A Software Engineering Approach to Designing Trustworthy Software

Mark R. Cornwell*

Center for Secure Information Technology
Naval Research Laboratory
Washington, D.C. 20375

Abstract

The literature is full of exhoratations to use formal techniques to help design trustworthy systems. We are advised to produce a formal security model, a formal top-level specification, and show a correspondence between the two. At the same time, software engineering philosophers tell us to extensively and formally document our design decisions in copius design documents. However, examples of these techniques rigorously applied to real systems are few. This paper records the results of taking to heart this common wisdom and attempting to apply it to the Secure Military Message System project over a period of several years. Formal techniques found helpful on the project are illustrated in detail by a scaled-down example.

1. Introduction

It is widely accepted that good software engineering practice is essential to the design of trustworthy software. Hence, it follows that if one is to construct a trustworthy system one should use the best software engineering techniques available. However, worked examples of software systems that apply a coherent, fully documented software engineering methodology are difficult to find. Lacking a paradigm, designers of secure software systems find themselves without clear guidance for the design documents and other work products that should be produced as part of a disciplined software engineering effort for a system with strong trust requirements.

The SMMS project has been experimenting with a software engineering approach to designing trustworthy software systems. The goal of the project is to produce a prototype system and a set of supporting design documents that illustrate a coherent software engineering methodology. If the project is fully successful these products will validate the methodology and serve as a model for future designers of secure systems.

This paper focuses on some specific formal techniques that have proven useful in the SMMS design. Since it would be impractical to describe in detail a significant portion of the SMMS design, we focus on a simplified example derived from our experiences with the actual design. The formal techniques we illustrate incorporate the use of a formal security model, formal specifications of module interfaces, and proofs of correspondence between the two. In designing the SMMS the need to prove specific security properties drove many of our design decisions and strongly influenced the design that resulted.

2. Benefits

In order to encourage the production of a design for which the strongest possible analytical evidence can be provided an interesting approach involves the construction of a security proof early in the design process. The goal is to motivate design decisions which lead to a system amenable to proof. Design decisions that make a security proof difficult are inhibited by the fact that the design and the proof are developed together.

By employing a proof-driven approach we hope to achieve the following benefits:

Security design can proceed in an orderly fashion. An early security proof provides guidance on how to proceed with the design on the basis of security requirements. It generates questions that designers must answer in order to construct the proof. Answers to these questions become design decisions. This process forces design decisions that are consistent with the security goals of the design.

The system will have high security assurance. A product of the methodology is a design-level security proof. The existence of this security proof for the system design should help provide strong analytical evidence necessary for assurance.

Security flaws are exposed early in the design. The requirement that security arguments be provided uncovers open issues and gaps in the security of the design. By exposing these gaps early, designers are in a better position to modify the design to eliminate or at least cope with the security flaws.

The system will be more amenable to change. Design decisions used in the security proof are recorded in the proof. This allows designers to determine how changes to parts of the design affect security assurance. Design decisions not used in the proof can be changed without affecting the proof. Design decisions used in the proof cannot be changed without revising the proof.

Specific security responsibilities will be traceable to individual components of the system. An early security proof generates a set of specific security responsibilities. By partitioning security responsibilities among system components based on the proof, one can ensure that the responsibilities and components are simple enough to be assured. This increases our chances of verifying that each component meets its responsibility. This is easiest when we have simple components with simple responsibilities.

3. On the Role of Proofs

A security proof can increase our confidence that a design meets its security requirements. Required security properties are linked by an explicit chain of reasoning to an explicit set of assumptions. This makes reviewing the system's security easier by separating concerns. To evaluate the design and its proof, we ask three separate questions:

(1) Do the properties stated accurately reflect the intuitive security requirements of the users? Do the users agree now that they do? Will they continue to agree in the future?

(2) Is the chain of reasoning valid? Do we accept that each step in the argument is a valid consequence of prior steps?

(3) Are the assumptions upon which the proof is based acceptable?

A major benefit of attempting proofs is that it enables designers to uncover critical hidden assumptions. In constructing a proof it is necessary to state assumptions just to force the proof to go through. Each assumption must either be provided for within the design itself or be something the

*Author's current address: The MITRE Corporation, 7525 Colshire Drive, McLean, VA, 22102.

designers assume about the the system outside their specific design responsibility. If an assumption must be provided within the design it becomes a design decision. With such a design decision there comes the obligation of recording it in the design documentation and maintaining review procedures to ensure that the remainder of the design and implementation effort follows through with this decision. Similarly, if the assumption is one made about its external environment there is an obligation to record it in the system documentation so users of the system will know what environment they must provide in order to trust the system.

Having recorded lists of assumptions, we review them for their acceptability. When assumptions appear that we cannot accept, we must go back and change the proof not to take advantage of such assumptions. This will entail either changing design decisions or changing assumptions that the system makes about its external environment.

4. A Detailed Example

We will illustrate our approach with a small, idealized example. In order to focus on methodological considerations, the example is deliberately kept simple. The security model used is a simplification of the MMS security model [1]. The example module specification is a simplification of a similar specification appearing in the SMMS prototype design [2]. Some of the specification techniques may appear inappropriate for such a small example, but they reflect techniques intended for large designs. The specific specification formats have been tuned for a set of design documents describing several hundred access programs. Our example describes only seven.

In this example, we present a formal application-based security model and outline some preliminary design decisions. In order to talk more precisely about the design we describe a mathematical structure for specifying a program module. Using that structure we define a specific module called the Entity Monitor or *EM* for short. In order to state a security theorem we define an interpretation mapping the *EM* module interface to the security model. Next we attempt to prove that the system is secure under this interpretation and in doing so discover specific additional assumptions we must make. The assumptions become design decisions. Finally we amend the specification to incorporate the additional design decisions.

5. Notation

This paper uses a number of different notations. The security model is presented in the familiar concise notational style of mathematical journal articles. In contrast, the module specifications are presented in strictly formated tables using notation developed to support writing large specifications. These large specifications can include hundreds or even thousands of programs, exceptions, and data types requiring symbolic names. For this reason the specifications abandon the short naming conventions of the security model and use long identifiers for the objects they deal with.

The module specifications also use special bracketing conventions to indicate the class of object an identifier denotes. For example, function names are bracketed with "++", exception names are bracketed with "%( )%". These conventions run against the taste of most people encountering them for the first time, but experience with real system developments indicates that the benefits outweigh this initial distaste.

Except for the bracketing conventions, most of the other mathematical notation we use is standard. Due to limitations of our text processing equipment (as well as personal taste) we use **A** to represent the universal quantifier, and **E** to represent the existential quantifier. Given a set S and a predicate P, when the predicate $P(s)$ is defined for $s \in S$, we define $P(S)$ to mean $\mathbf{A}s \in S.P(s)$. Implication is denoted as $a \rightarrow b$ (read "a implies b") and associates on the right. Thus $a \rightarrow b \rightarrow c$ means $a \rightarrow (b \rightarrow c)$.

6. Example Security Model

Our example security model is based on a subset of the MMS security model. This model might be appropriate to describe the security properties of a very simple document system or file system supporting security labels. This simple model is based on analogy to the paper world. The paper world includes documents, folders, or safes upon which a classification marking appears. Safes may contain folders and folders may contain documents. Security labels determine when it is acceptable for one of these things to contain another. It is typically considered unacceptable to store SECRET documents in folders marked UNCLASSIFIED.

In this formal security model *entities* are the analog to documents, folders, and safes. Each entity is associated with a *classification*. Analogously, entities may contain other entities. The system is insecure if the security level of each entity does not dominate the security level of all of the entities it contains. The mathematical formalism below is designed to capture this very simple requirement precisely.

We assume the existence of the following sets.

L is a set of security levels. $\geq$ is a partial order on L such that $(L, \geq)$ is a lattice.

ES is a set of entities. For all $e \in ES$ $CE(e) \in L$ is the classification of e. ES contains as a subset the set of entities that are containers. For any entity e in this set $H(e) = <e_1, \cdots e_n>$ where entity e_i is the *ith* entity contained in e.

RF is a set of references (e.g. strings available for naming entities)

A *reference function*, E, is a mapping from a (possibly improper) subset of RF into ES. For any reference r, if $E(r)=e$, we say that r is a reference to e (relative to E).

Definition (1):

A *system state*, s, is a 1-tuple (E) where E is a reference function.

Given a system state $s=(E)$, we abbreviate $E(r)$ by r_e,

Definition (2):

A state s is *secure* if $\mathbf{A}\ x, y \in rng(E)$:

$x \in H(y) \rightarrow CE(x) \leq CE(y)$,

Definition (3):

A *system* Σ is a 4-tuple (I, S, s_0, T) where

I is the set of well-formed system requests.

S is the set of possible system states;

s_0 designates a special state called the *initial state*; and

T is the *system transform*, i.e., a function from $I \times S$ into *powerset of* S.

Definition (4):

A *history*, Π, of a system is a function from the set of nonnegative integers N to $I \times S$ such that (1) the second element of $\Pi(0)$ is s_0, and (2) for all $n \in N$, if $\Pi(n)=(i,s)$ and $\Pi(n+1)=(i^*,s^*)$, then $s^* \in T(i,s)$.

Definition (5):

A history is *secure* if all its states are secure.*

Definition (6):

A system is *secure* if each of its histories is secure.

We derived this subset by excluding definitions from the full MMS model, but intentionally did not make all of the simplifications this might have allowed. (For example, we left s as a one-tuple, instead of talking directly in terms of E. This leaves us hooks for extending s to a two- or three-tuple. Such extensions would be necessary to extend this work to cover the full MMS model in the future.)

Ideally, the security model would be formulated as the result of analysis of the application requirements. The client would review an informal version of the model and come to an understanding with the model specifier. The specifier would formalize this understanding to produce a description such as the one above.

7. Preliminary Design Decisions

We assume that in addition to the security requirements we have also have a set of functional requirements for the application. Any system we construct that conforms to all the constraints set down in the requirements

*The MMS model adds a further requirement that the system transform be transform secure. It then defines a number of properties that require the system transform to be taken into account instead of just the states. With some effort it should be possible to extend the example described here to cover most if not all of the MMS model.

ought to be an acceptable system.

With this in hand, we move on to decomposing the system into modules, the work assignments for the programmers or programming teams that will build the system. In order to separate concerns and help gain control over the assurance task, we first decide to divide the system into two parts. One part, called the Entity Monitor or *EM*, assumes the main responsibility for enforcing the security model. The remainder does everything else.

The *EM* encapsulates the system state so far as the security relevant parts of the state are concerned. It's interface provides a structure that can be mapped meaningfully to the security model. With such a mapping the security model can determine what behavior the *EM* module should allow. If the *EM* can be proven to behave as it should, then it should follow that the entire system behaves securely.

8. Generic Module Interface Structure

In order to describe the *EM* precisely, it helps to have a formal foundation for describing a module in general. In this section we consider a generic structure for a software module interface. For now we will ignore the physical appearance of the module specifications and instead focus on the conceptual structure. If we are to rigorously show a correspondence to the formal model, this structure must describe a sufficiently formal mathematical object.

We characterize a program module in our system as a five-tuple consisting of the following:

TYPES — a set of named types (sets) recognized by the module.

STATES — a set of states the module may be in before and after calls to the module. We characterize the states by specifying a set of atomic predicates on those states. These predicates partition the states into equivalence classes based upon the truth of the predicates. We consider two states s_1, s_2 equivalent, if for each atomic predicate P it is the case that $P(s_1)=P(s_2)$. When two states are equivalent, we cease to distinguish between them and say they are the same state.

CALLS — a set of calls that may be made to the module. Each call consists of an access program name and a list of argument values. A set of access program names is specified. For each access program name F a signature is specified which determines the number of parameter positions k_F and a specific type $t_i \in$ TYPES for each parameter position $1 \le i \le k_F$.

SPEC — a function that maps CALLS into STATES x STATES. Intuitively, this function captures the "meaning" of the call.

INIT — a distinguished element of STATES. This element is intended to denote an initial state for the module upon module initialization.

This structure is close in spirit to that used in several conventional specification techniques for software modules.

9. Entity Monitor Specification

Now we consider the specification of the *EM* module in particular. We are concerned with what the *EM* module will do and how to distill that information from the tabular form in which the specification is written. To do this we describe the form of the *EM* specification and how it relates to the mathematical framework we just described for modules.

Intuitively, the *EM* provides the following capabilities. It allows users of the module to create and destroy entities. The clients of the module will indicate entities by providing a *name* when the entity is created that can be used to refer to it later. Given two names to entities clients can determine if one *contains* the other. Entities are also associated with classifications. One entity is said to *dominate* another if its classification is higher than that of the other.

We specify an abstract interface for the *EM* in a collection of tables. This specification format is designed to make the specification a useful reference. A user of the specification with a specific question should be able to find the answer to that question quickly and easily. The formats used here were developed for moderately large programming efforts and have been used with reported success in several projects [3,4] as well as the SMMS project. Motivations for this type of organization are discussed in [3]. Details of the organization we use here are described in [5].

This form of module specification will be used for different purposes by different people. Designers use the form to specify the facilities provided by the module. Programmers code the module from the specification. The test group specifies module level tests from it. It also forms the basis for performing specification level proofs that the module meets the security responsibilities it is assigned.

Access Program Table

The access program table in *Figure 1* gives the names of the access programs, their parameters, the types of parameters, whether a parameter is for input or output, and what exceptions are detected and reported by the module.

Figure 1. Access Program Table

Program	Parameters	Description	Exceptions
+NewEntity+	p1:level:I p2:ent:O	classif new entity	%(no space)%
+ExistsEnt+	p1:ent:I p2:bool:O	entity exists	
+DestroyEnt+	p1:ent:I	entity name	%(no entity p1)%
+Classif+	p1:ent:I p2:level:O	entity classif	%(no entity p1)%
+GetSubEnt+	p1:ent:I p2:str:I p3:ent:O	parent index child	%(no entity p1)% %(no index)%
+SetSubEnt+	p1:ent:I p2:str:I p3:ent:I	parent index new child	%(no entity p1)% %(no index p1 p2)% %(no entity p3)%
+Init+	none		

From this table, we can construct a number of formal objects to make our discussion of the specification more precise.

We write *CALLS* to denote the set of well-formed access program calls to the module. Each call $c \in CALLS$ is of the form $F(p_1, \ldots, p_k)$ where F is program name from the column labeled "Program" and $p_1, \ldots, p_k$ are each values of the type specified in the corresponding row from the column labeled "Parameters". Also, $p_1, \ldots, p_k$ indicate values only for the input parameters -- those indicated by ":I". For example, if $a \in ent$ then $+Classif+(a) \in CALLS$.

We write $ENAMES(c)$ to denote the set of exception names listed in the entries appearing in the column labeled "Exceptions" and in the row containing the program name F. If the exception name is parameterized, we substitute the same value as appears for that parameter in the call. Extending the above example, $ENAMES(+Classif+(a))=\{$ %(*no entity a*)% $\}$.

The access program table implicitly introduces some atomic predicates to characterize the module state. For each call c that has an output parameter we admit atomic predicates of the form $c[s]=x$, where x denotes a value of the appropriate type. The appropriate type is that given for the output parameter to the call, indicated in the table by ":O". Further extending our example, the above table admits, among others, atomic predicates of the form $+Classif+(a)[s]=l$ where $l \in level$.

Local Type Dictionary

The local type dictionary (*Figure 2*) describes the names of the data types and the sets of allowable values for those types recognized by the module. In addition we also provide information on the intended use for values of the given type.

Figure 2. Local Type Dictionary

ent	a cardinal number representing a handle to an entity
level	a security level; these levels are ordered with a relation $\geq$ which is transitive and reflexive.
boolean	enumerated: *TRUE* representing truth *FALSE* representing falsity

By specifying the sets of values for each type, *Figure 2* contributes to the specification of the atomic predicates. It also contributes to their definition by indicating additional operations to construct expressions on values of these types. For example, those calls that have security level as their output may now be compared using $\geq$.

Effects Table

The effects table in *Figure 3* describes the normal case effects of the calls that are defined by the access program table. The effects are described as a relation (a set of ordered pairs) on module states. If (s,s') is in this relation it means that when a program call is made in state s, then the module may be in state s' immediately after the call assuming no exception occurs. We describe later on what happens if an exception does occur. Notice that we do not require this relation to be a function.

We can describe the effects relation with a set of predicates on state pairs. The predicates on state pairs are made up of predicates on single states. If $F(p_1, \ldots, p_k)$ is a call, we write $F(p_1, \ldots, p_k)[s]=X$ to mean that if the call is made is state s then the output of the call will be X. For example, $+Exists+(p2)[s]=TRUE$ is a predicate on states. It holds for any state s such that the output of the call $+Exists+(p2)$ made in state s is the boolean constant *TRUE*.

Figure 3. Effects Table

+Init+	$\mathbf{A}x.+ExistsEnt+(x)[s']=FALSE$
+NewEntity+	$+Exists+(p2)[s']=TRUE$ $+Classif+(p2)[s']=p_1$
+DestroyEnt+	$+Exists+(p1)[s']=FALSE$
+SetSubEnt+	$+GetSubEnt+(p1,p2)[s']=p3$

We could write $EFFECT(c)$ to mean the set of state pairs satisfying the conjunction of the predicates listed in the effects table for the program name used in the program call c. For example,

$$EFFECT(+DestroyEnt+(x))=\{(s,s') \mid +ExistsEnt+(x)[s']=FALSE\}$$

Notice that the above relation does not place any constraint on what any access program other than $+ExistsEnt+(x)$ returns in state s'. We could augment the effects table with a lot of extra assertions that describe what aspects of the state remain unchanged. However, this is such a commonly desired constraint that we adopt the convention that any call whose values in state s' are not explicitly constrained are implicitly constrainted to be the same as in state s.

Following this convention, $EFFECT(+DestroyEnt+(x))$ would be the set of state pairs (s,s') satisfying the following:

$$+ExistsEnt+(x)[s']=FALSE$$

$$\mathbf{A}y \neq x.+ExistsEnt+(y)[s']=+ExistsEnt+(y)[s]$$

$$\mathbf{A}y.+Classif+(y)[s']=+Classif+(y)[s]$$

$$\mathbf{A}y,i.+GetSubEnt+(y,i)[s']=+GetSubEnt+(y,i)[s]$$

It is easy to see that this implicit constraints convention reduces the amount of text in the effects table. This convention requires that if we really do not care what the output of an access program is between states then we must say so explicitly. The equations above say that the classification of the entity destroyed remains unchanged. If we really don't care what the classification of the destroyed entity is, we have to say more.

To do this we can add the entry $+Classif+(x)[s']=$ ***undefined*** to the effects table. Here ***undefined*** is not a constant, but an indication that the list of equations should be weakened slightly so that $+Classif+(x)[s']$ is unconstrained. If we did this, the four equations above would be the same except that we would weaken the third changing the $\mathbf{A}y$ to $\mathbf{A}y \neq x$.

It is possible for the effects table to introduce new atomic predicates on the system state. In this particular example no new atomic predicates were introduced.

Dictionary

The dictionary (*Figure 4*) contains definitions of terms used in the specification. This dictionary allows the specification to use a short term in place of a longer term.

Figure 4. Dictionary

$ex(x)$	$+Exists+(x)=TRUE$
$ex(x,y)$	$ex(x)\&ex(y)$
$co(x,y)$	$\mathbf{E}i.+GetSubEnt+(x,i)=y$
$do(x,y)$	$+Classif+(x)\geq+Classif+(y)$

Here the dictionary did not introduce new atomic predicates, but it might easily have done so.

Exception Dictionary

Figure 5 lists the exceptions that may occur in using the access programs listed in the access program table. Each exception name is mapped to a predicate that describes the states in which the exception may occur. The predicates are stated as formally as possible.

Figure 5. Exception Dictionary

$^{c}\%$(no space)$^{c}\%$	There is not enough available storage to carry out the operation.
$^{c}\%$(no entity x)$^{c}\%$	*not* $ex(x)$
$^{c}\%$(no index x i)$^{c}\%$	*not* $\mathbf{E}y.+GetSubEnt+(x,i)=y$

Now that we have introduced the exception dictionary, we can start to talk about the exceptional case behavior of the module. Our basic exception handling policy is that if an access program is called under an exceptional condition, then the module should report the exception but not change state in any other way.

We can state this more formally by defining a relation. Given an exception name e we write $DICT(e)$ to denote the predicate associated with e in the exception dictionary. We define the exception relation for a program call c as follows

$$ER(c)=\{(s,s) \mid \mathbf{E}e \in ENAMES(c).DICT(e)[s]\}$$

This definition provides a relation on states meeting our intended goal that no change in state occurs if any of the exceptional conditions hold at the time of the call.

Notice that $dom\ ER(c)$ includes no state in which an exceptional condition does not occur.

The exception dictionary can introduce new atomic predicates on the system state. We allow predicates to be stated in natural language. In this example the statement "there is not enough storage to carry out the operation" is introduced as an atomic predicate.

10. Meaning of the Specification

Now we can combine the normal case effects and the exceptional case effects in a single relation describing the overall semantics of a call. Define

$$SPEC(c)=\{(s,s') \mid (s\in domER(c) \ \&\ s'=s)$$

$$or\ (not\ s\in domER(c)\ \&\ (s,s')\in EFFECT(c))\}.$$

This definition captures our intent that exception checking take priority over the normal case effects. The first clause says that if an exception condition occurs, then the state does not change. The second clause says that if no exception conditions are met, then the normal case effects occur.

Given a predicate P defined on module states we say that P is *invariant* if and only if $\mathbf{A}s.P(s)\rightarrow P(SPEC(c)[s])$.

The above specification captures the normal case behavior we want from our *EM* module. We have only tried to capture the normal case behavior without any special security checking. We will derive the security checking a little later on Before we attempt to add security constraints, we need to be more specific about what it means for the module to be secure. In order to do that, we need to define an interpretation.

11. Interpretation of the Security Model

An interpretation relates the structure defined by our module specification to the structure defined in the security model. The security model gives a very precise definition of what it means for Σ to be secure. We don't yet have a good definition of what it means for *EM* to be secure. If we map the structure of the *EM* to the structure of a specific Σ we can avail ourselves of the precise definition of security so rigorously developed for the latter. We will describe this construction by defining an interpretation function **I**. We then say that *EM* is secure if $\Sigma=\mathbf{I}[EM]$ is secure.

How do we go about making such a construction? To a large extent, we must rely on our intuition. This is not so bad really, because the structures we defined for the *EM* did have an intended interpretation. This intent guides us in defining a specific interpretation. But we are not left wholly to intuition. There are some formal criteria that an interpretation should meet.

Two formal properties desired of an interpretation are that it be total and that it be a function.** By *total* we mean that every state or state transformation in the *EM* specification should map to some state or state transformation in the model. Any *EM* state transformation that does not map to some state transformation in the model holds the potential to cause behavior that the security model cannot constrain. The interpretation is a *function* if it maps the *EM* module to exactly one Σ. If **I** mapped *EM* to more than one system Σ our argument might falter by proving only one of the possible interpretations secure while allowing an insecure interpretation.

We begin by defining **I** mapping *ent* to *RF*, *CALLS* to *I*, *STATES* to *S*, *SPEC* to *T*, and *INIT* to s_0. The details are given in *Figure 6*.

Figure 6. Interpretation of Security Model

1. $\mathbf{I}[n] = n$, if $n\in ent$
2. $\mathbf{I}[c] = c$, if $c\in CALLS$
3. $\mathbf{I}[co(m,n)] = E(\mathbf{I}[n])\in H(CE(\mathbf{I}[m]))$
4. $\mathbf{I}[do(m,n)] = CE(E(\mathbf{I}[n]))\leq CE(E(\mathbf{I}[m]))$
5. $\mathbf{I}[ex(m)] = \mathbf{I}[m]\in dom(E)$
6. $\mathbf{I}[P\ \&\ Q] = \mathbf{I}[P]\ \&\ \mathbf{I}[Q]$
7. $\mathbf{I}[P\ or\ Q] = \mathbf{I}[P]\ or\ \mathbf{I}[Q]$
8. $\mathbf{I}[not\ P] = not\ \mathbf{I}[P]$
9. $\mathbf{I}[P\rightarrow Q] = \mathbf{I}[P]\rightarrow\mathbf{I}[Q]$
10. $\mathbf{I}[\mathbf{A}x.P(x)] = \mathbf{A}x.\mathbf{I}[P(x)]$
11. $\mathbf{I}[\mathbf{E}x.P(x)] = \mathbf{E}x.\mathbf{I}[P(x)]$
12. $\mathbf{I}[SPEC(c)[s]] = T(\mathbf{I}[c],\mathbf{I}[s])$
13. $\mathbf{I}[INIT] = s_0$
14. $\mathbf{I}[EM] = \Sigma(\mathbf{I}[CALLS],\mathbf{I}[STATES],\mathbf{I}[INIT],\mathbf{I}[SPEC])$

Lines 1-2 show that ent's and calls are interpreted as themselves. Lines 3-5 show how states are interpreted based on the form of the atomic predicates that describe them. Lines 6-11 tell how states described by more complicated formulas are interpreted by preserving the structure of the formula and replacing the parts of the atomic predicates with their interpretations. Lines 12-14 relate the meaning of the overall specification to a security model system.

Notice that the mapping takes the specification to the model. Our first attempts tried to map the security model to the specification. Informally this approach seemed reasonable, but great difficulties arose when we attempted a formal mapping. For example, the mapping E that maps references to entities has no direct correlate in the specification of the interface. At the interface, entities are referred to only by their names, never as entire objects. Although an implementation of the module might include a table mapping names to some representation of an entity such a table is not part of the specification. We refused to make that structure a part of the interface because the interface would then reveal implementation decisions it was designed to hide. The solution was to reverse direction and map the specification to the security model. In retrospect this should have been obvious. What was needed was a means to reduce statements about the specification (which so far had no definition of security) to statements about the model system (where security was well-defined). Thus the mapping had to go the direction it did.

12. Security Theorem

Having defined a security model, a specification, and an interpretation, we are finally ready to tackle the question of security. Is the *EM* specification secure under the given interpretation? We want to demonstrate that when we use the *EM* specification structure we have defined to construct a particular Σ according to the construction defined by **I** we are guaranteed the resulting Σ is secure. Formally, is $\mathbf{I}[EM]$ secure?

We will tackle this question by trying to prove a theorem. Since we have yet to consider security checks in the *EM* specification we would be quite surprised if the specification could be proved secure without modification. We expect the current specification to be underconstrained. However, as we attempt to derive our proof we will uncover the additional constraints necessary to make the module secure. Then, after we know what the constrains are and we know that they are adequate, we can factor them into the specification.

The following proposition states that the *EM* design is secure. In attempting to prove the proposition we discover that when we assume a particular lemma (*HierInv*) the proof goes through easily. The proof reduces our problem to one of proving a lemma expressed solely in terms of the module interface functions. All of the structure of the security model simplifies away and we can safely forget about it for the remainder of the proof.

Proposition: $\mathbf{I}[EM]$ is secure

Proof:

Applying the interpretation

$$\mathbf{I}[EM]=\Sigma[I=\mathbf{I}[CALLS],s=\mathbf{I}[STATES],s_0=\mathbf{I}[INIT],T=\mathbf{I}[SPEC]]$$

By definition Σ is secure if all of its histories are secure. We will show that all the histories are secure by induction on the size of the history.

(Basis) Need to show $\Pi(0)$ contains only secure states. $\Pi(0)$ contains only one state $s_0=\mathbf{I}[EM.INIT]$. From the *EM* specification we know

$$[\mathbf{A}m.\ not\ ex(m)][EM.INIT]$$

Applying the interpretation function, we get:

$$\mathbf{I}[\mathbf{A}m.not\ ex(m)][EM.INIT]$$

$$=[\mathbf{A}m.not\ \mathbf{I}[ex(m)]][\mathbf{I}[EM.INIT]]$$

$$=[\mathbf{A}m.not\ m\in dom(E)][s_0]$$

Hence $[dom(E)=\emptyset][s_0]$ which implies $[rng(E)=\emptyset][s_0]$. We observe that substituting $\emptyset$ for $rng(E)$ in definition (2) gives us a predicate of the form $\mathbf{A}x\in\emptyset.P(x)$. Since this predicate is vacuously true s_0 is secure. Thus, $\Pi(0)$ contains only secure states.

**I am grateful to Virgil Gligor of the University of Maryland for pointing out these criteria.

(Induction) Assume $\Pi(0 \le i \le N)$ contains only secure states. Hence in $\Pi(N)=(-,s_N)$, we know s_N is secure. $\Pi(N+1)=(c,s_{N+1})$, where $s_{N+1} \in T(c,s_N)$. If we can show that $\mathbf{A}c\,\mathbf{A}s' \in T(c,s_N).\ s'$ is secure then the rest is easy. We assume the following lemma.

Lemma HierInv.

$$(HIER: \mathbf{A}m,n.ex(m)\&ex(n).co(m,n) \rightarrow do(m,n))\ \textit{is invariant}$$

Applying $\mathbf{I}$ to the lemma we obtain

$$\mathbf{I}[\mathbf{A}c.HIER[s] \rightarrow \mathbf{A}s' \in SPEC(c)[s].HIER[s']]$$

$$= \mathbf{A}c.\mathbf{I}[HIER]\mathbf{I}[s] \rightarrow \mathbf{A}s' \in \mathbf{I}[SPEC(c)[s]].\mathbf{I}[HIER][s'] \qquad (*)$$

$$= \mathbf{A}c.\,\mathbf{I}[HIER]\mathbf{I}[s] \rightarrow \mathbf{A}s' \in T(\mathbf{I}[c],\mathbf{I}[s]).\mathbf{I}[HIER][s']$$

Now, focusing on the interpretation of HIER,

$$\mathbf{I}[HIER] = \mathbf{A}m,n.\mathbf{I}[ex(m)]\,\&\,\mathbf{I}[ex(n)].\mathbf{I}[co(m,n)] \rightarrow \mathbf{I}[do(m,n)]$$

$$= \mathbf{A}m,n \in dom(E).E(n) \in H(E(m)) \rightarrow CE(E(n)) \le CE(E(m))$$

$$= \mathbf{A}x,y \in rng(E).x \in H(y) \rightarrow CE(x) \le CE(y)$$

which is exactly the definition of secure state. Thus line (*) can be rewritten as:

$$\mathbf{A}c.(\mathbf{I}[s]\ \textit{is secure}) \rightarrow (\mathbf{A}s' \in T(c,\mathbf{I}[s]).s'\ \textit{is secure})$$

If we choose $s_N = \mathbf{I}[s]$ in the above we obtain

$$\mathbf{A}c.(s_N\ \textit{is secure}) \rightarrow (\mathbf{A}s' \in T(c,s_N).s'\ \textit{is secure})$$

But we know s_N is secure so

$$\mathbf{A}c.\mathbf{A}s' \in T(c,s_N).\ s'\ \textit{is secure}$$

Since $s_{N+1} \in T(c,s_N)$ it follows that

$$s_{N+1}\ \textit{is secure}$$

From which we conclude that $\Pi(0 \le i \le N+1)$ contains only secure states. By induction Π includes only secure states, hence Π is secure, hence $\Sigma = \mathbf{I}[EM]$ is secure. QED

Now we need to justify the Lemma. In order to do that we will have to add additional constraints to the specification. Exactly what constraints will become apparent when we try to prove the lemma.

13. Security Invariants

Since the security responsibilities of the module are so important it makes sense to reserve a place in the module interface specification to record them. We include in our module interface documentation a table (see *Figure 7*) that lists the security invariants that the module must maintain. In this example there is only the single invariant *HIER* identified earlier. A more complex example would include a much longer list.

Figure 7. Security Invariants Table

HIER	$\mathbf{A}x,y.ex(x,y) \rightarrow co(x,y) \rightarrow do(x,y)$

The security invariants listed are intended to follow from the design decisions specified in the earlier tables. To the programmers these invariants are redundant information since the entire module interface has already been specified in the earlier sections. They are included mainly so that reviewers will know to check that the intended properties actually follow from the interface specification. If it can be shown that they do not, then the module specification is inconsistent with its security responsibilities.

The properties in this section should be proven to hold for the specification. What constitutes a proof may vary from a thorough reading by conscientious reviewers to a formal mathematical proof. What is required is that the proof present a clear and compelling chain of reasoning linking what is to be proven to a set of acceptable assumptions.

It is especially important that the specifier of the module attempt to prove that the security invariants hold, for it is close attention to these invariants that generate the many mini-decisions that are recorded in the tables that make up the specifications. Unless the specifier has done so, it is unlikely that the specification will describe anything with approximately the desired properties.

In the following section, we attempt to prove that the property in *Figure 7* holds for the specification. We say "attempt to prove" because errors and omissions will be found. When they are, we take note of what changes must be made to the module specification to make the proof go through. At the end of the exercise, we present an amended specification correcting the design flaws uncovered.

14. Calculating Weakest Preconditions

The techniques for calculating weakest preconditions developed by Dijkstra[1] and further elaborated by Gries[2] give us useful tools to determine whether our specification meets the security constraints we have defined. This technique is applicable when constraints can be described as invariants of the module state and state transformations can be described as textual manipulations of predicates.

The concept of a weakest precondition is central to this calculus. Given a program S and a predicate R the weakest precondition of S with respect to R, written $wp(S,R)$, is the set of states such that initiating S in one of those states will result in termination of S in a state satisfying R. In the context of our module interface specification we treat each module call as a program. When S is $c \in CALLS$, we observe that $wp(c,R) = \{s \mid R(SPEC(c)[s])\}$.

Now, lets look at our problem again. We have a predicate *HIER* that we want to keep invariant. We have programs that change the system state. We now want to know, under what conditions will these programs change the state in a way that *HIER* remains invariant?

If we can calculate $wp(c,HIER)$ then we can check that the *EFFECTS* transitions of our programs are only taken in states satisfying $wp(c,HIER)$. In states not satisfying $wp(c,HIER)$ we must be certain appropriate exceptions are defined in the specification to ensure that an *ER* transition is taken instead. If not, we will either provide them or alter the *EFFECTS* semantics to guarantee this is the case. Thus, ***by construction***, the desired invariant is maintained. This is the approach we will take to ensuring that *HIER* is invariant over all the calls to the *EM*.

In order to apply these techniques, we must recast the effects of the access programs in terms of *predicate transformers*. A predicate transformer is simply a rule for rewriting a predicate. As such it maps predicates to predicates. When a predicate is used to indicate a set of states (such as we are doing) the transformer also induces a mapping from sets of states to sets of states.

To help specify our predicate transformers we introduce some notation for substitution. We use substitution to reason about state changes. Since we characterize states with formulas built up from atomic predicates, we focus on substitutions involving the atomic predicates. When we write predicates as formulas that include quantifiers, substitution becomes more difficult to define because of the presence of bound variables. We avoid these complications by considering quantifiers as abbreviations for sequences of conjunctions or disjunctions. For example,

$$\mathbf{A}x \in T.P(x) = P(t_1)\ \&\ P(t_2)\ \&\ \cdots$$

where $T=\{t_1,t_2,\cdots\}$. We call this the ***expanded form*** of the formula.

Given a predicate R, we write R^{x}_{E} to mean the result of rewriting R in its expanded form and then substituting E for each occurrence of the atomic predicate x. In our formulas, the types of the variables are usually determined from context so we can write $\mathbf{A}x.P(x)$ as short for $\mathbf{A}x \in T.P(x)$.

It may appear that large formulas would cause problems, but in practice we don't need to write out a long expansion. Since we know what substitution we want to perform it suffices to factor out the term we are substituting for so that it appears outside the scope of any quantifiers.

$$(\mathbf{A}x.ex(x))^{ex(n)}_{true} = (ex(n)\ \&\ (\mathbf{A}x{\neq}n.ex(x)))^{ex(n)}_{true}$$

$$= ex(n)^{ex(n)}_{true}\ \&\ (\mathbf{A}x{\neq}n.ex(x))^{ex(n)}_{true}$$

$$= true\ \&\ (\mathbf{A}x{\neq}n.ex(x))$$

$$= \mathbf{A}x{\neq}n.ex(x)$$

The key observation is that $ex(n)$ does not appear at all in the expansion of $\mathbf{A}x{\neq}n.ex(x)$. This trick is used repeatedly in the examples that follow.†

We claim that the relevant parts of the effects transitions can be formulated with the following predicate transformers:

$$wp(+DestroyEnt+(n),R)=R^{ex(n)}_{false}$$

$$wp(+NewEntity+(n),R)=R^{ex(n)}_{true}$$

$$wp(+SetSubEnt+(m,n,i),R)=R^{co(m,n)}_{true}.$$

At this point the reader should compare the predicate transformers described above with the effects table specifications. The predicate transformers are not equivalent to the specification. The transformers are, however, consistent with the specification. They describe a slightly more abstract, weaker specification of the behavior focusing on the security relevant properties.

In earlier reviews of this work, some have stated that this intuitive leap calls into question the validity of what is advertised as a rigorous methodology. I must confess some discomfort at this step myself, but still choose this path because it helps solve the engineering problem at hand. Faced with a system to be built, a set of properties to be proved, and finite resources to be applied, this approach lets us get on with analyzing the system and generating the needed insight to specify it in conformance with the security model. It can be argued that in an engineering setting the utility of applying formal methods is not measured by the completeness of rigorous analysis, but the quality of the insights that the analysis yields. Had we devoted enough effort to push through the rigor, we might not have gotten the insights we did.

Still, we would like to have a more formal basis for transforming the relational form of the specification into a predicate transformer. In our attempts to derive such a basis, we were able to prove the following theorem: if $SPEC(c)=\{(s,s')\,|\,R[s']=R^{x}_{E}[s]\}$ then $wp(c,R)=R^{x}_{E}$. This suggests that with more fundamental work the gap in rigor can be bridged. pp

15. Proof of the Security Invariant

In this section we consider the proof that *HIER* is invariant. Our strategy is to show $HIER \rightarrow wp(c,HIER)$ for each call $c \in CALLS$.

Note that given a specific n we can factor *HIER* into $H1\ \&\ H2\ \&\ H3\ \&\ H4$ where

$$H1\colon \mathbf{A}x{\neq}n\,\mathbf{A}y{\neq}n.ex(x,y){\rightarrow}co(x,y){\rightarrow}do(x,y)$$

$$H2\colon \mathbf{A}x{\neq}n.ex(x,n){\rightarrow}co(x,n){\rightarrow}do(x,n)$$

$$H3\colon \mathbf{A}y{\neq}n.ex(n,y){\rightarrow}co(n,y){\rightarrow}do(n,y)$$

$$H4\colon ex(n){\rightarrow}co(n,n){\rightarrow}do(n,n)$$

From the fact that $\leq$ is reflexive, we can determine that $do(n,n)$ reduces to true. Hence, the $H4$ term goes away, and we are left with $HIER = H1\ \&\ H2\ \&\ H3$. This completes our preliminary observations. We are now ready to prove the security invariant.

† There are alternative solutions to this kind of problem, but the ones we explored all required us to express transformers in terms of the implementation variables. On principle, we do not want our specifications to prejudice the programmers toward any specific implementation. Hence we took pains to describe the state only in terms of abstract atomic predicates instead of program variables. For examples using program variables see the treatment of array objects in [6] and [7].

Assume *HIER*. We now need to show $wp(c,HIER)$ We perform a case analysis on the form of the call c. There are just three cases.

case (i) $c=+NewEntity+(n)$

Here we must calculate $wp(+NewEntity+(n),HIER)$. To do so we factor *HIER* as above and distribute wp over each of the factors.

$$wp(+NewEntity+(n),HIER) = H1^{ex(n)}_{true}\ \&\ H2^{ex(n)}_{true}\ \&\ H3^{ex(n)}_{true}$$

Performing the substitutions on each of the H terms we obtain

$$H1^{ex(n)}_{true} = H1$$

$$H2^{ex(n)}_{true} = (P2\colon \mathbf{A}x{\neq}n.ex(x){\rightarrow}co(x,n){\rightarrow}do(x,n))$$

$$H3^{ex(n)}_{true} = (P3\colon \mathbf{A}y{\neq}n.ex(y){\rightarrow}co(n,y){\rightarrow}do(n,y))$$

Notice that $HIER \rightarrow H1$. Since, HIER holds by assumption, we conclude $H1$ also holds.

At this point we get stuck. We can't simplify away $P2$ and $P3$. With a little reflection, it is apparent that the reason $P2$ does not simplify is that the specification admits state transitions where an entity comes into existence already contained by some other entity. Similarly $P3$ is the result of allowing the entity to come into existence already containing other entities.

The rigors of proof have uncovered a subtle omission. A less careful argument might easily have missed this problem. To produce a formal argument we are forced to make such assumptions an explicit part of the specification.

A sensible way to ensure that $P2$ and $P3$ hold in the result state is to further constrain the specification so that when an entity is created, it contains no other entities. This is a new design decision. We document this design decision by adding the following constraint to the effects table for $+NewEntity+$.

$$\mathbf{A}x\colon not\ co(p1,x)[s'] \ \&\ not\ co(x,p1)[s'] \qquad (E1)$$

This change alters the transformer for $+NewEntity+$.

$$wp(+NewEntity+(n),R)=R^{ex(n),co(n,y),co(x,n)}_{true,false,false}$$

With this additional constraint $P2$ and $P3$ simplify away to *true*.

$$H2^{ex(n),co(n,y),co(x,n)}_{true,false,false} = (\mathbf{A}x{\neq}n.ex(x){\rightarrow}false{\rightarrow}do(x,n)) = true$$

$$H3^{ex(n),co(n,y),co(x,n)}_{true,false,false} = (\mathbf{A}y{\neq}n.ex(y){\rightarrow}false{\rightarrow}do(n,y)) = true$$

Using the modified specification of $+NewEntity+$ we can now conclude that $wp(+NewEntity+(n),HIER)=true$.

case (ii) $c=+DestroyEnt+(n)$

Calculating $wp(+DestroyEnt+(n),HIER)$ as before we obtain

$$H1^{ex(n)}_{false} = H1$$

$$H2^{ex(n)}_{false} = (\mathbf{A}x{\neq}n.false{\rightarrow}co(x,n){\rightarrow}do(x,n)) = true$$

$$H3^{ex(n)}_{false} = (\mathbf{A}y{\neq}n.false{\rightarrow}co(n,y){\rightarrow}co(n,y)) = true$$

Since $HIER \rightarrow H1$ as before, we conclude $H1=true$. Hence $wp(+DestroyEnt+(n),HIER)=true$.

case (iii) $c=+SetSubEnt+(n,m)$

Here $wp(+SetSubEnt+(n,m),HIER)=HIER^{co(m,n)}_{true}$. We can apply a similar factorization technique as above, but in this case, we use $HIER=H1\ \&\ H2$ where

$$H1 = \mathbf{A}(x,y)\neq(m,n)\ .\ ex(x,y)\rightarrow co(x,y)\rightarrow do(x,y)$$

$$H2 = ex(m,n)\rightarrow co(m,n)\rightarrow do(m,n)$$

Applying the predicate transformer to the terms we get

$$H1^{co(m,n)}_{true} = H1$$

$$H2^{co(m,n)}_{true} = ex(m,n)\rightarrow do(m,n)$$

We can ensure that the above condition is met by adding a new exception %(*Hierr*)% to *ENAMES*(+*SetSubEnt*+) and defining *DICT*(%(*Hierr*)%) = *not do*(*p*1,*p*2).

This completes all the cases. Hence we conclude that *wp*(*c*,*HIER*)=*true* for any *EM* call *c*. QED.

The amended specification (*Figures 7,8, and 9*) is a definite improvement on the old one. It explicitly tells the programmer that entities created by calls to +NewEntity+ may neither contain nor be contained by other entities. The same assertions alert the code reviewers to check these two properties when comparing the specifications against the code. A new exceptional condition was discovered and documented for the +SetSubEnt+ call. Documenting that exception has the same benefits. In addition it alerts all of the programmers using the module to consider this exceptional case when coding each call to +SetSubEnt+. Because these omissions appeared on the module's external interface they would likely be very expensive to correct if they were not discovered until later in development.

Figure 7. Amended Access Program Table

Program	Parameters	Description	Exceptions
+NewEntity+	p1:level:I p2:ent:O	classif new entity	%(no space)%
+ExistsEnt+	p1:ent:I p2:bool:O	entity exists	
+DestroyEnt+	p1:ent:I	entity name	%(no entity p1)%
+Classif+	p1:ent:I p2:level:O	entity classif	%(no entity p1)%
+GetSubEnt+	p1:ent:I p3:str:I p3:ent:O	parent index child	%(no entity p1)% %(no index p1 p2)%
+SetSubEnt+	p1:ent:I p2:str:I p3:ent:I	parent index new child	%(no entity p1)% %(no index p1 p2)% %(no entity p3)% %(Hierr p1 p2)%
+Init+	none		

Figure 8. Amended Effects Table

+Init+	$\mathbf{A}x.+ExistsEnt+(x)[s']=FALSE$
+NewEntity+	$+Exists+(p2)[s']=TRUE$ $+Classif+(p2)[s']=p_1$ $\mathbf{A}x.not\ \mathbf{E}i.+GetSubEnt+(x,i)=p2$ $\mathbf{A}x.not\ \mathbf{E}i.+GetSubEnt+(p2,i)=x$
+DestroyEnt+	$+Exists+(p2)[s']=FALSE$
+SetSubEnt+	$+GetSubEnt+(p1,p2)[s']=p3$

Figure 9. Amended Exception Dictionary

%(no space)%	There is not enough available storage to carry out the operation.
%(no entity *x*)%	*not ex*(*x*)
%(no index *x i*)%	*not* $\mathbf{E}y.+GetSubEnt+(x,i)=y$
%(Hierr *x y*)%	*not Classif*(*x*)≥*Classif*(*y*)

16. Extensions

The simple hierarchy security property described here is only a small part of the security model for the SMMS. There are many extensions that can be made to this small example to illustrate design problems faced in the SMMS design. The techniques and basic framework described here should suffice to handle a number of them. Examples of the extensions required are:

Reclassification. In the SMMS users with appropriate authorization are allowed to reclassify entities. The *EM* specification can be extended with a +SetClassif+ command that alters the classification of an entity. When the predicate transformer for +SetClassif+ is specified and applied to *HIER* the appropriate exceptional conditions for +SetClassif+ can be derived. (In writing the SMMS specification we actually overlooked one of these exceptional conditions and discovered the omission only when we attempted to derive these conditions formally.)

ICL Operations. In the SMMS design the *EM* operations are not invoked directly by arbitrary programs in the system, but only by special programs called ICL operations. The ICL operations correspond closely to the commands visible at the human interface. In addition ICL operations are associated with an access set that determines what individual users may perform that command. The system is architected so that ICL operations can only be invoked via the *EM* interface. Thus the *EM* provides a set of access programs that allow a user to invoke ICL operations on particular entities. The security model can be extended with assertions to formalize this access policy. After formalizing the new *EM* operations as predicate transformers we can derive appropriate exceptional conditions.

Information Flow. In the SMMS security model entities are associated with values and information flow between these values is constrained. This is a difficult problem but some clever constructions may enable the techniques described here to handle it. The formalization of information flow described in[3] is based on system histories involving a sequence of states and is not expressed as a state invariant. In principle, any information about the history could be cast as a state invariant by recording what need be known about any past state in an atomic predicate on an individual state. How such a construction works in practice can only be determined by experience.

17. Discussion

Altering the specifications during the proof may appear suspect at first, but it is a quite practical approach. It gives us a systematic technique for deriving the additional constraints our specification needs to be secure. In addition, it provides a proof by construction of the sufficiency of these constraints.

Notice that the final specifications and proofs that are generated are not organized in the order in which we make the decisions. Instead they are organized as reference documents with extensive use of tables and indices so that facts may be filed and retrieved easily. Such an organization is necessary to cope with the inevitable iteration on the design. In the end, all of the constraints we derive are filed away in the specification. Similarly, any security proof documentation that results should be organized as a reference document describing the modified proof. It should incorporate all the refinements to the proof but not record of all the intermediate stages.

The size of the proofs may concern some readers. Even for this small example the amount of reasoning to be done to produce a convincing proof is

sizable. I do not think that introducing mathematical notation increases the amount of reasoning necessary. It simply confines the reasoning to a smaller and better defined set of concepts. One could attempt to provide the same reasoning in natural language alone, but it would be much more difficult to follow, harder to check and much more error prone.

In this example, the proofs were all hand proofs; no automated proofs were presented. We believe that for our system hand proofs reflect what is most useful, feasible, and cost effective given the current state of program verification. Our experience is that hand proofs were helpful in developing designs and gave useful guidance on the direction design decisions should take. Hand proof attempts uncovered subtle design errors while the design was in progress.

18. Conclusions

We have derived a specification for a software module with the aid of a security model and some formal techniques. To do so it was necessary to tackle some subtle issues. We had to define a interpretation mapping the specification to the model. The security proof provided a demonstration that the specifications derived were secure with respect to that interpretation. In the process of deriving the proof, new constraints that needed to be placed on the specifications were uncovered. These constraints were factored in to produce an amended specification.

The software engineering techniques we employed helped when it came time to reason about security properties of the design. The tabular formats and notations gave us the ability to organize the masses of detailed decisions that go into specifying module interfaces. When we needed to prove a security property for the specification, the organization enabled us to focus in on the relevant details. We could easily determine what access operations we needed to inspect and how their effects interacted with the property under investigation.

It is our hope that the experiences, approach, and examples described here help others aspiring to design trustworthy systems.

19. Acknowledgements

Most of the fundamental arguments for an application-based approach are due to Carl Landwehr. John McLean provided the formalization of the MMS security model upon which this model is based. Connie Heitmeyer also had a strong influence. Andy Moore has been active in verifying the SMMS prototype and his verification efforts first raised some of the specific semantic issues included in the example. We are grateful to Tom Taylor of the Space and Naval Warfare Systems Command and to John Campbell of the National Computer Security Center for funding this effort.

References

[1] C. Landwehr, C. Hietmeyer, and J. McLean, "A Security Model for Military Message Systems," *ACM Transactions on Computer Systems*, August 1984. Also published as NRL Report 8606, 31 May 1984.

[2] M. R. Cornwell, A. L. Evans, and J. T. Quinn, "Interface Specifications for the Entity Monitor Module," NRL Technical Memorandum 5590-76:MC, 7 March 1984.

[3] S. D. Hester, D. L. Parnas, and D. F. Utter, "Using Documentation as a Software Design Medium," *The Bell System Technical Journal*, Vol. 60, No. 8, pp. 1941-1977.

[4] P. C. Clements, "Using Information-Hiding as a Design Discipline: Techniques and Lessons," *Structured Development Forum VIII, Seattle, Washington*, August 1986.

[5] J. T. Quinn, "A Standard Organization for Specifying Abstract Interfaces for the SMMS Application," NRL Internal Memorandum, 1988.

[6] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Edgewood Cliffs, N.J., 1976.

[7] D. Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.