tool for structuring software. In Bayer, Graham, and Seegmuller, editors, *Operating Systems, An Advanced Course*. Springer-Verlag, 1978. Lecture Notes in Computer Science.

[7] D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symp. Security and Privacy*, 1987.

[8] R. Schantz and R. Thomas. CRONUS, a distributed operating system: Functional definition and system concept. Technical Report 5879, BBN Labs, Jan. 1985.

[9] S. T. Vinter, D. Weber, et al. A secure distributed operating system. In *IEEE Symp. Security and Privacy*, 1988.

[10] S. T. Vinter, D. Weber, et al. The secure distributed operating system project. Technical Report 6144, BBN Labs and Odyssey Research Associates, Feb. 1988.