UCRL-CR--105098

DE91 007138

Secure Resource Management:
Specifying and Testing Secure Operating Systems

Myla Archer
Deborah A. Frincke
Karl Levitt
Division of Computer Science
University of California, Davis

September 15, 1989
Revised April 10, 1990

Abstract

Much work has been devoted to methods for reasoning about the specifications of operating system specifications, the goal being to develop specifications for an operating system that are verified to be secure. Before the verification should be attempted, the specifications should be tested. This paper presents tools that can assist in the security testing of specifications. The first tool is based on the Final Algebra Specification and Execution (FASE) system, and would be used to test specifications with real input values. FASE is an executable specification language which is operational in style, in which entities are represented in terms of their observable behavior. To facilitate the testing of an operating system (and its specification), using FASE we have specified a Secure Resource Manager (SRM), a generic template of an operating system. The SRM specification can be specialized to a specification of a particular operating system; the SRM is quite general and handles most features of modern nondistributed operating systems. The second tool, called the PLANNER, is used to derive a sequence of operations that exhibits a security flaw, most often a covert channel for information flow. The PLANNER is based on classical methods of AI planning, specialized to achieve goals concerned with information flow. The tools are demonstrated with respect to a simple operating system specification developed by Millen.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

ps

1 Introduction

In response to the National Computer Security Center's demand for A1 certified systems, there is increasing interest in verification systems that can reason about specifications. An operating system is A1 certified if its specifications are verified to be correct with respect to a policy that limits (1) the objects a process can access based on the security level of the process and object, and (2) the flow of information between processes based on their security levels; as discussed by Goguen and Meseguer [GM82a][GM84] Feiertag, et al. [FLR77], and Rushby [Rus84], (2) subsumes (1). A number of verification systems have been developed with this need in mind, several specialized to carry out security flow analysis.

Clearly, program verification is difficult. However, verification of a specification with respect to the security flow policy need not be as difficult as verifying a large program such as an operating system with respect to a specification that fully describes what the operating system does. The security policy is expressed as follows: Given a sequence of operations invoked by processes at different security levels, an operation at a given level $L1$ can interfere with the results of an operation at level $L2$ only if $L1 \leq L2$. It is difficult to verify a system with respect to this policy, as it entails a proof that considers every sequence of operations. It can be shown that the verification effort can be significantly simplified to just showing that the individual specifications of operations satisfy a policy. However, this method generally requires that the objects have constant security levels [FLR77]. This assumption is clearly unacceptable if the specifications are at a level of abstraction where objects are shared among users of different security levels – clearly the case for most operating systems. There are various tricks for converting a specification that allows for mutable security levels to one in which the security levels are constant, but at the cost of significantly increasing the cost of certification of the implementation – by verification, testing, etc.

Furthermore, it does not make sense for verification to be the only supported certification method. Most specifications will contain errors that are better detected by conventional testing. Furthermore, although not mandated for A1 certification, verification of a system's specifications does not guarantee system correctness; errors can certainly occur in an implementation. Although verification of the implementation of complex operating systems is beyond the stamina of human and mechanical verifiers, verification of abstractions below the top-level operating system interface is feasible. However, at the more detailed levels the assumptions underlying the mechanical flow analyzers are no longer valid.

Accordingly, this paper presents two implemented tools that can assist in the testing of oper-

ating systems with respect to security policies; the tools can be used, in principle, at any level of abstraction.

The first tool is based on the Final Algebra Specification and Execution system (FASE) [KJA83], and would be used to test specifications with real input values. FASE uses an executable specification language which is operational in style, elements being represented in terms of their observable behavior. The style allows for specifications that are highly abstract but to a great extent executable. FASE is implemented in Lisp and allows the intermixing of Lisp code and FASE specifications.

To facilitate the testing of an operating system (and its specification), using FASE we have specified a Secure Resource Manager (SRM), a generic template of an operating system. The SRM specification can be specialized to a specification of a particular operating system; the SRM is quite general and handles most features of modern operating systems.

The second tool, called the PLANNER, is used to derive a sequence of operations that exhibits a security flaw, most often a covert channel for information flow. The PLANNER is based on classical methods of AI planning, specialized to achieve goals concerned with information flow. The PLANNER attempts to achieve a goal wherein one of a process' visible registers acquires a value different from its value in an initial state, the change being caused by a process at a level not less than or equal to that of the original process. The PLANNER uses a backwards-based search strategy. To make the search informed, the planner has heuristics that identify the operations that are most likely to be useful in achieving a goal. The planner can also be used to determine a measure of the channel capacity associated with a particular sequence of insecure operations through determining the number of possible values assignable to unbound variables in the plan - in essence, the uncertainty of the information flow. The PLANNER is implemented in Prolog.

Although these two tools can be used independently, they can be used together in two ways. For example, the FASE tool can be used to determine the initial and final states associated with a sequence of operations suspected to be a subsequence of an insecure sequence; then the PLANNER can determine a prefix and suffix to this sequence that achieves the unwanted flow. In addition, the FASE tool can be used to test partial plans produced by the PLANNER, the plans being extended by the user if the PLANNER becomes involved in long and fruitless searches.

The tools are demonstrated with respect to a simple operating system specification developed by Millen [Mil79]. This operating system provides user operations to read and write memory, and system operations to swap processes, and purge and claim memory blocks, the memory blocks

accessible only indirectly through segment registers. Not surprisingly, the system contains a covert channel, the heart of which is the flow through the purging of a block at one security level and claiming of the same block at a different level.

The organization of the paper is as follows. Section 2 presents a brief overview of FASE, and a description of the generic Secure Resource Manager specification and its instantiation to the Millen operating system. Section 3 presents the planner, commencing with an overview of how it derives plans, followed by two examples exhibiting how it derives sequences of insecure operations for the Millen operating system. Section 4 discusses the tools and further development in progress.

Appendix A presents the specifications of the SRM. Appendix B gives those corresponding signatures needed to understand the example “user expressions”. Appendix C illustrates the form of a PLANNER specification. Appendix D presents the specifications of the Millen operating system in his predicate calculus-based specification language.

2 A generic specification for a secure resource manager

To provide a general tool for the rapid prototyping of secure systems, we have developed a general specification for a secure resource manager. To illustrate its usefulness, we will show in detail how it specializes to a specification of the Millen operating system, and describe how the same process could be applied to obtain specifications of arbitrary complexity.

The specification method we use is that of final algebra specifications, as in [Kam83], [KJA83]. It is less familiar than those usually encountered, which include, for example, algebraic specifications, and specifications given by defining pre- and post-conditions on operations. Therefore, before giving details of the general specification and its specializations, we will review the method itself.

2.1 Writing final algebra specifications: the methodology

Despite its name, the final algebra specification method is not what is generally considered to be an algebraic method; rather, it is operational. It represents elements, essentially, by their observable behavior. Thus, a set S of objects of sort Elt is represented by a map from the carrier of Elt to Bool ; this map determines for each e of sort Elt whether it is a member of S . If the operation for observing membership is named isin , then the map corresponding to S is the same as the map on elements e of Elt obtained by holding S constant in the expression $\text{isin}(S, e)$.

In general, an element of an abstractly specified sort s will be represented by a tuple of maps (some of which may be 0-ary, and hence “maps” only by courtesy). Each element of the tuple will correspond to an operation on elements of sort s in analogy to the way a map from Elt to Bool corresponds to isin : it is obtained by holding the element of sort s constant while letting the other arguments of the operation vary. The operations corresponding to tuple components of elements of sort s form the *distinguishing set* for the sort s . Thus, the distinguishing set of the sort SetofElt consists of the single operation isin .

To give a final algebra specification of an abstract data type, one must first determine the operations in its distinguishing set. Any choice of distinguishing set operators leads to a representation of elements of the specified sort as elements of a product space; this representation then permits one to define the value returned by an operation as a tuple whenever that value is of the specified sort.

There is no fixed recipe for finding the distinguishing set, but there are some heuristics. In some cases, what certain components of this product space must be is clear. For example, it was clear to us in writing our generic specification that any SRM (secure resource manager) must have, among other things, a State component, a Scheduler component, and a SecPol (security policy) component. When this happens, it is clear that there will be corresponding distinguishing set operations that yield these components. In the case of an SRM, we have chosen to call them stateofSRM , schedofSRM , and polofSRM . In other cases, one thinks in the other direction, and starts from the operations to get the product space representation. A typical example of this is an abstract sort StackofElt , in which one realizes that two stacks behave the same if one gets the same things by reading their **top** and by **popping** them. This leads to a (recursive) representation of StackofElt as a product $\text{Elt} \times \text{StackofElt}$.

As we have indicated, once one has determined the tuple representation of elements of the sort being specified, one can then define certain of the operations that return values of this sort — the constructor operations — by giving their results as tuples. All other operations must be defined in terms of these constructors, the distinguishing set operations, and the visible operations from other data type specifications, without the use of tuples. A corollary of this requirement is that tuples can be used to express elements of any sort only inside the specification of that sort.

In the next section, we will first describe the generic SRM specification in terms of the product space representations of certain fixed sorts, and the sorts and operations that will always be present. We will then describe in some detail how it specializes to describe the Millen system, and indicate how it could be specialized to describe other systems of almost arbitrary complexity.

We finish this section with a brief description of the notation and conventions used in our specifications; these are as in the FASE system as described in [KJA83]

Each specification begins with the name of the sort being specified, followed by a declaration of operations and arities; this is called the signature section. Any operations in the AUXILIARY OPERATIONS section are available only inside the given specification; all other operations are visible operations, and may be used by programs (including other specifications). Following the signature section is the list of those declared operations that constitute the distinguishing set.

Distinguishing set operations do not require explicit definitions. To compute the application of such an operation to a list of arguments, one extracts the (unique) argument of the specified sort, and applies the corresponding function in its tuple to the remaining arguments (a trivial application if the tuple function is 0-ary). All other operations, however, require definitions. Tuples in these definitions are indicated by square brackets, their elements separated by commas. 0-ary functions inside tuples are simply given as expressions. For functions with arguments, the notation “<a, b> |-> ...” can be read as “ $\lambda a, b. \dots$ ”.

With regard to error values, there is an error value of each sort, together with a hidden element of the distinguishing set that detects it: thus, there is an `errSRM` and an `iserrSRM`, etc.. Function definitions are strict with respect to errors. Occasionally, the specifications will include definitions of the form `iserrType(opType(args)) => ...`; these are translated as a prefix to the definition of `opType` which, when satisfied, causes `opType` to return an error.

The following notation is used for Boolean operations: “&” is used for `and`, “|” for `or`, “~” for `not`, and “=” for `eqType` for an appropriate choice of `Type`; these have the usual precedences. Note that in the case of `=`, except for a primitive sort `Type`, it is not required that `eqType` be a true equality relation, but only that it be explicitly defined and have the appropriate arity (except that the system forces `eqType` to correctly detect equality to `errType`).

2.2 Structure of the generic SRM specification

For us, the term “secure resource manager” (abbreviated SRM) includes practically any operating system. In our attempt to construct a generic specification for an SRM, we wished to create a template which could be elaborated into specifications of many particular systems. We at first envisioned that such a template would consist of certain fixed, unchanging specifications, with subsorts being allowed variable specifications whose details would depend on a given application. Instead, it became clear to us that the specified sorts in the template would fall into three categories.

We will call specifications in the first category “fixed.” These are essentially what we originally envisioned as fixed, but to make specialization to an application more convenient, we have loosened the requirement to permit addition of operations derived from other operations in the (combined) specification. For instance, our Millen example has an added derived operation `updateobjState` in the specification of `State`. In such specifications, the *reachable carrier* of the specified sort is fixed relative to those of the sorts in its representation. Hence, any properties proved from the general specification about all reachable elements of the specified sort, say by structural induction, will hold in any application.

The second category contains what we will call “fixed representation” specifications. These will specify a sort of fixed name, and will have a fixed list of distinguishing set operations of fixed arity. As a result, the *representation* of the specified sort will be fixed relative to the sorts involved in its representation. In general, such abstract types will have a fixed set of constants whose definitions are independent of the application, and sometimes, other operations with fixed definitions. Any properties that can be proved without structural induction about the elements of such a sort based on provable properties of the sorts in its representation will still hold in any application. Two sorts in our template which have fixed representation specifications are `SRMop` and `SecPol`. `SRMop` has a fixed null operation `NOop`, and a fixed “equality” operation `eqSRMop` (which is not a true equality, but behaves like one in this and any application in which no two operations can have the same name). `SecPol` has default policies `noPol` and `recalcitrantPol`.

The third category of specifications contains those having a fixed subsignature. In fixed subsignature specifications, at the very least, there must be a sort of the same name as the specified sort. In general, certain operations of a certain arity also must be present. Other than arity, there is no restriction on how the operations are defined. Such specifications correspond to the parameter part of parameterized specifications which occur in other specification methods (e.g., that of OBJ ([GM82b], [GM84])). Two of the sorts with such specifications in our template are `Object` and `Request`. `Object` has a fixed operation `eqObject`, which has fixed arity. `Request` has the fixed operations `opofRequest` and `argsofRequest`, also with fixed arities.

In any application, there will be subsidiary data type specifications, which we simply refer to as application specific. The sorts specified need not appear in all applications. Frequently, they are required in order to define sorts needed in the representation of sorts of fixed subsignature, or, possibly, of other application specific sorts. Among the application specific sorts present in our Millen example is `Content`; it exists only to define part of the representation of sort `Object`.

Instead of giving the SRM template specification in isolation, we will show its completion for

the Millen example, and indicate which parts belong to the template. Table 1 gives a complete list of the sort specifications in our Millen specification, together with an indication of the kind of the specification (1 = fixed, 2 = fixed representation, 3 = fixed subsignature, 4 = application specific) and the abstract representation of each sort. Note that the alternate representations mentioned in the table as being the “real” representations of certain sorts show the structure of the reachable part of the sorts.

As we have indicated, the general template consists of parts of those specifications of kinds 1, 2, and 3. In table 2, we indicate for each of these specifications those operations in the Millen example that are present in the template. The *degree* to which they are “present” is determined as described earlier by the specification kind. For kind 2 specifications, we don’t mention the distinguishing set operations, since they must always be present in the template.

As a further aid to understanding the template specification, figure 2.2 gives a graphic representation of the structures of sorts in the template having kind 1 and kind 2 specifications. The sort at each node in the tree is represented abstractly by the sorts of its children.

As indicated in table 1, the representations of ObjectSet and ProcessSet are different in the Millen specification from those in the generic specification. In general, for the purposes of rapid prototyping, it is sometimes convenient to replace a specification by an implementation, for a number of reasons. In our Millen example, we have actually given *implementations* of ObjectSet and ProcessSet, rather than their ideal specifications (in which sets would be represented as maps to Bool). These ideal specifications can, in fact, be given in the FASE language, but would require the use of quantifiers (the operations `findinObjectSet` and `findinProcessSet` would use the quantifier “some” — the existential operator that returns an instance). While many quantified expressions can, in fact, be executed in FASE [JK86], this execution tends to be slow. Moreover, for printing all members of an element of a “set” data type, as one frequently wants to do in an interactive driver for a prototype, it is much more convenient to represent the set as a list. Thus, we have chosen to do so in the Millen specification. Of course, there is, strictly speaking, a proof obligation attached — one must verify that the implementation is correct. In this case, the implementation is actually *partial* — one cannot create an arbitrary set of Objects, for example, but only one in which no two Objects have the same ObjectId. However, one can show that no SRMop causes one to attempt to create a disallowed ObjectSet, provided one starts with a State having a permissible one. Thus, in the context in which the Millen ObjectSet specification will be used, it can be considered an implementation of the generic ObjectSet ([KA], [Arc88]).

As a matter of fact, the notion of partial implementation crops up frequently in the context

Figure 1: Overview of the SRM template specification

of specializing the generic template to a specific example. One place, as we have indicated earlier, is the specification of “equality” operators that are not true equality operators, but are sufficient in context. Another place is in the use of a History component in the generic SRM. It is there to facilitate detection of information flow in cases where this cannot be observed simply by looking at the rest of the State. Provided that the History is never used to affect operations (i.e., in the definition of any SRMop), but only used to observe the system, there is no reason to implement it in implementing a system whose specification one has proved secure.

SORT	KIND	REPRESENTATION
SRM	1	$\text{SRMopSet} \times \text{State} \times \text{Scheduler} \times \text{Interp} \times \text{SecPol}$
State	1	$\text{ObjectSet} \times \text{ProcessSet} \times \text{RequestList} \times \text{History}$
SRMopSet	1	$\text{SRMop} \rightarrow \text{Bool}$
SRMop	2	$\text{Symbol} \times (\text{State} \times \text{ArgList} \rightarrow \text{State})$
Scheduler	2	$(\text{State} \rightarrow \text{Request}) \times (\text{State} \rightarrow \text{RequestList})$
Interp	2	$\text{Request} \rightarrow \text{Request}$
SecPol	2	$(\text{State} \times \text{Request} \rightarrow \text{Bool}) \times (\text{State} \times \text{Request} \rightarrow \text{Request})$
ObjectSet	1	$\text{Object} \times \text{ObjectSet}$ [in generic template, $\text{Object} \rightarrow \text{Bool}$]
ObjectPred	2	$\text{Object} \rightarrow \text{Bool}$
ObjectId	3	$\text{Symbol} \times \text{Symbol} \times \text{Int} \times \text{ProcessId}$ [really $\{\text{'X'}\} + \text{Int} + \text{Int} \times \text{ProcessId}$]
Object	3	$\text{ObjectId} \times \text{Content} \times \text{Level} \times \text{Int} \times \text{Int}$
ProcessSet	1	$\text{Process} \times \text{ProcessSet}$ [in generic template, $\text{Process} \rightarrow \text{Bool}$]
ProcessPred	2	$\text{Process} \rightarrow \text{Bool}$
ProcessId	3	$\text{Symbol} \times \text{Int}$ [really $\{\text{'System'}\} + \text{Int}$]
Process	3	$\text{ProcessId} \times \text{Bool} \times \text{Level} \times [\text{Int} \rightarrow \text{Object}] \times [\text{Int} \rightarrow \text{Object}]$
RequestList	1	$\text{Request} \times \text{RequestList}$
Request	3	$\text{SRMop} \times \text{ArgList} \times \text{ProcessId}$
History	3	$\text{State} \times \text{RequestList}$
ArgList	1	$\text{Arg} \times \text{ArgList}$
Arg	3	$\text{Object} \times \text{Process} \times \text{Int}$ [really $\text{Object} + \text{Process} + \text{Int}$]
Content	4	$\text{Symbol} \times \text{Int} \times (\text{Int} \rightarrow \text{Int})$ [really $\text{Int} + (\text{Int} \rightarrow \text{Int})$]
Level	4	$\text{Bool} \times \text{Symbol}$ [really $\{\text{'syshi'}\} + \text{Symbol}$]
RegAssn	4	$\text{Int} \rightarrow \text{Int}$
ComputeFn	4	$\text{Symbol} \times (\text{RegAssn} \rightarrow \text{Int})$

Table 1: Millen specification overview

SORT	KIND	GENERIC OPERATIONS
SRM	1	(all)
SRMopSet	1	(all but mitreSRMopSet)
SRMop	2	eqSRMop , NOop
State	1	(all but upateobjState)
Scheduler	2	null- and trivScheduler
Interp	2	trivInterp
SecPol	2	no- and recalcitrantPol
ObjectSet	1	(all)
ObjectPred	2	trivObjectPred
ObjectId	3	eqObjectId and precedesObjectId
Object	3	idofObject
ProcessSet	1	(all)
ProcessPred	2	trivProcessPred
ProcessId	3	eqProcessId and precedesProcessId
Process	3	idofProcess
RequestList	1	(all)
Request	3	opof-, argsof-, procidof-, and nullRequest
History	3	init- and appendHistory
ArgList	1	(all)
Arg	3	objidof-, procidof- , objidto- , and procidtoArg

All distinguishing set operations of kind 2 and kind 1 specifications are also generic.
For operations in kind 3 specifications, only the arity is generic.

Table 1: Generic template

2.3 Specializing the template SRM specification

We believe that the usefulness of the template SRM specification lies in how it organizes one's thinking about the workings of an (almost) arbitrary secure system by factoring the system into its components that play conceptually different roles, and also in relieving one of having to specify certain high-level operations. Some of these components (e.g., the Scheduler) will play more important roles in some systems than others, but for any relatively complex system, most of them will be non-trivial. The template has one further advantage: when it is completed (and in many cases, only partially completed), it can be used as a rapid prototype of a system (or part it).

In this section, we first give an example in which we have used the template to analyze and specify a particular system, namely the one described in [Mil79]. We then describe a general approach to specializing the specification that can be applied to nearly any system.

2.3.1 Obtaining the Millen specification

As a concrete illustration of how the template SRM specification can be useful, we will describe here the major decisions we needed to make in specializing it to a specification of the Millen system. In the process of specialization, a number of omissions in the description of the Millen system in [Mil79] became clear, and we had to make some decisions about how to handle them. We tried to make these decisions so as to allow as much flexibility as possible in the system objects one might wish to create for testing purposes.

In specializing the template, one first has to make decisions about how to complete the kind 2 specifications. We will take these in order.

The first such specification we encounter is SRMop, which determines what can be in an SRMopSet. Here, we soon discovered that the list of operations in [Mil79] needed to be expanded. For example, a user, not being privileged, cannot directly execute certain commands such as Swap and Purge (SWAP and PURGE in [Mil79]). The user is also unable to do a Get (GET, in [Mil79]), not having access to information about free memory blocks. Thus we have added user operations uSleep, uRelease, and uGet which, in effect, ask the system to request the corresponding privileged operations. We similarly decided that the privileged operation Raise will normally be invoked as the indirect result of certain user calls to uGet, and that the privileged operations ReadX and WriteX (READ and WRITE the X register in [Mil79] — created there to illustrate a possible way to introduce a security flaw, which we wanted to be able to exhibit) similarly required

corresponding user calls to `uReadX` and `uWriteX`, respectively. There are also problems with the nominally purely user operations `uFetch`, `uStore`, and `uCompute` (`FETCH`, `STORE`, and `COMPUTEk` in [Mil79]) since a process would have to pass itself (or its name) as an argument so that the correct memory block and registers to reference can be determined. Thus we also added corresponding system (although not privileged) operations `Fetch`, `Store`, and `Compute`.

Our template specification is structured to permit the definitions of operations to be factored into what the operations do, and the pre-conditions under which the operations will be allowed. In our Millen example, we isolate the pre-conditions into the `SecPol` factor of the SRM, particularly since their enforcement is the system's attempt to enforce the "true" security policy. Thus, our operation definitions in the SRMop specification are free of the clutter of the pre-conditions.

The actual definitions of the operations force a number of implementation decisions concerning what will be the elements of `Arg` and what some of the operations on Objects and Processes must be. E.g., there must be `Int` and `ObjectId` arguments, and there must be an operation by which to observe the Content of an Object.

The next kind 2 specifications we encounter are `ObjectPred` and `ProcessPred`. We see that we cannot add constructors to these data types without knowing more about the structure of Objects and Processes. Since the Millen specification refers to Processes and memory blocks (which will be one kind of Object) by number, and registers (another kind of Object) by name or number, we decide that both Processes and Objects must have Ids, and that there must be predicates to decide when a given Object or Process has a given Id. Thus we are also led to create additional data types called `ObjectId` and `ProcessId`, having equality operators. Having done this, we can now define the desired predicates that test equality of an Id to a given one.

We next encounter the components of the State. The only component not of kind 1 is the History (kind 3). Though we are mainly discussing only kind 2 specifications, it is worth looking more closely at History in order to understand the role it plays. In general, a History object could be less detailed than we make it; since it is there for flow-detection purposes, it is enough to record whatever information is needed for this purpose. We have chosen to keep essentially all the information needed to reconstruct all changes of system State. Thus, a History is a State and a RequestList; `initHistory` saves the initial State, and `appendHistory` is used to successively save requests.

There being no explicit mention of scheduling in [Mil79], we have taken the Scheduler to be `trivScheduler`.

In general, we see the Interp component of an SRM — the Request interpreter — to be a mechanism by which the system can trap user requests and interpret them in terms of commands only known to the system. This seems to us to be the natural way to allow for operations such as TRAP [Mil79]. The specific element of sort Interp that needs to be included in any Millen system is **mitreInterp**, which changes user Requests into lists of Requests involving “internal” operations; thus, a user **uGet** request becomes one of the sequences **<Get>** or **<Raise, Get>** of system requests.

In any SRM, the **SecPol** component will be the “effective security policy” — actually, the filter on Requests that attempts to enforce the true security policy — of the system. In operation, it tests a Request for whether the pre-conditions associated with its **SRMop** are satisfied. In the Millen example, these pre-conditions actually simply implement an access policy; in general, they could make much more complex tests that might even examine the History component of the State. However, the way we have structured **mitreSecPol** is typical of the way we would expect the **SecPol** of any SRM to be structured: separate **SecPols** for the operations are combined into the overall **SecPol**. The tests made by the **SecPol** on operations force some further decisions. For example, the fact that the **SecPol** wants to know the Level of the Process requesting certain operations means that such information must be obtainable from a Request. We have chosen to give each Request a **ProcessId** component, although it would be sufficient to just use the Level of the corresponding Process. Other decisions forced by the definition of **SecPol** are that each Object must have a Level, and **Int** representing its access count, and so on.

2.3.2 Obtaining a specification of your pet system

The procedure we used above for the Millen example can be followed to yield a specification of almost any system. Basically, one completes the kind 2 specifications; in the process, one finds out the structure of the sorts having kind 3 specifications, and the operations needed in these specifications. In the process of elaborating the kind 2 specifications, one creates the needed kind 4 specifications.

More particularly, one starts by determining the **SRMops** — the operations provided in the system. Any restrictions on the permissibility of these operations can be factored out and made part of the **SecPol**. Which constructors to add to **ObjectPred** and **ProcessPred** depend mainly on the needs of the **SRMops**, since these operations are the ones that ultimately select Objects or Processes in order to change (or, in our applicative system, replace) them. There may be cases, however, in which **SecPol**, for example, needs particular **ObjectPreds** or **ProcessPreds**. The basic

properties of an Object or Process can usually be determined from the high-level description of the system. Further details that distinguish individual Objects or Processes are determined by what various SRMops do, what the SecPol needs to test for, and so on. The needs of the Scheduler and the SecPol, primarily, determine what, besides an SRMop and an ArgList, distinguishes one Request from another. In the Millen example, a Request had an associated ProcessId; in systems with a nontrivial Scheduler, Requests may need to have an associated priority instead, or as well. Whether there will be a nontrivial History component in the State depends on what information flows one wants to test for, and how much information about the previous states of the system one needs in order to do so.

2.4 Running the specification: the Millen example

As mentioned in the previous section, we attempted to make our Millen specification as flexible as possible; the reason for this is that one can then initialize the system to any permissible configuration desired: any number of user processes (numbered starting from 0), any number of registers for any given user process, any number of memory blocks, etc.. Having done this, we *assume* that the system will be initialized in a permissible configuration; we also assume that the driver accepts requests only if they could occur in the current state of the system. These conditions could be enforced inside the specification, but it would be cumbersome to do so, and we have chosen not to.

In the case of the Millen example, the SRMopSet of the initial configuration must be `mitreSRMopSet`. Also, the State of an initial configuration will have only one swapped-in Process, and no memory block Objects that are referenced by more than one Process; moreover, its ProcessSet must consist of the `systemProcess` and zero or more user Processes with ProcessIds numbered consecutively from 0. Finally, the Scheduler of any initial configuration must be `trivScheduler`, its Interp must be `mitreInterp`, and its SecPol must be `mitreSecPol`.

Once an initial configuration is assigned to a Lisp variable, updates to this configuration must be done by `stepSRM`, which will never violate the conventions of the system, and `newreqSRM`. The driver should accept a new Request only if it comes from a Process of which `inoffProcess` is true (meaning the Process is swapped in).

Thus, to complete the rapid prototype of the Millen system one should write a driver that 1) creates a permissible initial configuration of the system upon reading input that controls the variables in such a configuration (such as how many Processes there are, what memory block Objects of what sizes there are, etc.), 2) takes as input instructions either to call `stepSRM` or to

call `newreqSRM` with a new Request, and 3) checks that incoming Requests could in fact arise at the point they are given as input. As we will explain in the next section, the FASE system provides a means to make the input instructions convenient to give interactively without the use of the long function names appearing in the specification, and makes it convenient to write a driver for the system in Lisp.

Although we have not yet written a driver for the Millen system, we have written drivers for a number of other FASE specifications, and anticipate no difficulties in doing so.

2.5 Advantages of the FASE system

The FASE system [KJA83] is a system for executing final algebra specifications [Kam83] of abstract data types. We find its use convenient for several reasons. Perhaps the most important reason is that the specifications are executable whenever they do not involve quantifiers. Even some specifications with quantifiers are executable, and the system is able to identify a class of such specifications which it guarantees it can execute (although, for efficiency reasons, it is best to avoid quantification in a rapid prototype).

There are, of course, other systems for executing specifications, such as OBJ and Affirm [GHM78]. These mostly involve algebraic (equational) specifications. While such specifications are sometimes straightforward to write, knowing when one has enough equations (or so many as to lead to inconsistency) can be a problem. By contrast, we find final algebra specifications to be among the easiest specifications to write. One does not have the problem of proving sufficient completeness and consistency of equations. Once one has determined the abstract representations as tuples of elements of various sorts, the definitions of their associated operations are usually straightforward to derive from verbal descriptions.

In addition, the FASE system is integrated with Lisp, so that one can easily write Lisp drivers to exercise, and hence test, one's specifications interactively. One can easily incorporate into a driver print routines that call operations from the specification in order to display, for example, a system state represented in some convenient fashion by its observable behavior.

Specification and testing are further much facilitated by a provision for almost arbitrary user-defined syntax. With an appropriate grammar, one can easily handle the formation of sets and sequences, as well as coercions, and avoid typing in lengthy (though possibly, as in our example, descriptive) function names. User expressions are delimited in Lisp by exclamation points, and in specifications by underscores. Thus, in our Millen example, we can provide a grammar that will

allow us to say at the lisp level:

```
!user 0 calls uReadX on 2, 5 in M!
```

to abbreviate

```
(newreqSRM (mkRequest (uReadX)
                      (appendArgList (inttoArg 5)
                                      (appendArgList (inttoArg 2) nilArgList))
                      (mkuserProcessId 0))
M)
```

(note that M in both the above can be a Lisp variable whose value is of sort SRM). While overloading of operators in the specifications per se is not allowed, it is feasible to a great extent in the grammars, since the parser is able to do type-checking of all but variables.

3 The PLANNER

This section discusses an approach to formal testing of an operating system with respect to a security policy. The goal of the work is as follows: Given specifications for the operations of an operating system, determine a sequence of operations that effect a disallowed information flow between a pair of processes, assuming that such a flow exists.

This concept is implemented in a prototype system called the PLANNER; it is implemented in Prolog, but uses its own searching code to make the search more informed than the basic Prolog resolution method. It is based on an extension of classical AI planning methods [Nil80].

This section discusses the general approach to planning as it bears on finding sequence of operations that exhibit disallowed flows, followed by a detailed description of the PLANNER. It has been used to detect flows in a simple operating system specification [Mil79].

3.1 Detecting flows

Planning (of actions) is concerned with the automatic derivation of a sequence of operations (called the plan) that achieve a goal. In classical planning, the goal is expressed in terms of two states: an

initial state and a final state, and the sequence of operations is to achieve the final state given the initial state. Planning has been used to derive plans for moving robot vehicles so as to achieve a goal such as "refueling a tank."

The basic approach to planning involves searching combined with unification. The searching can be forward driven, backward driven, or mixed. The backward approach is as follows. The goal and initial states are described in terms of predicates on constants and variables. The operations are specified in terms of preconditions and postconditions, each expressed in terms of variables and constants. Variables can be bound during the planning process.

If the goal state unifies with the initial state, the plan is complete. The collection of operations is scanned to locate an operation with a postcondition that unifies with the goal state. This operation might have preconditions; for simplicity, assume there is only a single precondition. The variables of the precondition might be bound to values as a result of the unification of the goal state with the postconditions. Then the postcondition, with the variables appropriately bound, becomes the new goal.

Complications arise when the goal is a conjunction of terms; a conjunction of terms can arise at any point in the planning process; for example, when an operation is used that has more than one precondition. Significant research has been devoted to this problem, none of the solutions avoiding possible long and fruitless searches. One technique is as follows. When encountering a conjunction of goals, pick one to work on; it might be possible to identify the most difficult of the goals assuming a measure of difficulty is available. Identify an operation with a postcondition that unifies with this selected goal. Then the other goals are regressed backwards through this operation. If the regressed goals conflict with the preconditions, then the choice of operation or the goal selected to pursue was flawed, and the procedure backtracks to select a different operation or a different goal. The planning process continues until the collection of goals unifies with the initial state.

Our security-based PLANNER is similar, but introduces some new concepts to cope with information flow.

The goal state expresses an insecure flow, and is based on the concept of "interference" of processes in an operating system. The concept of flow and of a model "process" is wired-into the PLANNER to reduce the search space. A more general planner would take an arbitrary security policy and model of processes, but at the expense of search time.

Assume a goal state in which process P_i is running and has read access to a collection of registers; it is through these registers that a process acquires information. Assume an initial state

in which P_i is also running. The contents of a register in the initial state and final state are to differ. In order to exhibit information flow, the difference is to be caused by an operation by a process P_j whose security level is not less than or equal to that of P_i ; for simplicity the PLANNER assumes a category model of security levels: processes at different levels are forbidden to communicate with each other. Still working backwards, the PLANNER will have two types of goals to achieve:

1. Goals that involve information flow between processes: These goals are achieved by two operations, one that produces a change in state (e.g., to a register) and the other that causes a swap of processes. Assuming no single operation satisfies both of these needs, the final plan must contain these two operations but not necessarily contiguously.
2. "Ordinary" goals as faced by the classical planner. These goals will arise from preconditions of operations that the planner conjectures will become part of the plan.

These concepts are discussed in the next section with reference to the PLANNER deriving a plan for insecure flow in the Millen operating system.

3.2 Detailed description

The current implementation of the PLANNER consists of a collection of rules written in Prolog. Prolog was selected in order to take advantage of backtracking and resolution. Use of backtracking provides a method whereby all possible plans may be considered, and use of resolution permits planning variables to be left unbound as long as possible. The PLANNER contains four types of rules: *planning rules*, *difference computation rules*, *operation rules*, and *architecture rules*.

Planning rules: Planning rules are used to manipulate plans. There are two types: those that eliminate "unnecessary" goals, and those that select the subgoal which will be achieved next. Unnecessary goals are goals that have already been achieved (as a side effect of solving other goals), or goals that do not cause any real change in state. For example, the system may have two goals: causing user A's register i (denoted $R(A, i)$) to contain value X (Goal 1), and causing user A to become the currently active user (Goal 2). Suppose that in the initial state, user B is active and user A is blocked, and that only active processes may modify their registers. Further suppose that the system chooses to work on Goal 1 first, and achieves it via the following plan:

1. Make user A active so that X may be written into $R(A, i)$

2. Write X into $R(A, i)$

Clearly, Step 1 of the plan also achieves Goal 2, so that goal may be eliminated from the system. Step 1 and Step 2 may be considered *subgoals* of Goal 1.

The second type of planning rules consists of those rules that determine which goal the PLANNER will try to achieve first. In theory, the goals are achievable in any order: if the PLANNER determines that it is impossible to achieve all goals at some point in the computation, then it backtracks and tries them in a different order. However, in practice this does not always work with the current version of the PLANNER, since it does not recognize infinite loops in planning sequences. The PLANNER may attempt to achieve a sequence of goals where the solution to the first goal "undoes" the solution to the last goal. This may be seen clearly in the following example.

Consider a system containing Goal 1 (described earlier) and Goal 2': make user B active. This time, user A is active initially. The following sequence will loop infinitely:

1. Make user B active (to achieve Goal 2').
2. Make user A active (to achieve Subgoal 1 of Goal 1). This, however, undoes Goal 2'.
3. Make user B active (to achieve Goal 2')
- ⋮

Both subgoals could have been achieved if the PLANNER had chosen to complete both of Goal 1's subgoals before attempting to achieve Goal 2'. In order to avoid this type of looping, the current version of the PLANNER uses two heuristics: complete all the subgoals of a goal at one time, and complete the most complicated goals first (these are the goals most likely to undo other goals). The complexity of a rule is measured by counting the number of subgoals it contains. These two heuristics are not sufficient to prevent all infinite loops, so future implementations of the PLANNER are expected to contain some form of loop detection and escape. Possible methods of achieving this are discussed in the conclusion.

Difference rules: The PLANNER operates by determining the differences between an initial state and a final state, and then attempting to eliminate these differences. The PLANNER computes differences by comparing state components in the initial and final states. For example, if process A's register $R(A, i)$ contained the value X in the initial state and the value Y in the final

state, then the PLANNER would add a *register difference* of $[[A, i, X], [A, i, Y]]$ to the current goal list. Every state component has its own collection of difference rules, since these depend upon the representation of that component.

Operation rules: The operation rules embody the semantics of system operations. Each operation the system provides is described in terms of preconditions and postconditions. For example, it was mentioned earlier that a precondition of writing into a given process' register is the requirement that the process be currently active. A postcondition of writing the value X into register R is that register R now contains the value X.

Figure 2 is a high-level description of the *Fetch* operation defined in the Millen system. Operationally, *Fetch* reads a value from a specified memory location and places it into a local register. *Fetch* is called with four arguments: the user doing the fetching, the register into which the new value will be written, and the virtual address of the memory block and physical offset within the memory block of the new value. The *with* clause in the definition describes the system state that will result from applying *Fetch*. The original system state, which is described in the full Millen system definition given in Appendix 5, is *Memory, Register, Mmu, CurrentUid, WaitingUid*; thus, *Fetch* is defined to modify only the *Register* portion of the system state.

The *produces* clause describes the actual difference of the operation: the value of register Rn for User will be changed from its previous value (DontCare) to the new value (New). The value DontCare indicates that the actual value that was originally stored in register Rn does not affect the validity of the *Fetch* operation. The *by* clause states that this modification is produced by the architecture rule *doStoreReg*; this rule is the one that the PLANNER uses to modify its internal version of state. Architecture rules are described in more detail in a later section.

The *with* clause introduces the preconditions that must hold before *Fetch* may actually be applied. In this example, the three preconditions state that (1) there is a physical memory location which contains the value New, (2) User has access to that location (i.e., User has a virtual address corresponding to the physical block address), and (3) User is currently active (i.e., the user id of the currently active user—CurrentUid—is identical to User). If any of these preconditions does not hold at a stage in the plan where the PLANNER wishes to apply *Fetch*, then the PLANNER will make them subgoals and attempt to achieve them first. If these subgoals cannot be achieved, then the PLANNER cannot add *Fetch* to the plan.

Associated with each of the preconditions there is a *trigger* and a *difference* clause. These clauses are used to indicate ways in which the preconditions can be satisfied if they are not currently

```

operation  Fetch(User, Rn, Mvirtual, Mn) with
state  Memory, RegisterOut, Mmu, CurrentUid, WaitingUid
produces Register:
    [[ User, Rn, DontCare], [ User, Rn, New]]
    by doStoreReg( Register, User, Rn, Newval, RegisterOut)
with
    precondition  fetchMemory(Memory, Mphys, Mn, New)
                    trigger Memory
                    difference [[Mphys, Mn, undefined], [Mphys, Mn, Newval]]
                    end ;
    precondition  memMap(Mmu, User, Mvirtual, Mphys)
                    trigger Uid
                    difference [[Uid], [WaitingUid]]
                    end ;
    precondition  equal(CurrentUid, User)
                    trigger Uid
                    difference [[Uid], [WaitingUid]]
                    end

```

Figure 2: High-level description of Fetch: $R(i) \leftarrow Mem(j, k)$

true. The trigger clause specifies the particular component of state that must be modified. For example, if there is a memory location containing the desired information but the current user cannot access it (i.e., *memMap* fails), then the PLANNER triggers a change in current user *Uid*. The difference clauses describes the exact difference that must be achieved in order for the precondition to be satisfied.

Operation rules are used by the system to create steps in the plan. As mentioned earlier, the PLANNER operates by trying to achieve goals which are stated as differences between states. The PLANNER starts from a given final state and works backward to satisfy the given initial state. (Recall that information flow to a user *A* is defined here as information that was not available to *A* in the initial state but is available to *A* in the final state. For the Millen system, information is considered to be available to a user when it is in one of that user's registers.) When the PLANNER is working on a particular goal, it searches through the system operations until it finds one which

Figure 3: Insertion of a subplan

has a postcondition containing the desired difference. In our example, the PLANNER would select *Fetch* to achieve the goal of modifying a register's contents. In order to use an operation, the PLANNER must ensure that the operation preconditions hold at the point in the plan where it is to be used. These preconditions are then subgoals. Note that preconditions do not themselves contain operations, but only architecture rules (described later).

Some complications can arise when an operation has more than one postcondition, since all of the postconditions of an operation must hold at the stage in the plan following the operation's application. If one of the operation's postconditions does not hold, it is necessary to insert a *subplan* between these stages. The initial state of the subplan corresponds to the state that holds after the current operation is applied (i.e., operation postconditions hold), and the goal state is the one to which the operation is to be prepended (see Figure 3).

Architecture rules: The PLANNER's architecture rules are used internally by the PLANNER to modify its view of the system state. They may also be considered predicates on the state that are instantiated or revoked depending upon the operation being applied. For example, in a system

containing a collection of registers, there must be architectural rules that will allow the PLANNER to observe and modify register values within the current state. Alternately, one may consider the system to contain predicates such as "Register I of user A has value X" and "Register I of user A is modified to contain value X." Architectural rules are used within the plan to describe preconditions and postconditions.

3.2.1 High-level System Description

As was alluded to in an earlier section, the PLANNER is written in Prolog. However, a high-level translator has been developed so that it is not necessary for the user to describe entire systems directly using Prolog code. This translator is capable of generating most of the planning rules and operation rules, and some of the difference computation rules. It is still necessary for the user to write Prolog code describing most of the difference computation rules and the architecture rules. Fortunately, there are not many of these and they tend to be easy to formulate. The full high-level description of the Millen system is given in Appendix 5.

3.3 Planner input vs. specification input

There are many similarities between the high-level description of the planner and the generic specification of the system. Both systems contain an internal state and operation descriptions. The preconditions of the PLANNER's operation rules correspond to the security policy described in the generic specification. The postconditions of the PLANNER's operation rules correspond to the functional description of the operations in the generic specification. Further, plans found by the PLANNER are executable by the generic specification (see Figure 4). At present, the PLANNER does not contain all of the information in the full specification: for example, there is no explicit history, interpreter or scheduling policy. The PLANNER does, however, contain history information implicitly in the plan that it produces. An interpreter is not necessary for the PLANNER, since it uses the expanded version of user operations to generate plans. However, it is easy to find examples where the scheduling policy of a system can itself be either a source of flow violations, or may eliminate certain types of flow. The only notion of scheduling available to the PLANNER lies in its ordering of goals, and the preconditions associated with operations.

Figure 4: Planner and SRM specification

3.4 Examples

3.4.1 An obvious example of insecure flow

This section describes the path which the PLANNER follows to come up with a very simple example of information flow. The operations available to the PLANNER include all of those used by the general Millen operating system, plus two that have been added to induce flow: *ReadX*, *WriteX*. The Millen operating system has been expanded to include a special system register $\mathcal{X}$, which any user may read or modify using the new operations. Clearly, flow between users may occur via this special register. To further simplify the example, the system is assumed to contain only two users, each with exclusive access to two blocks of memory. At each step in the plan, the system state and the plan state will be given.

The PLANNER is given the goal of finding a plan whereby one user obtains information originally contained in the second user's memory. This is stated by defining an initial state where User 1's block does not contain User 2's information, and a final state where User 1's block does contain User 2's information. Recall that we are working backwards from the final state toward the initial state.

Initial state:

$$\left(\begin{array}{lll} R(\text{User1}, i) = R_1 & R(\text{User2}, i) = R_2 & \text{Mem}(0, k) = M_0 \\ Mmu(\text{User1}, m) = 0 & Mmu(\text{User2}, j) = 1 & \text{Mem}(1, k) = \text{Secret} \\ \mathcal{X} = \text{DontCare} \\ \text{Current user} = 2 \end{array} \right)$$

Final state:

$$\left(\begin{array}{lll} R(\text{User1}, i) = \text{Secret} & R(\text{User2}, i) = R_2 & \text{Mem}(0, k) = M_0 \\ Mmu(\text{User1}, m) = 0 & Mmu(\text{User2}, j) = 1 & \text{Mem}(1, k) = \text{Secret} \\ \mathcal{X} = \text{DontCare} \\ \text{Current user} = 1 \end{array} \right)$$

1. $\text{Mem}(Mmu(\text{User1}, m), k)_0 \neq \text{Mem}(Mmu(\text{User2}, j), k)_0 \neq 0$
 $\text{Mem}(Mmu(\text{User1}, m), k)_t = \text{Mem}(Mmu(\text{User2}, j), k)_t \neq 0$

To shorten the plan, use User2 as the active process initially and User1 as the active process in the final state. (The PLANNER will produce a correct plan even if this is not the case.)

Possible Plans for Goal 1:

- $Store(i, m, k)$ by User1, with $R(User1, i) = Mem(Mmu(User2, j), k)_0$
- $Purge(Mmu(User1, m))$

There are two possible plans, since two operations contain a postcondition with a modified memory. However, $Purge$ can only write 0 into memory and we have excluded 0 as a possible secret message.

Choose plan $Store(i, m, k)$.

$$\overbrace{\dots Store(i, m, k)}^{User1}$$

This sets up a new goal:

2. $R(User1, i) = Mem(Mmu(User2, j), k)_0$ Possible Plans for Goal 2:

- $Fetch(i, j, k)$ with $t=0$
- $ReadX(i)$ with $\mathcal{X} = Mem(Mmu(User2, j), k)_0$

(Either possibility will produce a valid plan here. If plan $Fetch$ is chosen, the PLANNER will attempt to satisfy the precondition that there must be a register within User1's register set containing the value in User2's block. This produces a longer plan, so for illustration, $ReadX$ will be chosen.)

Choose plan $ReadX(i)$.

$$\overbrace{\dots ReadX(i) Store(i, m, k)}^{User1}$$

This sets up a new goal:

3. $\mathcal{X} = Mem(Mmu(User2, j), k)_0$ Only a $WriteX$ can cause $\mathcal{X}$ to contain the desired value. The problem is that the current process cannot do the write, since it does not have User2's information. Thus, User2 must have done the write into $\mathcal{X}$ earlier. This sets up a subgoal that must be achieved before goal 3 can be achieved.

4. Current process = User2

Possible Plans for Goal 4:

- $Swap$

Only one possible plan, so choose plan *Swap* and propagate the goal 3.

$$\overbrace{\dots \text{Swap}}^{\text{User2}} \overbrace{\text{ReadX}(i) \text{Store}(i, m, k)}^{\text{User1}}$$

There is now a plan for goal 3 which may be applied: Possible Plans for Goal 3:

- *WriteX*

Choose plan *WriteX(i)*. The plan now becomes:

$$\overbrace{\dots \text{WriteX}(i) \text{Swap}}^{\text{User2}} \overbrace{\text{ReadX}(i) \text{Store}(i, m, k)}^{\text{User1}}$$

This sets up a final goal:

5. $R'(User2, i) = Mem(Mmu(j), k)_0$

This may be achieved directly by plan *Fetch(i, j, k)*.

Thus, the resultant plan is:

$$\overbrace{\text{Fetch}(i, j, k) \text{WriteX}(i) \text{Swap}}^{\text{User2}} \overbrace{\text{ReadX}(i) \text{Store}(i, m, k)}^{\text{User1}}$$

3.4.2 A more subtle example of insecure flow

The previous example described an obvious example of flow that was easily discovered by the PLANNER. In that example, information was directly transferred from one user to another. This section describes a more complex example of information flow where information is transferred indirectly. Here, one user will observe one of two possible results, depending upon the actions of a second user. The operations used in creating this plan are those described in Appendix ??, excluding both the *Compute* operations and the two operations that modify register $\mathcal{X}$.

Some new notation will be introduced at this point. Objects and operation names may be subscripted with plan stages: $R_t(U, i)$, $Store_t(i, j, k)$. Plan goals such as $=$, $\neq$ may also be subscripted with a user level to indicate that they may only be satisfied by a user operating at that particular level. For example, $R_t(U, i) \neq_y R_0(U, i)$ indicates that user U 's register i at plan stage t must not

have the same value that it did at plan stage 0; further, the change in value must have been caused by a user operating at level y .

The following describes the reasoning the PLANNER could use to locate such a security flaw. The relevant part of the system state is shown in this format:

$$\left(\begin{array}{lll} Mmu(X, j) = b_1 & Mmu(Y, j) = b_2 & Mem(b_1, k) = M \\ R(X, i) = R_1 & R(Y, i) = R_2 & \end{array} \right)$$

1. $R_t(X, i) \neq_y R_0(X, i)$

The initial goal of the system is to find a plan where user X has a different value in one of its registers if user Y takes a particular action (note that this violates the restrictiveness property). Only user X can modify $R_t(X, i)$. Possible Plans for Goal 1:

- $Fetch(i, j, k)$ by user X

Thus, goal 1 will be achieved when the following goal is achieved:

2. $Mem_{t-1}(Mmu(X, j), i) \neq_y R_0(X, i)$

To fulfill this goal, we must ensure that something other than $R_0(X, i)$ is written into the memory block b_1 , where $b_1 = Mmu_{t-1}(X, j)$. Since this inequality is restricted to be caused by user Y, user Y must have had access to the block at some stage in the plan. The current plan looks like this:

$$\overbrace{\dots Swap \dots Swap \dots Fetch(i, j, k)}^{\begin{array}{ccc} UserX & UserY & UserX \end{array}}$$

User Y cannot modify memory unless it has a pointer to it in its Mmu. Both users cannot point to memory simultaneously, so user X must have regained access to the memory block after Y. This sets up a new goal:

3. $Mmu_{t-2}(X, j) = b_1$

Possible Plans for Goal 3:

- $Get(b_1, j)$

The preconditions to this operation require block b_1 to be active, have no other user accessing it, and have the proper security level, leading to three new goals:

4. b_1 is active

5. b_1 is unused
6. b_1 has level X

Possible Plans for Goal 6:

- *Raise*(b_1)

This plan also achieves goal 4. The goals which still need to be achieved are: 2, 5. Possible Plans for Goal 2:

- *Purge*(b_1)

This portion of the plan should take place before user Y performs *Swap*, since user X no longer has access to b_1 here. The current plan and system state at $t - 4$ are:

$$\overbrace{\dots \text{Swap}}^{\text{User X}} \dots \overbrace{\text{Purge}_{t-4}(b_1)}^{\text{User Y}} \overbrace{\text{Swap} \dots \text{Raise}(j)}^{\text{User X}} \overbrace{\text{Get}(b_1, j) \text{ Fetch}(i, j, k)}^{\text{User X}}$$

$$\left(\begin{array}{l} Mmu(X, j) = b_n \quad Mmu(Y, j) = b_1 \quad Mem(b_1, k) = 0 \\ R(X, i) = R_1 \quad R(Y, i) = R_2 \end{array} \right)$$

Note that goal 2 has been achieved, since b_1 now contains 0 after the *Purge*. It is necessary to achieve the preconditions of *Purge* (same as goal 5). This goal is achieved as a side effect of *Get*, since that is the only operation which makes blocks available to the system. Additionally, since we assumed that user X originally had b_1 , it makes sense to place this plan before user X's first *Swap*. Assuming that User X initialized $R(i)$ to something other than 0, the plan and initial system state become:

$$\overbrace{\text{Initialize} \text{ Get}(b_3, j) \text{ Swap}}^{\text{User X}} \overbrace{\text{Purge}_{t-4}(b_1) \text{ Swap}}^{\text{User Y}} \overbrace{\text{Raise}(j) \text{ Get}(b_1, j) \text{ Fetch}(i, j, k)}^{\text{User X}}$$

$$\left(\begin{array}{l} Mmu(X, j) = b_1 \quad Mmu(Y, j) = b_n \quad Mem(b_1, k) = X \neq 0 \\ R(X, i) = R_1 \quad R(Y, i) = R_2 \end{array} \right)$$

A necessary component of this plan is that the system is in collusion with the users, since a user cannot directly name blocks or force the system to obtain a particular block.

4 Conclusions and future directions

Although verification of secure operating systems specifications is important and, hopefully, feasible, testing is also important and less expensive than verification at discovering flaws. Furthermore, testing can be used to discover properties of a system aside from those related to security. Accordingly, we have developed two testing tools as a first step towards an automated testing methodology for secure systems.

The first tool is based on the FASE executable specification language. FASE has the advantages of an algebraic language (abstraction, parameterization, and error handling) but has the intuitive feel of an operational language. The FASE tool is used to execute specifications with real values. To facilitate the testing of specifications for a secure operating system, we have written a specification for a generic secure resource manager (SRM) in FASE, the specifications being a template for a large class of secure operating systems. Although it remains to be determined that the SRM is a suitable template for many of the operating systems of interest, our initial experience has convinced us that generic specifications are possible for an operating system and for its security policies; at the abstract level, operating systems have much in common. Although the generic feature is lost at lower levels of abstraction, FASE can be used to mix specifications with decisions expressed imperatively.

The second tool, the PLANNER, is used to derive a sequence of operations that exhibits a security flaw, usually a covert channel. It is based on classical planning techniques, but extended to handle goals that express a flow of information between processes. Different from the classical planners, PLANNER is specialized to the domain of security flow. This specialization has led to a system whose performance is better than that achievable with a general planner.

Both the SRM and the PLANNER have been evaluated on small examples. The main example is an operating system described by Millen, whose purpose is to demonstrate security verification. Its main operating system features are process switching including swapping, and reclaiming of memory blocks. Although clearly a toy, it has the basic features of a multiprogrammed operating system with dynamic allocation of main memory.

Clearly, it is essential to consider larger systems. A suitable next step would be a system based on MINIX, an operating system with the system calls of UNIX, but a vastly simpler kernel.

We have several improvements in mind for FASE. Currently, many uses of quantifiers lead to specifications that are not executable. When FASE is unable to execute a quantified expression,

the user has no choice but to try a different specification. We are considering a feature whereby a user can define an implementation for a quantified expression (such as through an iterator over a type). Furthermore, FASE can execute only specifications with respect to real values of input. Extensions to symbolic evaluation would improve its utility in testing, but at the cost of producing complicated expressions to the user.

The SRM is an initial step towards a generic operating system specification. It should be extended to include additional features in support of distributed systems. A good first step would be generic message passing and remote procedure call.

The PLANNER's primitive search strategies work for operating systems that contain relatively few operations. As the number of operations increases, so does the the probability of the PLANNER making poor choices in attempting to achieve goals. Heuristics are needed to help the planner in making choices.

References

- [Arc88] M. Archer. *Partial Implementations of Abstract Data Types: Theory and Practice*. PhD thesis, University of Illinois at Urbana-Champaign, June 1988.
- [FGJM85] K. Futatsugi, J. A. Goguen, J. Jouannaud, and J. Meseguer. Principles of OBJ2. *Conference Record of the 12-th Annual ACM Symposium on Principles of Programming Languages*, 52-66, 1985.
- [FLR77] R. J. Fiertag, K. Levitt, and L. Robinson. Proving multilevel security of a system design. *Proceedings, Sixth ACM Symposium on Operating Systems Principles*, 57-95, 1977.
- [GHM78] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Comm. ACM*, 21(12):1048-1064, December 1978.
- [GM82a] J. A. Goguen and J. Meseguer. Rapid prototyping in the OBJ executable specification language. *ACM SIGSOFT Software Engineering Notes*, 7(5):75-84, December 1982.
- [GM82b] J. A. Goguen and J. Meseguer. Security policies and security models. *Proceedings of the 1982 Symposium on Security and Privacy*, 11-20, April 26-28 1982.
- [GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. *Proceedings of the 1984 Symposium on Security and Privacy*, 75-86, April 29-May 2 1984.
- [JK86] S. Jefferson and S. Kamin. Executable specifications with quantifiers in the FASE system. *13th ACM Conf. on Principles of Programming Languages*, January 1986.
- [KA] S. Kamin and M. Archer. Partial implementations of abstract data types: a dissenting view on errors. *Intl. Symp. on Semantics of Data Types, Sophia-Antipolis, France, 1984; LNCS*, 173:317-336.
- [Kam83] S. Kamin. Final data types and their specification. *Transactions on Programming Languages and Systems*, 5(1):97-123, January 1983.
- [KJA83] S. Kamin, S. Jefferson, and M. Archer. The role of executable specifications: the FASE system. *Proc. IEEE Symposium on Application and Assessment of Automated Tools for Software Development*, 105-114, November 1983.
- [Mil79] J. K. Millen. *Operating System Security Verification*. Technical Report M79-223, The MITRE Corp, Bedford, Mass, 1979.
- [Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[Rus84] J. Rushby. The security model of enhanced HDM. *Proceedings of the 7th Dod/NBS Computer Security Conference*, 120–136, September 1984.

A Description of Millen system operations

This section describes the modified Millen operating system operations used in the generic specification and in the planner. Table 1 contains the commands that only the operating system supervisor may issue (i.e., requires privileged mode). All block addresses used in these commands are the physical block addresses. Table 2 contains the commands that only the users may issue (requires unprivileged mode). All block addresses used in these commands are virtual addresses, which transformed to physical addresses via the memory management unit (MMU).

The Millen system has been modified so that each user has its own MMU and register set R , thus eliminating the need for the private storage areas described in the original paper.

In the operations described below, the following notation is used:

- b Physical block number
- j Virtual block number
- k Offset within block
- R' New value of R (after operation)
- i, m Register index

There are several system variables appearing in the operation tables that have not yet been discussed. These variables are used in the original Millen specification to enforce security constraints. The meaning of each of these is given below:

- C_p Current process id
- $Pal(c)$ Security (access) level of user process c
- $Bal(b)$ Security level of block b
- $Bap(b)$ Activity status (block in use) of block b
- $Bac(b)$ Number of processes with access to block b
- $Bdf(b)$ Indicates whether block b is attached to an IO device

The security system used in the Millen operating system relies upon distinct security levels. These levels are not ordered in the low water mark sense: a user may only gain access to a block at its own level. Blocks can change level via the system command *Purge*, which sets the block security level equal to *syshi* (a level which no user process can reach). Block levels are also modified by system command *Get(b,n)*: this command is issued on behalf of a user process, and causes the block level to match the user security level. In addition, the user's MMU is set to point to the block, which has the side effect of releasing one of the user's current blocks.

Plan	Preconditions	Postconditions
<i>Purge(b)</i>	$Bac(b) = 0$ $\neg Bdf(b)$ mode = privileged	$Bal'(b) = \text{syshi}$ $\neg Bap'(b)$ $\forall k Mem'(b, k) = 0$ mode = unpriv
<i>Raise(b)</i>	$\neg Bap(b)$ mode = privileged	$Bal'(b) = Pal(Cp)$ $Bap'(b)$ mode = unpriv
<i>Get(b, n)</i>	$Bap(b)$ $Bac(b) = 0$ $Bal(b) = Pal(Cp)$ mode = privileged	$Bac'(b) = 1$ $Bac'(MMU(n)) = 0$ $MMU'(n) = b$ mode = unpriv
<i>Swap</i>	mode = privileged	$Cp' = (Cp + 1) \bmod P$

Table 1: Commands issued by the supervisor

Plan	Preconditions	Postconditions
<i>Fetch(i, j, k)</i>	mode = unpriv	$R'(i) = Mem(MMU(j), k)$
<i>Store(i, j, k)</i>	mode = unpriv	$Mem'(MMU(j), k) = R(i)$
<i>Compute_k(m)</i>	mode = unpriv	$R'(m) = f_k(\cup R(m))$

Table 2: Commands issued by users

Plan	Preconditions	Postconditions
<i>ReadX(i)</i>	mode = privileged	$R'(i) = \mathcal{X}$ mode = unpriv
<i>WriteX(i)</i>	mode = privileged	$\mathcal{X}' = R'(i)$ mode = unpriv

Table 3: Commands added to induce a flow violation

B Specification of the Millen example

SORT	PAGE	SORT	PAGE
SRM	36	ProcessPred	52
State	37	ProcessId	53
SRMopSet	39	Process	54
SRMop	40	RequestList	56
Scheduler	42	Request	57
Interp	43	History	58
SecPol	45	ArgList	59
ObjectSet	47	Arg	60
ObjectPred	48	Content	61
ObjectId	49	Level	62
Object	50	RegAssn	63
ProcessSet	51	ComputeFn	64

```
SRM /* kind 1 */
```

```
/* Secure Resource Manager */
```

```
stateofSRM : SRM -> State
```

```
opsofSRM : SRM -> SRMopSet
```

```
schedofSRM : SRM -> Scheduler
```

```
interpofSRM : SRM -> Interp
```

```
polofSRM : SRM -> SecPol
```

```
initSRM : ObjectSet ProcessSet RequestList SRMopSet Scheduler Interp SecPol  
-> SRM
```

```
newreqSRM : Request SRM -> SRM
```

```
stepSRM : SRM -> SRM
```

```
DISTINGUISHING SET stateofSRM opsofSRM schedofSRM interpofSRM polofSRM;
```

```
initSRM (OB, PR, RL, OP, SCH, INT, POL) =>  
[initState(OB,PR,RL), OP, SCH, INT, POL];
```

```
newreqSRM (r, M) =>  
[addreqState(r,stateofSRM(M)),  
opsofSRM(M), schedofSRM(M), interpofSRM(M), polofSRM(M)];
```

```
stepSRM (M) =>  
[stepState(stateofSRM(M),schedofSRM(M),interpofSRM(M),polofSRM(M)),  
opsofSRM(M), schedofSRM(M), interpofSRM(M), polofSRM(M)];
```

```

State /* kind 1 */

/* The operation updateobjState is a derive operation included for
   convenience in this application. */

/* This specification contains the definition of the most central operation
   of the entire specification, stepState. */

trivState : -> State
initState : ObjectSet ProcessSet RequestList -> State
updhistState : Request State -> State
addobjState : Object State -> State
addprocState : Process State -> State
addreqState : Request State -> State
remobjState : Object State -> State
remprocState : Process State -> State
remreqState : Int State -> State
objsofState : State -> ObjectSet
procsofState : State -> ProcessSet
reqsofState : State -> RequestList
histofState : State -> History
stepState : State Scheduler Interp SecPol -> State
updateobjState: Object Object State -> State

DISTINGUISHING SET objsofState procsofState reqsofState histofState;

trivState =>
  [emptyObjectSet, emptyProcessSet, nilRequestList, initHistory(trivState)];

initState (O, P, R) => [O, P, R, initHistory(initState(O,P,R))];

updhistState (r, S) =>
  [objsofState(S), procsofState(S), reqsofState(S),
   appendHistory(r,histofState(S))];

addobjState (o, S) =>
  [addObjectSet(o,objsofState(S)),
   procsofState(S), reqsofState(S), histofState(S)];

addprocState (p, S) =>
  [objsofState(S), addProcessSet(p,procsofState(S)),
   reqsofState(S), histofState(S)];

addreqState (r, S) =>
  [objsofState(S), procsofState(S), appendRequestList(r,reqsofState(S)),
   histofState(S)];

remobjState (o, S) =>
  [remObjectSet(o,objsofState(S)),
   procsofState(S), reqsofState(S), histofState(S)];

remprocState (p, S) =>

```

```

[objsofState(S), remProcessSet(p,procsofState(S)),
 reqsofState(S), histofState(S)];

remreqState (n, S) =>
[objsofState(S), procsofState(S), remRequestList(n,reqsofState(S)),
 histofState(S)];

stepState (S, SCH, I, P) =>
let R be getreqScheduler(SCH,S) in
let SS be
    updhistState(R,
        [objsofState(S), procsofState(S), getreqlisScheduler(SCH,S),
         histofState(S)]) in
let RR be getreqSecPol(SS,R,P) in
let RRI be getreqInterp(SS,RR,I) in
let f be opofRequest(RRI) and A be argsofRequest(RRI) in apSRMop(f,SS,A);

updateobjState (o1, o2, S) => addobjState(o2,remobjState(o1,S));

```

```

SRMopSet /* kind 1 */

/* All but the derived operator mitreSRMopSet are part of the template. */

emptySRMopSet : -> SRMopSet
addSRMopSet : SRMop SRMopSet -> SRMopSet
isinSRMopSet : SRMop SRMopSet -> Bool
mitreSRMopSet : -> SRMopSet

DISTINGUISHING SET isinSRMopSet;

emptySRMopSet => [<o> |-> false];

addSRMopSet (o, 0) =>
  [<oo> |-> if oo = o then true else isinSRMopSet(oo, 0)];

mitreSRMopSet =>
  addSRMopSet(uStore, addSRMopSet(Store, addSRMopSet(uCompute,
    addSRMopSet(Compute, addSRMopSet(uFetch, addSRMopSet(Fetch,
    addSRMopSet(uReadX, addSRMopSet(ReadX, addSRMopSet(uWriteX,
    addSRMopSet(WriteX, addSRMopSet(uRelease, addSRMopSet(Purge,
    addSRMopSet(uGet, addSRMopSet(Raise, addSRMopSet(Get,
    addSRMopSet(uSleep, addSRMopSet(Swap, emptySRMopSet))))))))))));

```



```
SRMop  /* kind 2 */
```

```
/* The last seventeen constructors (from uStore on) correspond to operations
   peculiar to this application. Those prefixed with "u" are user operations;
   the rest are system operations. The test okargsSRMop is not really part
   of the template, but we expect some such operator will be useful in
   simplifying SRMop definitions in most applications.*/
```

```
/* We have not given all the specifications of operators here (and in
   SecPol.spec and Interp.spec) due to space considerations.*/
```

```
nameSRMop : SRMop -> Symbol
apSRMop : SRMop State ArgList -> State
eqSRMop : SRMop SRMop -> Bool
NOop : -> SRMop
okargsSRMop : SRMop ArgList -> Bool
uStore : -> SRMop
Store : -> SRMop
uCompute : -> SRMop
Compute : -> SRMop
uFetch : -> SRMop
Fetch : -> SRMop
uReadX : -> SRMop
ReadX : -> SRMop
uWriteX : -> SRMop
WriteX : -> SRMop
uRelease : -> SRMop
Purge : -> SRMop
uGet : -> SRMop
Raise : -> SRMop
Get : -> SRMop
uSleep : -> SRMop
Swap : -> SRMop
```

```
DISTINGUISHING SET nameSRMop apSRMop;
```

```
eqSRMop (o1, o2) => nameSRMop(o1) = nameSRMop(o2);
```

```
NOop => ['NOop, <S,A> |-> S];
```

```
uReadX =>
  ['uReadX,
   <S, A> |->
     if ~okargsSRMop(uReadX, A)
     then errState
     else S];
```

```
ReadX =>
  ['ReadX,
   <S, A> |->
     if ~okargsSRMop(ReadX, A)
     then errState
```

```

else let 0 be objsofState(S) in
  let Xreg be findinObjectSet(isidObjectPred(X),0) in
  let Mb be getblkidArg(1,A) and Mn be getoffsetArg(2,A) in
  let B be findinObjectSet(isidObjectPred(Mb),0) in
  let c be assignblkContent(contofObject(B),Mn,
    regContent(contofObject(Xreg))) in
  let BB be updatecontObject(B,c) in updateobjState(B,BB,S)];

```

WriteX =>

```

['WriteX,
<S, A> |->
  if ~okargsSRMop(WriteX, A)
  then errState
  else let 0 be objsofState(S) in
    let Xreg be findinObjectSet(isidObjectPred(X),0) in
    let Mb be getblkidArg(1,A) and Mn be getoffsetArg(2,A) in
    let B be findinObjectSet(isidObjectPred(Mb),0) in
    let c be assignregContent(blkContent(contofObject(B),Mn)) in
    let XX be updatecontObject(Xreg,c) in updateobjState(Xreg,XX,S)];

```

okargsSRMop (o, A) =>

```

let N be nameSRMop(o) in
if N = 'ReadX | N = 'WriteX then
  (lengthArgList(A)=2 & ~getblkidArg(1,A)=errObjectId
  & ~getoffsetArg(2,A)=errInt)
else if N = 'uReadX | N = 'uWriteX then
  (lengthArgList(A)=2&~getublkidArg(1,A)=errInt&~getoffsetArg(2,A)=errInt)
else if N = 'Store | N = 'Fetch then
  (lengthArgList(A)=4 & ~getublkidArg(1,A)=errInt
  & ~getoffsetArg(2,A)=errInt & ~getregindArg(3,A)=errInt
  & ~getuidArg(3,A)=errProcessId)
else if N = 'uStore | N = 'uFetch then
  (lengthArgList(A)=3 & ~getublkidArg(1,A)=errInt
  & ~getoffsetArg(2,A)=errInt & ~getregindArg(3,A)=errInt)
else if N = 'Compute then
  (lengthArgList(A)=3 & ~getfnArg(1,A)=errComputeFn
  & ~getregindArg(2,A)=errInt & ~getuidArg(3,A)=errProcessId)
else if N = 'uCompute then
  (lengthArgList(A)=2 & ~getfnArg(1,A)=errComputeFn
  & ~getregindArg(2,A)=errInt)
else if N = 'uGet | N = 'uRelease then
  (lengthArgList(A)=1 & ~getublkidArg(1,A)=errInt)
else if N = 'Get then
  (lengthArgList(A)=2 & ~getuidArg(1,A)=errProcessId
  & ~getublkidArg(2,A)=errInt)
else if N = 'Purge | N = 'Raise then
  (lengthArgList(A)=1 & ~getblkidArg(1,A)=errObjectId)
else if N = 'uSleep | N = 'Swap then
  lengthArgList(A)=0
else false;

```

```

Scheduler /* kind 2 */

/* Additional constructors will be needed for systems which, unlike this
   application system, have nontrivial schedulers. The definitions of these
   constructors will normally be application dependent, because the structure
   of the components of a State -- e.g., Requests in the Requestlist -- will
   vary.*/

getreqScheduler : Scheduler State -> Request
getreqlisScheduler : Scheduler State -> RequestList
nullScheduler : -> Scheduler
trivScheduler : -> Scheduler

DISTINGUISHING SET getreqScheduler getreqlisScheduler;

nullScheduler => [<S> |-> nullRequest, <S> |-> reqsofState(S)];

trivScheduler =>
  [<S>|->hdRequestList(reqsofState(S)), <S>|->tlRequestList(reqsofState(S))];

```

```

Interp /* kind 2 */

/* All constructors from uStoreInterp on are peculiar to this application.
   We expect that in most applications, it will be convenient to define a
   new constructor individually for each SRMop, as we have done here, and
   collect them into one Interp analogous to our mitreInterp. */

getreqInterp : State Request Interp -> Request
trivInterp : -> Interp
uStoreInterp : -> Interp
StoreInterp : -> Interp
uComputeInterp : -> Interp
ComputeInterp : -> Interp
uFetchInterp : -> Interp
FetchInterp : -> Interp
uReadXInterp : -> Interp
ReadXInterp : -> Interp
uWriteXInterp : -> Interp
WriteXInterp : -> Interp
uReleaseInterp : -> Interp
PurgeInterp : -> Interp
uGetInterp : -> Interp
RaiseInterp : -> Interp
GetInterp : -> Interp
uSleepInterp : -> Interp
SwapInterp : -> Interp
mitreInterp : -> Interp

DISTINGUISHING SET getreqInterp;

trivInterp => [<S,r> |-> r];

uReadXInterp =>
  [<S,r> |->
    mkRequest(
      ReadX,
      let A be argsofRequest(r) and pid be prociidofRequest(r) in
      let i be getublkindArg(1,A) and j be getoffsetArg(2,A) and
        P be findinProcessSet(isidProcessPred(pid),procsofState(S)) in
      appendArgList(inttoArg(j),
        appendArgList(objidtoArg(idofObject(memblkofProcess(i,P))),
          nilArgList)),
      synProcessId)];

ReadXInterp => [<S,r> |-> nullRequest];

mitreInterp =>
  [<S,r> |->
    let f be opofRequest(r) in
    if nameSRMop(f) = 'uStore then getreqInterp(S,r,uStoreInterp)
    else if nameSRMop(f) = 'Store then getreqInterp(S,r,StoreInterp)
    else if nameSRMop(f) = 'uCompute then getreqInterp(S,r,uComputeInterp)

```

```

else if nameSRMop(f) = 'Compute then getreqInterp(S,r,ComputeInterp)
else if nameSRMop(f) = 'uFetch then getreqInterp(S,r,uFetchInterp)
else if nameSRMop(f) = 'Fetch then getreqInterp(S,r,FetchInterp)
else if nameSRMop(f) = 'uReadX then getreqInterp(S,r,uReadXInterp)
else if nameSRMop(f) = 'ReadX then getreqInterp(S,r,ReadXInterp)
else if nameSRMop(f) = 'uWriteX then getreqInterp(S,r,uWriteXInterp)
else if nameSRMop(f) = 'WriteX then getreqInterp(S,r,WriteXInterp)
else if nameSRMop(f) = 'uRelease then getreqInterp(S,r,uReleaseInterp)
else if nameSRMop(f) = 'Purge then getreqInterp(S,r,PurgeInterp)
else if nameSRMop(f) = 'uGet then getreqInterp(S,r,uGetInterp)
else if nameSRMop(f) = 'Raise then getreqInterp(S,r,RaiseInterp)
else if nameSRMop(f) = 'Get then getreqInterp(S,r,GetInterp)
else if nameSRMop(f) = 'uSleep then getreqInterp(S,r,uSleepInterp)
else if nameSRMop(f) = 'Swap then getreqInterp(S,r,SwapInterp)
else errRequest];

```

```
SecPol /* kind 2 */
```

```
/* All constructors from uStoreSecPol on are peculiar to this application.
   We expect that in most applications, it will be convenient to define a
   new constructor individually for each SRMop, as we have done here, and
   collect them into one SecPol analogous to our mitreSecPol. */
```

```
chkwSecPol : State Request SecPol -> Bool
getreqSecPol : State Request SecPol -> Request
noPol : -> SecPol
recalcitrantPol : -> SecPol
uStoreSecPol : -> SecPol
StoreSecPol : -> SecPol
uComputeSecPol : -> SecPol
ComputeSecPol : -> SecPol
uFetchSecPol : -> SecPol
FetchSecPol : -> SecPol
uReadXSecPol : -> SecPol
ReadXSecPol : -> SecPol
uWriteXSecPol : -> SecPol
WriteXSecPol : -> SecPol
uReleaseSecPol : -> SecPol
PurgeSecPol : -> SecPol
uGetSecPol : -> SecPol
RaiseSecPol : -> SecPol
GetSecPol : -> SecPol
uSleepSecPol : -> SecPol
SwapSecPol : -> SecPol
mitreSecPol : -> SecPol
```

```
DISTINGUISHING SET chkwSecPol getreqSecPol;
```

```
noPol => [<S, r> |-> true, <S, r> |-> r];
```

```
recalcitrantPol => [<S, r> |-> false, <S, r> |-> nullRequest];
```

```
uReadXSecPol =>
  [<S, r> |-> true, <S, r> |-> r];
```

```
ReadXSecPol =>
  [<S, r> |-> let p be findinProcessSet(isidProcessPred(procIdofRequest(r)),
                                     procsofState(S))
              in isprivileged(p),
   <S, r> |-> if chkwSecPol(S,r,ReadXSecPol) then r else nullRequest];
```

```
mitreSecPol =>
  [<S, r> |->
    let f be opofRequest(r) in
    if nameSRMop(f) = 'uStore then chkwSecPol(S,r,uStoreSecPol)
    else if nameSRMop(f) = 'Store then chkwSecPol(S,r,StoreSecPol)
    else if nameSRMop(f) = 'uCompute then chkwSecPol(S,r,uComputeSecPol)
    else if nameSRMop(f) = 'Compute then chkwSecPol(S,r,ComputeSecPol)
```

```

else if nameSRMop(f) = 'uFetch then chkWSecPol(S,r,uFetchSecPol)
else if nameSRMop(f) = 'Fetch then chkWSecPol(S,r,FetchSecPol)
else if nameSRMop(f) = 'uReadX then chkWSecPol(S,r,uReadXSecPol)
else if nameSRMop(f) = 'ReadX then chkWSecPol(S,r,ReadXSecPol)
else if nameSRMop(f) = 'uWriteX then chkWSecPol(S,r,uWriteXSecPol)
else if nameSRMop(f) = 'WriteX then chkWSecPol(S,r,WriteXSecPol)
else if nameSRMop(f) = 'uRelease then chkWSecPol(S,r,uReleaseSecPol)
else if nameSRMop(f) = 'Purge then chkWSecPol(S,r,PurgeSecPol)
else if nameSRMop(f) = 'uGet then chkWSecPol(S,r,uGetSecPol)
else if nameSRMop(f) = 'Raise then chkWSecPol(S,r,RaiseSecPol)
else if nameSRMop(f) = 'Get then chkWSecPol(S,r,GetSecPol)
else if nameSRMop(f) = 'uSleep then chkWSecPol(S,r,uSleepSecPol)
else if nameSRMop(f) = 'Swap then chkWSecPol(S,r,SwapSecPol)
else false,
<S, r> |->
  let f be opofRequest(r) in
  if nameSRMop(f) = 'uStore then getreqSecPol(S,r,uStoreSecPol)
  else if nameSRMop(f) = 'Store then getreqSecPol(S,r,StoreSecPol)
  else if nameSRMop(f) = 'uCompute then getreqSecPol(S,r,uComputeSecPol)
  else if nameSRMop(f) = 'Compute then getreqSecPol(S,r,ComputeSecPol)
  else if nameSRMop(f) = 'uFetch then getreqSecPol(S,r,uFetchSecPol)
  else if nameSRMop(f) = 'Fetch then getreqSecPol(S,r,FetchSecPol)
  else if nameSRMop(f) = 'uReadX then getreqSecPol(S,r,uReadXSecPol)
  else if nameSRMop(f) = 'ReadX then getreqSecPol(S,r,ReadXSecPol)
  else if nameSRMop(f) = 'uWriteX then getreqSecPol(S,r,uWriteXSecPol)
  else if nameSRMop(f) = 'WriteX then getreqSecPol(S,r,WriteXSecPol)
  else if nameSRMop(f) = 'uRelease then getreqSecPol(S,r,uReleaseSecPol)
  else if nameSRMop(f) = 'Purge then getreqSecPol(S,r,PurgeSecPol)
  else if nameSRMop(f) = 'uGet then getreqSecPol(S,r,uGetSecPol)
  else if nameSRMop(f) = 'Raise then getreqSecPol(S,r,RaiseSecPol)
  else if nameSRMop(f) = 'Get then getreqSecPol(S,r,GetSecPol)
  else if nameSRMop(f) = 'uSleep then getreqSecPol(S,r,uSleepSecPol)
  else if nameSRMop(f) = 'Swap then getreqSecPol(S,r,SwapSecPol)
  else nullRequest];

```

```
ObjectSet /* kind 1 */
```

```
/* This is an implementation of the "ideal" ObjectSet specification that
   is better for interactive use in that it speeds the computation necessary
   to display an ObjectSet and to perform the operation findinObjectSet.
   These things could, in principle, be done by allowing quantifiers
   (specifically, the "some" quantifier) in the definitions of findinObjectSet,
   firstinObjectSet, and restofObjectSet; such definitions would remain
   executable in the FASE system. */
```

```
emptyObjectSet : -> ObjectSet
addObjectSet : Object ObjectSet -> ObjectSet
remObjectSet : Object ObjectSet -> ObjectSet
isinObjectSet : Object ObjectSet -> Bool
findinObjectSet : ObjectPred ObjectSet -> Object
firstinObjectSet : ObjectSet -> Object
restofObjectSet : ObjectSet -> ObjectSet
isemptyObjectSet : ObjectSet -> Bool
```

```
DISTINGUISHING SET firstinObjectSet restofObjectSet;
```

```
emptyObjectSet => [ errObject, emptyObjectSet ];
```

```
addObjectSet (o, S) =>
  if isemptyObjectSet(S)
  | precedesObject(o, firstinObjectSet(S))
  then [ o, S ]
  else if o = firstinObjectSet(S)
  then S
  else [ firstinObjectSet(S),
         addObjectSet(o, restofObjectSet(S)) ];
```

```
remObjectSet (o, S) =>
  if isemptyObjectSet(S) then S
  else if o = firstinObjectSet(S) then restofObjectSet(S)
  else [ firstinObjectSet(S),
         remObjectSet(o, restofObjectSet(S)) ];
```

```
isinObjectSet (o, S) =>
  if isemptyObjectSet (S)
  then false
  else (o = firstinObjectSet(S) | isinObjectSet(o, restofObjectSet(S)));
```

```
findinObjectSet (P, S) =>
  if isemptyObjectSet(S)
  then errObject
  else let o be firstinObjectSet(S) in
    if apObjectPred(P, o)
    then o
    else findinObjectSet(P, restofObjectSet(S));
```

```
isemptyObjectSet (S) => iserrObject(firstinObjectSet(S));
```


ObjectPred

```
/* The sort ObjectPred exists in order to make the definition of
   findinObjectSet, useful in the definitions of many SRMops, independent
   of the application. */
```

```
/* Only the constructor isidObjectPred is specific to this application. */
```

```
apObjectPred : ObjectPred Object -> Bool
trivObjectPred : Bool -> ObjectPred
isidObjectPred : ObjectId -> ObjectPred
```

```
DISTINGUISHING SET apObjectPred;
```

```
trivObjectPred (b) => [<o> |-> b];
```

```
isidObjectPred (i) => [<o> |-> idofObject(o) = i];
```

```

ObjectId /* kind 3 */

/* Only the declarations of eqObjectId and precedesObjectId in this
   specification are part of the template. */

/* The carrier of sort ObjectId in this application is essentially the union
   of one symbol ('X') with Int and the product of Int and ProcessId. */

eqObjectId : ObjectId ObjectId -> Bool
precedesObjectId : ObjectId ObjectId -> Bool
X : -> ObjectId
mkregObjectId : Int ProcessId -> ObjectId
mkblkObjectId : Int -> ObjectId
typeofObjectId : ObjectId -> Symbol
symofObjectId : ObjectId -> Symbol
intofObjectId : ObjectId -> Int
procidofObjectId : ObjectId -> ProcessId
isregObjectId : ObjectId -> Bool
isblkObjectId : ObjectId -> Bool
isX : ObjectId -> Bool

DISTINGUISHING SET typeofObjectId symofObjectId intofObjectId procidofObjectId;

eqObjectId (i1, i2) =>
  typeofObjectId(i1)=typeofObjectId(i2)
  & symofObjectId(i1)=symofObjectId(i2)
  & intofObjectId(i1)=intofObjectId(i2)
  & procidofObjectId(i1)=procidofObjectId(i2);

precedesObjectId (i1, i2) =>
  if (typeofObjectId(i1)='Register & typeofObjectId(i2)='Memblock) then true
  else if (typeofObjectId(i1)='Memblock & typeofObjectId(i2)='Register) then false
  else if symofObjectId(i2)='X then false
  else if symofObjectId(i1)='X then true
  else if (typeofObjectId(i1)='Memblock & typeofObjectId(i2)='Memblock
    & intofObjectId(i1) < intofObjectId(i2))
    then true
  else if precedesProcessId(procidofObjectId(i1), procidofObjectId(i2)) then true
  else eqProcessId(procidofObjectId(i1), procidofObjectId(i2))
    & intofObjectId(i1) < intofObjectId(i2);

X => ['Register, 'X, errInt, errProcessId];

mkregObjectId (n, i) => ['Register, errSymbol, n, i];
mkblkObjectId (n) => ['Memblock, errSymbol, n, errProcessId];

isregObjectId (i) => typeofObjectId(i) = 'Register;
isblkObjectId (i) => typeofObjectId(i) = 'Memblock;
isX (i) => symofObjectId(i) = 'X;

```

```
Object /* kind 3 */
```

```
/* Only the name and arity of the operator idofObject is part of the template specification. */
```

```
/* There are two kind of (reachable) Objects: register objects and memory block objects. Register objects are "owned" by user (i.e, non-system) processes, with the exception of the X Register. Memory block objects are not owned, although user processes are partly characterized by the blocks to which they have access. The facts of ownership are imbedded in the ObjectIds. */
```

```
idofObject : Object -> ObjectId  
eqObject : Object Object -> Bool  
contofObject : Object -> Content  
lowindexObject : Object -> Int  
highindexObject : Object -> Int  
levofObject : Object -> Level  
mkregObject : ObjectId Content Level -> Object  
mkblkObject : ObjectId Content Level Int Int -> Object  
precedesObject : Object Object -> Bool  
updatecontObject : Object Content -> Object
```

```
DISTINGUISHING SET idofObject contofObject levofObject  
lowindexObject highindexObject;
```

```
eqObject (o1, o2) => idofObject(o1) = idofObject(o2);
```

```
iserrObject (mkregObject (i, c, l)) => ~(isregObjectId(i) & isregContent(c));
```

```
mkregObject (i, c, l) => [i, c, l, errInt, errInt];
```

```
iserrObject (mkblkObject (i, c, l, lo, hi)) =>  
  ~(isblkObjectId(i) & isblkContent(c)) | hi < lo;
```

```
mkblkObject (i, c, l, lo, hi) => [i, c, l, lo, hi];
```

```
precedesObject (o1, o2) => precedesObjectId(idofObject(o1), idofObject(o2));
```

```
iserrObject (updatecontObject (o, c)) =>  
  ~typeofObjectId(idofObject(o))=typeofContent(c);
```

```
updatecontObject (o, c) =>  
  [idofObject(o), c, levofObject(o), lowindexObject(o), highindexObject(o)];
```

```
ProcessSet /* kind 1 */
```

```
/* This is an implementation of the "ideal" ProcessSet specification that is better
   for interactive use in that it speeds the computation necessary to display a
   ProcessSet and to perform the operation findinProcessSet. These things could, in
   principle, be done by allowing quantifiers (specifically, the "some" quantifier)
   in the definitions of findinProcessSet, firstinProcessSet, and restofProcessSet;
   such definitions would remain executable in the FASE system. */
```

```
emptyProcessSet : -> ProcessSet
addProcessSet : Process ProcessSet -> ProcessSet
remProcessSet : Process ProcessSet -> ProcessSet
isinProcessSet : Process ProcessSet -> Bool
findinProcessSet : ProcessPred ProcessSet -> Process
firstinProcessSet : ProcessSet -> Process
restofProcessSet : ProcessSet -> ProcessSet
isemptyProcessSet : ProcessSet -> Bool
```

```
DISTINGUISHING SET firstinProcessSet restofProcessSet;
```

```
emptyProcessSet => [ errProcess, emptyProcessSet ];
```

```
isinProcessSet (p, S) =>
  if isemptyProcessSet (S)
  then false
  else (p = firstinProcessSet(S) | isinProcessSet(p,restofProcessSet(S)));
```

```
addProcessSet (p, S) =>
  if isemptyProcessSet(S) | precedesProcess(p, firstinProcessSet(S))
  then [p, S]
  else if p = firstinProcessSet(S)
  then S
  else [ firstinProcessSet(S),
         addProcessSet(p,restofProcessSet(S)) ];
```

```
remProcessSet (p, S) =>
  if isemptyProcessSet(S) then S
  else if p = firstinProcessSet(S) then restofProcessSet(S)
  else [ firstinProcessSet(S),
         remProcessSet(p,restofProcessSet(S)) ];
```

```
isemptyProcessSet (S) => iserrProcess(firstinProcessSet(S));
```

```
findinProcessSet (P, S) =>
  if isemptyProcessSet(S)
  then errProcess
  else let o be firstinProcessSet(S) in
    if apProcessPred(P,o)
    then o
    else findinProcessSet(P,restofProcessSet(S));
```

```
isemptyProcessSet (S) => iserrProcess(firstinProcessSet(S));
```

ProcessPred

```
/* The sort ProcessPred exists in order to make the definition of  
   findinProcessSet, useful in the definitions of many SRMops, independent  
   of the application. */
```

```
/* Only the constructor isidProcessPred is specific to this application. */
```

```
apProcessPred : ProcessPred Process -> Bool  
trivProcessPred : Bool -> ProcessPred  
isidProcessPred : ProcessId -> ProcessPred
```

```
DISTINGUISHING SET apProcessPred;
```

```
trivProcessPred (b) => [<p> |-> b];
```

```
isidProcessPred (i) => [<p> |-> idofProcess(p) = i];
```

```

ProcessId /* kind 3 */

/* Only the declarations of eqProcessId and precedesProcessId in this
specification are part of the template. */

/* The carrier of the sort ProcessId is simply a union of Int with one symbol
('System'). */

sysProcessId : -> ProcessId
mkuserProcessId : Int -> ProcessId
typeofProcessId : ProcessId -> Symbol
intofProcessId : ProcessId -> Int
eqProcessId : ProcessId ProcessId -> Bool
precedesProcessId : ProcessId ProcessId -> Bool

DISTINGUISHING SET typeofProcessId intofProcessId;

sysProcessId => ['System, errInt];

mkuserProcessId (n) => ['User, n];

eqProcessId (i1, i2) => intofProcessId(i1)=intofProcessId(i2);

precedesProcessId (i1, i2) =>
  if typeofProcessId(i2) = 'System then false
  else if typeofProcessId(i1) = 'System then true
  else intofProcessId(i1) < intofProcessId(i2);

```

```

Process /* kind 3 */

/* Only the name and arity of the operator idofProcess is part of the template
specification. */

idofProcess : Process -> ProcessId
eqProcess : Process Process -> Bool
inofProcess : Process -> Bool
levofProcess : Process -> Level
memblkofProcess : Int Process -> Object
regofProcess : Int Process -> Object
systemProcess : -> Process
mkuserProcess : Int Level Int -> Process
swapinProcess : Process -> Process
swapoutProcess : Process -> Process
precedesProcess : Process Process -> Bool
isprivileged : Process -> Bool
isregofProcess : Int Process -> Bool
updregProcess : Process Int Content -> Process
updmemblkProcess : Process Int Object -> Process

DISTINGUISHING SET idofProcess inofProcess levofProcess
regofProcess memblkofProcess;

eqProcess (p1, p2) => idofProcess(p1) = idofProcess(p2);

systemProcess =>
[ sysProcessId, true, syshi, <n> |-> errObject, <n> |-> errObject ];

mkuserProcess (i, l, k) =>
[ mkuserProcessId(i), false, l,
  <n> |-> if k > n then errObject
        else mkregObject(mkregObjectId(i,mkuserProcessId(i)),
                          errContent,
                          errLevel),
  <n> |-> errObject ];

swapinProcess (p) =>
[idofProcess(p), true, levofProcess(p),
 <n> |-> regofProcess(n,p), <n> |-> memblkofProcess(n,p)];

swapoutProcess (p) =>
[idofProcess(p), false, levofProcess(p),
 <n> |-> regofProcess(n,p), <n> |-> memblkofProcess(n,p)];

precedesProcess (p1, p2) => precedesProcessId(idofProcess(p1), idofProcess(p2));

isprivileged (p) => idofProcess(p) = sysProcessId;

isregofProcess (n, p) => ~iserrObject(regofProcess (n, p));

updregProcess (p, n, c) =>

```

```

[idofProcess(p), inofProcess(p), levofProcess(p),
  <m> |-> if m=n then updatecontObject(regofProcess(m,p),c)
        else regofProcess(m,p),
  <m> |-> memblkofProcess(m,p)];

updmemblkProcess (p, n, o) =>
[idofProcess(p), inofProcess(p), levofProcess(p), <m> |-> regofProcess(m,p),
  <m> |-> if m=n then o else memblkofProcess(m,p)];

```



```

RequestList /* kind 1 */

nilRequestList : -> RequestList
nullRequestList : RequestList -> Bool
prependRequestList : Request RequestList -> RequestList
appendRequestList : Request RequestList -> RequestList
concatRequestList : RequestList RequestList -> RequestList
remRequestList: Int RequestList -> RequestList
hdRequestList: RequestList -> Request
tlRequestList: RequestList -> RequestList

DISTINGUISHING SET hdRequestList tlRequestList;

nilRequestList => [errRequest, nilRequestList];

nullRequestList (R) => iserrRequest(hdRequestList(R));

prependRequestList (r, R) => [r, R];

appendRequestList (r, R) =>
  if nullRequestList(R) then [r, R]
  else [hdRequestList(R), appendRequestList(r,tlRequestList(R))];

concatRequestList (R1, R2) =>
  if nullRequestList(R1) then R2
  else [hdRequestList(R1), concatRequestList(tlRequestList(R1),R2)];

remRequestList (n, R) =>
  if nullRequestList(R) then R
  else if n = 1 then tlRequestList(R)
  else [hdRequestList(R), remRequestList(n-1,tlRequestList(R))];

```

Request

```
nullRequest : -> Request
opofRequest : Request -> SRMop
argsofRequest : Request -> ArgList
oktoRequest : Request State -> Bool
procidofRequest : Request -> ProcessId
mkRequest : SRMop ArgList ProcessId -> Request

DISTINGUISHING SET opofRequest argsofRequest procidofRequest;

nullRequest => [NOop, errArgList, sysProcessId];

mkRequest (f, A, i) => [f, A, i];

oktoRequest (r, S) =>
  isprivileged(findinProcessSet(isidProcessPred(procidofRequest(r)),
                                procsofState(S)));
```

History /* kind 3 */

/* Only the operations initHistory and appendHistory, with their declared arities, are part of the template. Their definitions and the remaining operations are peculiar to this application. */

startHistory : History -> State
listofHistory : History -> RequestList
initHistory : State -> History
appendHistory : Request History -> History

DISTINGUISHING SET startHistory listofHistory;

initHistory (S) => [S, nilRequestList];

appendHistory (r, H) => [startHistory(H), appendRequestList(r, listofHistory(H))];

```

ArgList /* kind 1 */

nilArgList : -> ArgList
nullArgList : ArgList -> Bool
appendArgList : Arg ArgList -> ArgList
hdArgList : ArgList -> Arg
tlArgList : ArgList -> ArgList
nthArgList : Int ArgList -> Arg
lengthArgList : ArgList -> Int

DISTINGUISHING SET hdArgList tlArgList;

nilArgList => [errArg, nilArgList];

nullArgList (R) => iserrArg(hdArgList(R));

appendArgList (r, R) =>
  if nullArgList(R) then [r, R]
  else [hdArgList(R), appendArgList(r,tlArgList(R))];

nthArgList (n, R) =>
  if n<1 then errArg
  else if n=1 then hdArgList(R)
  else nthArgList(n-1,tlArgList(R));

lengthArgList (R) =>
  if nullArgList(R) then 0 else 1 + lengthArgList(tlArgList(R));

```

```
Arg /* kind 3 */
```

```
/* Arg is an application dependent union type allowing a variety of arguments.
   ObjectIds and ProcessIds are guaranteed to be valid arguments in any
   application, and hence objidtoArg, objidofArg, procidtoArg, and procidofArg
   are part of the template. Int and ComputeFn Args are special to this
   application. The get...Arg operators are also special to this application,
   but we anticipate that their analogues (which, essentially, select
   arguments and do the appropriate coercions) will be useful in defining the
   SRMops of most other applications as well. */
```

```
objidtoArg : ObjectId -> Arg
procidtoArg : ProcessId -> Arg
inttoArg : Int -> Arg
fntoArg : ComputeFn -> Arg
objidofArg : Arg -> ObjectId
procidofArg : Arg -> ProcessId
intofArg : Arg -> Int
fnofArg : Arg -> ComputeFn
getblkidArg : Int ArgList -> ObjectId
getublkidArg : Int ArgList -> Int
getoffsetArg : Int ArgList -> Int
getregindArg : Int ArgList -> Int
getuidArg : Int ArgList -> ProcessId
getfnArg : Int ArgList -> ComputeFn
```

```
DISTINGUISHING SET objidofArg procidofArg intofArg fnofArg;
```

```
objidtoArg (i) => [i, errProcessId, errInt, errComputeFn];
```

```
procidtoArg (i) => [errObjectId, i, errInt, errComputeFn];
```

```
inttoArg (n) => [errObjectId, errProcessId, n, errComputeFn];
```

```
fntoArg (f) => [errObjectId, errProcessId, errInt, f];
```

```
getblkidArg (n, A) => objidofArg(nthArgList(n, A));
```

```
getublkidArg (n, A) => intofArg(nthArgList(n, A));
```

```
getoffsetArg (n, A) => intofArg(nthArgList(n, A));
```

```
getregindArg (n, A) => intofArg(nthArgList(n, A));
```

```
getuidArg (n, A) => procidofArg(nthArgList(n, A));
```

```
getfnArg (n, A) => fnofArg(nthArgList(n, A));
```

```
Content /* kind 4 */
```

```
/* The sort Content is specific to this application. A Content "is" either an
   Int (this will be the content of a register) or a map from Int to Int (which
   will be the content of a memory block). */
```

```
typeofContent : Content -> Symbol
regContent : Content -> Int
blkContent : Content Int -> Int
assignregContent : Int -> Content
assignblkContent : Content Int Int -> Content
isregContent : Content -> Bool
isblkContent : Content -> Bool
```

```
DISTINGUISHING SET typeofContent regContent blkContent;
```

```
assignregContent (n) => ['Register, n, <m> |-> errInt];
```

```
iserrContent (assignblkContent (c, n, N)) => typeofContent(c) = 'Register;
```

```
assignblkContent (c, n, N) =>
  ['Memblock, errInt, <m> |-> if m=n then N else blkContent(c,m)];
```

```
isregContent (c) => typeofContent(c) = 'Register;
```

```
isblkContent (c) => typeofContent(c) = 'Memblock;
```

```
Level /* kind 4 */
```

```
/* The sort Level is particular to this application. It is essentially the  
union of the sort Symbol with a one-element set (containing syshi). The  
intent is to have a distinguished highest Level and a selection of other  
otherwise incomparable Levels. */
```

```
eqLevel : Level Level -> Bool  
syshi : -> Level  
mkLevel : Symbol -> Level  
symofLevel : Level -> Symbol  
issyshi : Level -> Bool
```

```
DISTINGUISHING SET issyshi symofLevel;
```

```
syshi => [true, errSymbol];
```

```
mkLevel (s) => [false, s];
```

```
eqLevel (l1, l2) => issyshi(l1)=issyshi(l2) & symofLevel(l1)=symofLevel(l2);
```

```
RegAssn /* kind 4 */
```

```
/* The sort RegAssn is specific to this application. Its function is to  
facilitate the definition of elements of sort ComputeFn, which permit  
user processes to do computations on the contents of the registers available  
to them. */
```

```
retrRegAssn : Int RegAssn -> Int  
procRegAssn : Process -> RegAssn
```

```
DISTINGUISHING SET retrRegAssn;
```

```
procRegAssn (p) => [<n> |-> regContent(contofObject(regofProcess(n,p)))];
```



```

ComputeFn  /* kind 4 */

/* We have defined only the most obvious of the possible elements of sort
   ComputeFn here.  Clearly, one could also define any desired arithmetic
   operations. */

nameofComputeFn : ComputeFn -> Symbol
evalComputeFn : ComputeFn RegAssn -> Int
selectComputeFn : Int -> ComputeFn
constantComputeFn : Int -> ComputeFn

DISTINGUISHING SET nameofComputeFn evalComputeFn;

selectComputeFn (n) => [ 'select, <R> |-> retrRegAssn(n,R) ];

constantComputeFn (n) => [ 'constant, <R> |-> n ];

```

C High-level description of simple PLANNER

```

program
  state Memory, Registers, Mmu, RegX, Uid, WaitingUid
  begin Memory
    doStoreMem(Memoryin, Mb, Ml, Rn, Memoryout) ;
    doPurge(Memoryin, Memoryout) ;
    doRaise(Memoryin, Mb, Memoryout) ;
    doGet(Memoryin, Mmuin, Mb, MMUId, Memoryout, Mmuout)
  end;

  begin Registers
    doFetchMem(Memoryin, Mb, Ml, Rn, Memoryout) ;
    doReadX(Memoryin, Mb, Ml, Memoryout)
  end;

  begin RegX
    doWriteX(Xin, Newvalue, Xout)
  end;

  begin Mmu
    doGet(Memoryin, Mmuin, Mb, MMUId, Memoryout, Mmuout)
  end;

  begin Uid
    doSwap(UidIn, WaitingUidIn, UidOut, WaitingUidOut)
  end;

  begin WaitingUid
    doSwap(UidIn, WaitingUidIn, UidOut, WaitingUidOut)
  end

  operation Store(Mb, Mn, Rn) with
    state MemoryOut, Registers, Mmu, RegX, Uid, WaitingUid
    produces Memory: [[Mphys, Mn, undefined], [Mphys, Mn, Newval]]
    by doStoreMem( Memory, Mphys, Mn, Newval, MemoryOut)
    with
      precondition fetchReg(Registers, Uid, Rn, Newval)
      trigger Register
      difference [[Uid, Rn, undefined], [Uid, Rn, Newval]]
    end ;
    precondition memMap(Mmu, Uid, Mphys, Mb)
    trigger User
    difference [[Uid], [WaitingUid]]
  end

end;

  operation Fetch(Mb, Mn, Rn) with
    state Memory, RegistersOut, Mmu, RegX, Uid, WaitingUid
    produces Registers: [[User, Rn, undefined], [User, Rn, Newval]]

```

END

DATE FILMED

02 / 26 / 91