Essay 8

Formal Methods and Models

James G. Williams and Marshall D. Abrams

The motivation for using formal methods in the context of trusted development stems primarily from their ability to ensure precision, consistency, and added assurance during the elaboration of security requirements, across different stages in the development of a trusted computing system. The subject matter of formal models and specifications can be illustrated by the various kinds of security attributes and requirements that have turned up in published security models. Examples discussed in this essay relate to nondisclosure policies, data integrity policies, and user-controlled policies. This essay concludes with a discussion of technical and methodological issues relating to the effective use of formal methods.

While it is appealing to approach security intuitively, it is important to recognize that many "good ideas" turn out to be not so good upon close inspection. This essay is concerned with several ways to increase the probability that the trust placed in an automated information system (AIS) is justified. It discusses a set of approaches described as *formal*, including models written in a formal notation and the use of formal proofs.

In the context of information security, *formal* methods imply use of specialized language and reasoning techniques. *Informal* methods, by way of contrast, are couched in natural language and rely on common sense. The use of a formal notation, especially one with a well-understood semantics, can improve the precision with which a security policy is stated. The use of formal proofs can provide added assurance that specified policy-enforcement techniques succeed in satisfying a formal definition of security. Formal methods and proofs are required by the TCSEC for systems rated B2 and above.

By way of introduction, we first discuss the motivation for formal methods, and then we look at their application during the construction of secure systems.

Doing the "right thing"

How is one convinced that a system does the "right thing"? There could be many contributing factors, such as the reputations of the hardware and software vendors, some degree of testing, and prior experience. In the case of computer security, these factors are frequently not sufficient. The security features of a computing system are often considered to be *critical* system functions whose correctness must be assured to a high degree because the consequences of failure are unacceptable. Yet we know from experience that latent security flaws in operating system modules or in the system's underlying hardware can cause trouble after years of use.

In general, inadequacies in a system can result either from a failure to understand requirements or from flaws in their implementation. The former problem, defining what a system should do, is relatively difficult in the case of security, so the definition has to be unusually precise and well understood. Moreover, subtle flaws can show up at any spot where the system design has left a loophole, so the definition must be faithfully reflected down through all the various stages of requirements analysis, design, and implementation [GASS87, Sec. 9].

A primary strategy for assuring security is to provide a *security policy* model that simultaneously models both a security policy and an AIS. Such a model consists of two submodels: a *definition of security* that captures the security policy, and *rules of operation* that show how the AIS enforces the definition of security. In the case of a formal model, it is possible to be quite precise in stating the definition of security and to rigorously prove that the rules of operation guarantee that the definition of security will be satisfied. In some formalizations, initial-state assumptions and rules of operation are treated as axioms. In this case, the definition of security becomes a theorem of the resulting theory, in addition to defining the modeled portion of the security policy.

Stages of elaboration in the creation of a secure system. To gain a more detailed understanding of the role of formal methods, it is necessary to investigate the various stages in the development of a secure AIS, as pictured in Figure 1.

High-level policy objectives specify what is to be achieved by proper design and use of a computing system; they constrain the relationship between the system and its environment. Higher-level security objectives for the use of DoD computing systems have been given in DoD Directive 5200.28, "Security Requirements for Automated Information Systems (AISs)" [DOD88a]. Objectives relating to the disclosure of classified information stem from Executive Order 12356 [REAG82] and can be found in 5200.1-R [DOD86] and in the TCSEC "control objectives." Classified and other sensitive information needed for the conduct of federal programs is

protected by the Computer Security Act of 1987 [CONG87], which requires safeguards against loss and unauthorized modification. The collection, maintenance, use, and dissemination of personal information are protected by DoD Directive 5400.11 [DOD82, § E.2], by the Privacy Act of 1974 [CONG74], and by the Computer Matching and Privacy Protection Act [CONG88, CONG90]. Security objectives for federal systems have been enumerated by the US Congress [CONG82], Government Accounting Office [GAO88], and Office of Management and Budget [OMB82, OMB84]. Security objectives for commercial systems have been articulated by Clark and Wilson [CLAR87, CLAR89b]. Finally, site-dependent organizational security policies may also contribute objectives for the design and use of a secure computing system.

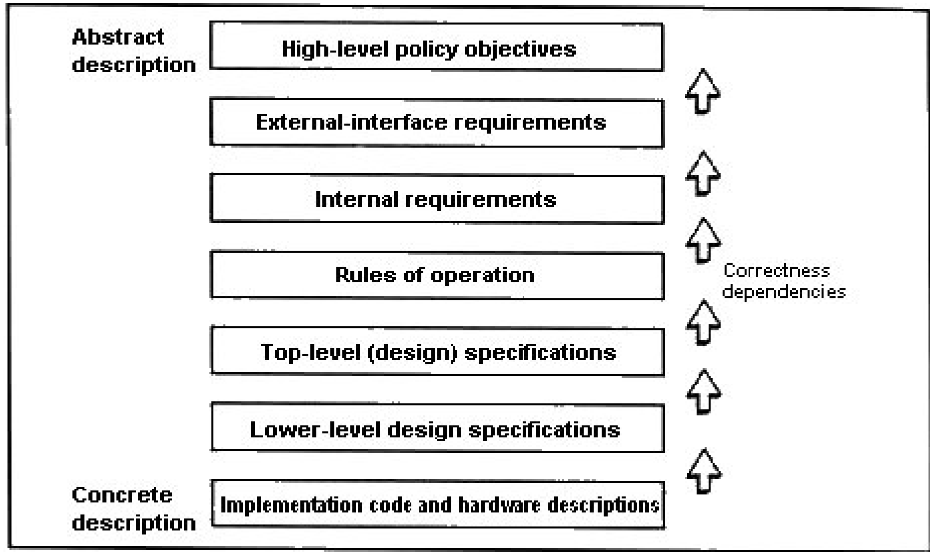

Figure 1. Stages of elaboration.

An *external-interface requirement* applies a higher-level objective to a computing system's external interface; it explains what can be expected from the system in support of the objective but does not unduly constrain internal system structure. The Goguen-Meseguer noninterference model is perhaps the best-known example of a formal external-interface requirement [GOGU82, GOGU84]. Various derivatives of this model have been advanced in support of DoD nondisclosure objectives. They say, essentially, that outputs from a system at a given nondisclosure level must not depend on inputs at a higher or incomparable level.

Internal requirements constrain relationships among system entities or components. The trusted computing base (TCB) emerges at this stage of

elaboration; it contains those portions of the system which must be constrained to support the higher-level objectives and achieve the external-interface requirements. Bell and LaPadula's simple-security property is an example of an internal requirement; it says, essentially, that a subject may have read access to an object only if the security level of the subject dominates the security level of the object [BELL76]. Formal definitions of security usually contain a combination of internal and external-interface requirements. They are given as a formalization of a *system security policy* that specifies how an AIS helps prevent information and computing resources from being used to violate a corresponding organizational security policy [STER91].

Rules of operation explain how internal requirements are enforced by specifying access checks and related behaviors that guarantee satisfaction of the internal requirements. A top-level specification (TLS), like rules of operation, specifies the behavior of system components or controlled entities, but it is a more complete functional description of the TCB interface. There is no firm boundary between rules of operation and top-level specifications. Both explain how internal requirements are enforced by the TCB, but more detail is required in a TLS, including accurate descriptions of the error messages, exceptions, and effects of the TCB interface. In designing rules of operation, it may be convenient to separate access decisions from other kinds of system functionality, including the actual establishment or removal of access. This separation facilitates the exploration of new access control policies: Often, only the access-decision portion is affected by a change in policy because access enforcement depends only on the access decision, not on how it was made [LAPA90, ABRA90]. Isolation of the access-decision mechanism occurs in the LOCK system design [SAYD87] and in the security policy architecture of SecureWare's Compartmented Mode Workstation [NCSC91b, Sec. 2.2.1].

Finally, depending on the development methodology, there may be several lower levels of system description, ending at the level of executable code and hardware diagrams.

Correctness of elaboration. The appropriateness (as opposed to correctness) of higher-level objectives for the design of a computing system is determined by experience: Objectives are *appropriate* if they counter security threats and are achievable. Security objectives are seldom formalized. The adequacy of a system security policy is also a matter of judgment. Whether a system security policy is *adequate* plays a key role in the successful enforcement of an organizational security policy. In contrast to objectives, the adequacy of a system security policy can be supported through the use of engineering arguments showing how the system policy combines with constraints on the system's use and environment to achieve the identified objectives. Such an argument may be simpler and more persuasive if the system security policy is cast in terms

of external-interface requirements. The formal definition of security must also be informally demonstrated to successfully codify the system security policy. For such a demonstration to be possible, it is necessary to assign real-world interpretations to the key constructs in the formal definition.

If the definition of security is cast at the system level, in terms of formal external-interface requirements, then the correctness of internal requirements can be formally proven. The TCB is largely determined by the elaboration of internal requirements. If the adequacy of the system security policy is based on external-interface requirements, then the trustworthiness of the TCB is established by requiring it to contain those portions of the system whose functionality must be constrained to prove the external-interface requirements. Notice that the TCB enforces only a system security policy. Enforcement of the corresponding organizational policy depends jointly on mechanisms within the TCB, on the correct input by system administrative personnel of parameters of the system security policy (for example, a user's clearance), and on the proper actions of users (proper labeling of input, password secrecy, and so on).

Rules of operation are normally verified by showing that they guarantee satisfaction of the internal requirements. This is a relatively straightforward mathematical effort. A formal top-level specification (FTLS) can be verified either by proving satisfaction of internal requirements or by proving that the FTLS is consistent with the rules of operation.

Formal assurance is not often applied to later stages of elaboration, due to cost constraints. Correctness of the implementation itself is established informally with the help of modern high-level languages, structured walk-throughs, and testing. During these later stages, it is essential to include in the TCB all code needed to support the TCB functionality identified at higher levels.[1] At whatever point formal assurance stops, there must be a transition from mathematics back to real-world concepts; the transition requires physical interpretations for all key concepts in the formal specification. These physical interpretations must be consistent with those used to establish the adequacy of the model's definition of security.

Formal security models

This section looks first at the basic concepts that turn up in security models, then at key distinctions among various kinds of security models, and finally at how these concepts and distinctions are reflected in well-known examples.

---

[1]The contents of the TCB might increase for other reasons as well [SCHA89, Sec. 2.2.1.2].

Basic concepts to be modeled. A secure computing system may decompose into data structures, processes, information about users, I/O devices, and security attributes for controlled entities. The following paragraphs discuss this decomposition in light of the need for accurate physical interpretations of model constructs.

As discussed earlier, the identification of controlled entities plays a crucial role in the development of a security policy model. For systems with a TCSEC evaluation of B2 or above, the controlled entities must include all system resources. An *explicitly controlled entity* is one that has explicitly associated security attributes. In addition to explicitly controlled entities, a system will have implicitly controlled entities: Such an entity might be contained in an explicitly controlled entity or might be a composite entity containing several explicitly controlled entities with differing security attributes.

A *data structure* is a repository for data with an internal state, or value, that can be written (changed) and/or read (observed) by processes (and possibly devices) using well-defined operations available to them. A data structure that is also a minimal, explicitly controlled entity is a *storage object*. Security attributes of storage objects may include security level and user access permissions (access permissions based on user identities, groups, and/or roles). In a model that includes security levels, a storage object has a unique security level, as a result of minimality. Storage objects may be combined to form multilevel data structures that are assigned their own security levels. Multilevel data structures called "containers" are used in the Secure Military Message System (SMMS) to address aggregation problems[2] [LAND84]; the level of a container is required to dominate the levels of any objects or containers inside it.

A *process* may create, destroy, and interact with storage objects and other processes, and it may interact with I/O devices. It has an associated runtime environment and is distinct from the program or code that defines it. An explicitly controlled process is a *subject*. It normally has a variety of associated security attributes, including perhaps a security level, hardware security attributes such as its process domain, the user and user group on whose behalf it is executing, indications of whether it belongs to the TCB, and, in this case, indications of whether it is exempt from certain access control checks. In general, the security-relevant attributes of a subject are part of its runtime environment.

User-related ideas often turn up in security models, despite the fact that users are not controlled entities and are not directly addressed in a system security policy. The users of a system may perform specific *user roles* by executing associated *role-support* programs. In general, a user may have a combination of several roles. As a matter of policy, a given

[2]An *aggregation problem* can occur when a user has access to individual pieces of data in a data set, but not to the entire data set.

user role may require system-mediated authorization and may provide specific system resources needed for performance of the role. The roles of administrative users are of particular interest in the construction of system security policies (for example, security administrator, system operator). Almost by definition, the corresponding role-support programs have security properties that are unacceptable without special constraints on the behavior of administrative users.

User interactions with the system may be modeled in terms of constraints on I/O devices if there is a convention for discussing the current user of a device. External policy on use of the system normally requires that devices pass information only to authorized users. An additional reason for interest in I/O devices is that I/O typically accounts for a large fraction of the TCB (in fact, upwards of 30 percent of an entire operating system [TANE87, Preface]). From the abstract perspective of a security model, I/O devices are encapsulated by TCB software and are typically seen as passive entities. In the TCSEC, devices may transport either unlabeled data or labeled data and are classified as *single-level* or *multilevel* devices accordingly. A significant aspect of multilevel devices is that they are not normally associated with a single security level; thus, they do not act like storage objects.

As already indicated, *security attributes* may be explicitly associated with explicitly controlled entities such as data structures, processes, and devices. Additionally, there may be security-relevant attributes of the system environment such as time of day, day of week, or whether the system or its environment is in a state of emergency.

Taxonomy of security models and attributes. Primary aspects of security policies that are usefully reflected in the security model include the policy objective, locus of policy enforcement, strength of policy enforcement, granularity of user designations, and locus of administrative authority. These policy aspects are normally reflected in the security attributes of controlled entities when the policies are formalized. Security attributes may be implicit; they need not be directly implemented in data structures.

AIS security objectives are traditionally classified as nondisclosure, integrity, or availability objectives [NCSC88a]. A draft Canadian standard has added accountability as a fourth, supporting objective [CSSC92]. *Nondisclosure* objectives guard against inappropriate disclosure of information. Associated system policies and attributes are used to prevent unauthorized release of information. There are two principal kinds of integrity objectives. *Source-integrity* objectives promote the correctness or appropriateness of information sources. Associated system policies guard against unauthorized creation or modification of information. *Content-*correctness objectives promote correctness of information. Associated system policies and attributes not only guard against unauthorized crea-

tion and modification of information but also actively assist in checking correctness of input and in ensuring correctness and appropriateness of software used for processing. *Availability* promotes the availability of information in a timely, useful fashion. Relatively little work has been done with availability policies per se. Availability is a broad topic that includes fault tolerance as well as the prevention of failures that can result from denial-of-service attacks, software errors, and/or design oversights. Finally, *accountability* promotes awareness of user actions. Associated accountability policies involve the collection, analysis, and use of audit data. There have been almost no formal models of accountability policies.

Informational policies maintain security attributes for use by the system and its users, whereas *access control* policies and attributes limit access to system resources and the information they contain. Access control policies are often described in terms of accesses by subjects or other active entities to objects or other passive entities. Access control is enforced by the *reference monitor* portion of the TCB. At lower levels of abstraction, the reference monitor is referred to as the TCB *kernel*.

Access control attributes may be classified according to what they control. A *loose* attribute is associated with a controlled entity, whereas a *tight* attribute is associated with both the entity and the information it contains. Access restrictions determined by tight access control attributes must propagate from one object to another when (or before) information is transferred, because control over the information must still be maintained after it is transferred. The *user granularity* of an attribute may be "coarse," controlling access on the basis of broadly defined classes of users, or it may be "per-user," controlling access by individual users and processes acting on their behalf. Finally, *centralized* authority implies that policy for use of attributes is predefined and takes place under the control of a system security administrator, whereas *distributed* authority implies that attributes are set by individual users for entities under their control. Most security policies use a mix of centralized and distributed authority.

A security attribute belonging to a partially ordered set may be referred to as a *level*. The use of partially ordered levels is usually associated with tight access control policies and constraints on information flow. Constraints on information flow can arise for several different reasons. As a result, a multifaceted policy might associate both *integrity levels* and *nondisclosure levels* with controlled entities to support both kinds of objectives. Within a computing system, the behavior of nondisclosure levels normally depends only on the abstract properties of the partial ordering, rather than on details of the internal structure of the levels (for example, hierarchical classifications and category sets). This fact can be exploited to enlarge the class of policies that a given system can enforce [BELL90].

Example definitions of security. Many if not most security models are deterministic state-machine models that formalize accesses of subjects to objects and are descendants of Lampson's access-matrix model [LAMP71]. In Lampson's model, objects are viewed as storage containers; the access control checks do not depend on the values stored in these containers. Security-critical aspects of the system state are summarized in an access matrix such as the one in Figure 2. In addition to state-machine models, other models of computation have been used successfully in formal definitions of security. Most of these are external-interface models based on event histories. The work of Andrew Moore is a good example [MOOR90].

	Objects		
Subjects	O_1	O_2	O_3
S_1	Read/Execute		
S_2		Write	
S_3	Execute		Read

Figure 2. An access matrix.

The following paragraphs discuss three classes of security models that have arisen in the literature, with emphasis on the taxonomic features mentioned above. All of these examples extend Lampson's access-matrix model by assigning security attributes to objects and adding definitions of security that constrain the possible forms that the access matrix can take.

Traditional mandatory access control (MAC)

- *Objective:* Nondisclosure
- *Locus of use:* Internal and external access control
- *Binding strength:* Tight
- *User granularity:* Coarse
- *Authority:* Centralized

MAC policies that meet TCSEC requirements must enforce nondisclosure constraints on information and may enforce integrity constraints as well. For systems evaluated at B2 and above, there are covert channel analysis requirements which imply that the controls are tight, controlling access to information as well as to individual objects and subjects. Although the TCSEC emphasizes labeling requirements at the user interface, this aspect of MAC has not received much formal attention. MAC

policy identifies users only in terms of their associated maximum nondisclosure levels. Consequently, per-user granularity is not achieved because information available to a user at a given level is also available to other users at that or higher levels. In most systems, most users are not authorized to alter an object's nondisclosure level because decisions about availability of classified information are under centralized control.

A variety of external-interface models have been given for the nondisclosure objective, but not all succeed in providing tight controls on access to information. Noninterference is tight in the case of deterministic systems [GOGU82, GOGU84]. But the modeling problem is more difficult for nondeterministic systems whose behavior is not entirely predictable on the basis of system inputs [WITT90, GRAY91]. Internal requirements models are more common than external-interface models; they are "subject-point-of-view" models that describe constraints on the behavior of individual subject instructions. The best known of these internal constraints are the simple security property (a subject may read only at or below its level) and the *-property (a subject may write only at or above its level). But additional constraints on subject instructions are needed for tight policy enforcement, as are constraints on the scheduling of subjects [WILL91].

An interesting variant of traditional MAC is found in the floating-label policy of the Compartmented-Mode Workstation (CMW) [WOOD87, MILL90]. Each file has a floating "information" label and a fixed "sensitivity" label that dominates the floating label. The floating label is a potentially more accurate label that informs the user but is not used internally for access control. As implemented in the CMW, the floating-label mechanism is not tightly associated with information because of some unresolved covert channel issues.

Most MAC models require a subject that does not belong to the TCB to have a level that dominates the levels of its previous inputs and is dominated by those of its future outputs. This requirement fails to address the possibility that a subject may be able to produce information which is more highly classified than its inputs through some form of aggregation or inference. Definitions of security that accommodate aggregation have been given in the SMMS model [LAND84] and in the work of Meadows [MEAD90a]. Formal treatments of the inference problem are also available [HAIG90, DENN82, Ch. 6].

Traditional discretionary access control (DAC)

- *Objective:* Nondisclosure and integrity
- *Locus of use:* Internal access control
- *Binding strength:* Loose
- *User granularity:* Per user
- *Authority:* Distributed

The objectives behind DAC mechanisms are generally assumed to be user-dependent and have not been well articulated, for the most part. Typically, users are not automatically informed when they print out an object whose access is limited by another owner or, for that matter, when they copy it to another less-protected object. Authority to change discretionary attributes is usually distributed among all users on the basis of ownership, but this is not a TCSEC requirement. The primary distinguishing characteristic of the TCSEC DAC requirements is the ability to control access on a per-user basis.

Bell and LaPadula listed only one internal requirement for discretionary access: In each state, access to an object implied permission to access the information contained in that object [BELL76]. This and other internal requirements of their model were phrased as *state invariants*, properties that must hold in every "secure" state of the system. Unfortunately, their discretionary constraint is not actually satisfied in Unix-like systems because the following can happen: The owner of a file allows access to another user; a process acting on behalf of the other user opens the file; the owner revokes access; the process continues to access the file until it terminates. In Unix-like systems, the discretionary access constraint [BELL76] needs to be weakened to a *state-transition* constraint: If a process does not have a particular kind of access to a file in one state but does in the next state, then it must have had the necessary DAC permissions in the preceding state. The traditional weaknesses associated with DAC are not inherent. Methods for efficient revocation of access have also been devised [KARG89]. Moreover, tight access control mechanisms with per-user granularity and distributed control have been described and modeled by Millen and others [MILL84, ISRA87, GRAU89a, MCCO90].

Integrity à la Clark and Wilson

- *Objective:* Integrity
- *Locus of use:* External and internal consistency
- *Binding strength:* Loose
- *User granularity:* Per user and per procedure
- *Authority:* Centralized

The announced objective of Clark and Wilson's model [CLAR87] is to "ensure integrity of data to prevent fraud and errors." This objective implies some form of "external" consistency between data and real-world situations that the data refers to. External consistency, in turn, imposes recognizable structure on data, a form of "internal" consistency that can be automatically checked. Integrity is supported through various forms of external and internal redundancy. *Separation of duty* ensures that two or more users will independently be involved in key facts known to the

computer, and *integrity-validation procedures* check the expected internal structure of data objects. Access control in the Clark-Wilson model is based on a set of triples of the form (user, program, object-list) used to control how and whether each user may access a given collection of objects. Manual methods are used to ensure that the triples adequately restrict user roles and enforce separation of duty. Clark and Wilson's original model was informal in the sense of this essay. Their access control mechanisms have since been formally modeled in the context of a Unix-based implementation [LAPA91]. But no formal definition of security is available for their work, and it is possible that their mechanisms do not fully address their objectives [MILL91].

Another access control mechanism that has been used to constrain user roles is type enforcement [BOEB85]. The type-enforcement mechanism itself assigns "types" to both subjects and objects. Subject types are referred to as "domains," and the accesses of each subject are constrained by a "type table" based on subject domain and object type. In the lock type-enforcement mechanism, each user has an associated set of domains that subjects running on his behalf may belong to, and these domains are used to define user roles [THOM90a]. In this way, the lock access control mechanisms can also enforce three-way constraints among users, subjects, and objects. Lock type enforcement lacks per-user granularity, but this may be an advantage, if role definitions change less frequently than users. Separation of duty itself has also turned up in role definitions for administrative users. Instructive examples include the Secure Military Message System model [LAND84] and the administrative roles for Secure Xenix [GLIG86], which is now referred to as Trusted Xenix.

Issues in the use of formal methods

Just how "formal" should formal methods be? What are the requirements for their use and what should they be? Is cost-effective use feasible within the current state of the art? The following paragraphs partially answer these questions.

Mathematical rigor and formal logical systems. There are three different common notions of "formal proof" that might be used with formal methods (and thus three somewhat different notions of what formal methods are):

1. *Mathematical proof.* A formal proof is a complete and convincing mathematical argument, presenting the full logical justification for each proof step, for the truth of a theorem or set of theorems [NCSC85].

2. *Machine-checked proof.* A formal proof is evidence accepted by a proof checker showing that a conjecture is a valid consequence of given axioms.
3. *Hilbert proof.* A formal proof in a theory *T* is a sequence of formulas, each of which is either an axiom of *T* or a direct consequence of preceding formulas in the sequence by virtue of a rule of inference associated with the underlying formal system [MEND79].

Mathematical proofs depend on the use of mathematical English for the formulation of a model or specification. They benefit from the basic flexibility of English, but they do not allow for automated assistance. Inaccuracies can result from the fact that there is no precise definition of mathematical English. In most contexts other than the TCSEC, mathematical proofs are regarded as *rigorous* but not *formal* — meaning that they conform to commonly accepted standards of mathematical reasoning but are not supported by a formal semantics in the sense mentioned below.

A machine-checked proof relies on the use of a mechanical *proof checker,* a tool with the following properties:

1. It accepts as input an assertion (called a conjecture), a set of assertions (called assumptions), and a proof.
2. It terminates and outputs either success or failure.
3. If it succeeds, then the conjecture is a valid consequence of the assumptions [NCSC89].

A proof checker has the potential of improving user efficiency. The need for such automated assistance is usually apparent in security policy models for operating systems. The definition of security typically contains dozens of requirements, possibly in the form of state invariants and state-transition constraints. There can easily be upwards of twenty rules of operation, each of which must be consistent with all of the requirements, and this amounts to hundreds of lemmas. Typically, a requirement deals with only a few state components, and any rule of operation that does not alter those components trivially fails to violate that requirement. Consequently, as many as 90 percent of the needed lemmas are likely to be essentially trivial.

A mechanical proof checker is unlikely to achieve a high level of assurance unless it is supported by a formal semantics that determines interpretations of formulas. In this case, a formula ϕ is a *valid consequence* of a set Ψ of formulas if and only if every interpretation that satisfies Ψ also satisfies ϕ. The problem is that, without a formal semantics, the proof checker might not be *sound* and might accept proofs of false conjectures. A *formal semantics* thus involves an interpretation function for formulas and an associated demonstration that a system's proof techniques are

sound. Formal logical systems in the tradition of Hilbert, Tarsky, and others have excellent reputations for soundness, but they are not suitable for practical work because they fundamentally lack powerful semantic conventions [WILL90] needed for readable specifications and proofs [FARM86]. This conclusion has been independently arrived at in an unpublished "Review of Tools and Methods for System Assurance," by Andrews and MacEwen:

> Mathematical formalism has quite different goals for automated reasoning than for mathematical foundations. In the former context, the formalism exists primarily to aid human beings in the task of proving mathematical results. In the latter context, the goal is to clarify philosophical questions about basic mathematical questions: e.g., "What is a proof?", "What are the fundamental limitations of mathematical reasoning?", etc. The practical consequence is that mathematical formalism for automated reasoning ought to resemble quite closely the language of ordinary mathematics.

Fortunately, the goals of formal soundness and expressiveness are not fundamentally inconsistent. An Interactive Mathematical Proof System (IMPS) has been developed at MITRE [FARM91] that not only has an expressive semantics and convenient automated proof mechanism, but also is based on a rigorous, formal semantics whose soundness has been demonstrated relative to older, better known formal systems [FARM90].

Problems in the use of formal methods. Since the TCSEC was first published in 1983, there have been few successes in producing computing systems backed up by formal assurance. A lengthy investigation into the use of formal methods in trusted systems by Marvin Schaefer has the following conclusion:

> Hence, there is some reason to believe that there is value in the very analysis required to perform the exercise of producing a formal abstraction of the security-enforcing portion of a system design... However, it appears that the added value is much less than what is needed to conclude that a product with a verified FTLS is "secure" or even "more secure" than a comparable B3 product [SCHA89].

Schaefer identifies many of the underlying problems in the use of formal methods. The problems may be grouped into three main classes having to do with premature application of the methodology, holes in the chain

of reasoning, and misapplication of the maxim "a chain is as strong as its weakest link."

Early efforts to develop trusted systems underestimated pragmatic difficulties involved in designing, building, and marketing them. Attempts to produce A1 systems without first developing and field testing B1, B2, or B3 systems led to expensive, poorly designed products with no established market. These efforts were accompanied by a belief that NCSC-endorsed verification tools were better than mathematical English. Until recently, the available tools have been unevaluated prototypes, and there has been little published experience in using them. The two existing systems are without a formal basis, for the most part, and they lag behind the current state of the art in automatic theorem proving. However, evaluations of the two currently supported tools have recently been performed, and relevant examples of their use are now available.

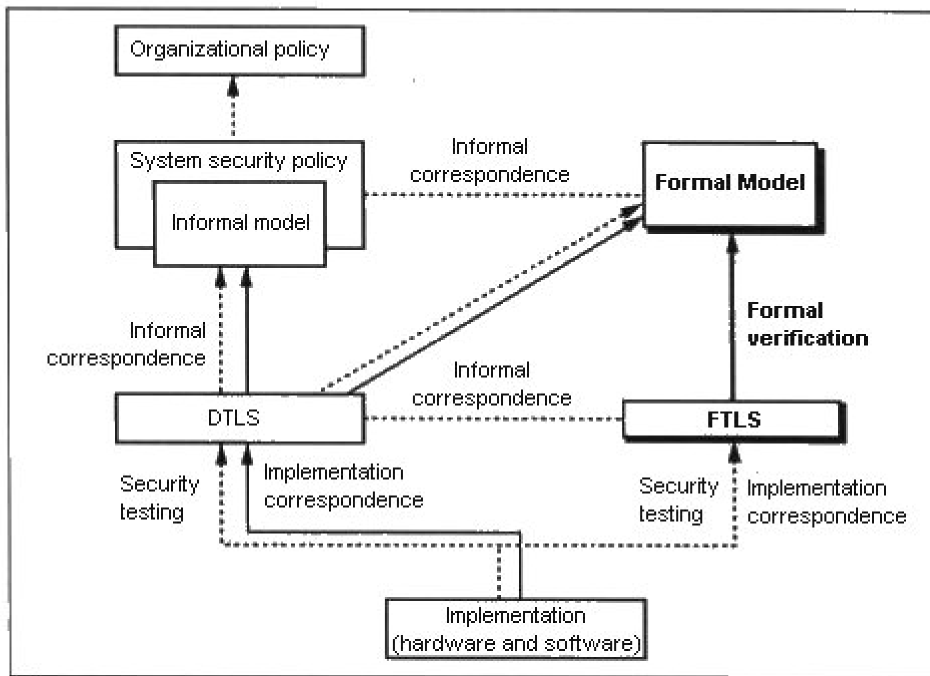

Figure 3. TCSEC assurance requirements.

Figure 3 summarizes TCSEC requirements for the use of formal methods in systems evaluated at B2 and above. Dotted lines indicate potential weaknesses found at B2, B3, and/or A1. To begin with, the TCSEC does not distinguish between the system security policy enforced by the AIS and organizational policies for use of the system. Consequently, there is no requirement for vendors to map their system security

policy to higher-level objectives, including the "Control Objectives" of the TCSEC itself. In the case of nondisclosure objectives, some informal analysis is given in the TCSEC, and a vendor could voluntarily place additional information regarding higher-level objectives in the required "Philosophy of Protection."

The formal model is typically a subject-point-of-view model whose definition of security describes the reference monitor interface in terms of internal constraints on controlled entities. There is no requirement to model the system as a whole and thus no formal demonstration that the purported TCB has been correctly identified. Portions of the TCB not directly involved in access control are typically not modeled, and this tradition is affirmed in the *Trusted Network Interpretation* of the TCSEC [NCSC87a]. Devices are not explicitly modeled. TCB administrative software is not modeled. Typically, various subjects within the TCB are exempt from some part of the access mediation enforced by the TCB on non-TCB subjects, and the proper use of these exemptions is not modeled.

At B2 and above, there is no explicit requirement to make an English translation of the formal model available for nonspecialists, thereby reducing the likelihood of its consistency with the system security policy and descriptive top-level specification (DTLS).

Unmodeled portions of the TCB are constrained by the system security policy and the DTLS, but justification of the DTLS relative to the system security policy is required only at B3. At B2 there is also no requirement to justify the DTLS relative to either the formal model or the system security policy. The FTLS is required only at A1. The FTLS must be used in system testing, but there is no required correspondence between the FTLS and the DTLS, and unformalized portions of the DTLS no longer have to be tested. The implementation correspondence between the FTLS and the source code can take the form of a cheap syntactic comparison, and unresolved semantic discrepancies between the FTLS and the implementation are common. For example, the FTLS might use infinite storage buffers to avoid storage-allocation issues.

In summary, a strong link between an FTLS and a formal model need not be useful, if surrounding links above and below are weak or missing. The maxim that "a chain is only as strong as its weakest link" does not fully apply to the use of formal methods, however. The ideas and specifications associated with the development of a secure system tend to form an exponential hierarchy, with the vendor's definition of security at the top. Each succeeding layer (rules of operation, top-level description, pseudocode, TCB code, and hardware description) is several times the size of the previous layer, with perhaps a hundred lines in the definition of security and 100K lines of TCB source code. As a result, an error near the top of this hierarchy may affect a large portion of the finished code and is likely to be far more serious than an error farther down.

The proper use of formal methods. Formal methods are best suited to the study of relatively small, abstract issues. Researchers routinely provide formal security models, and the cost-effectiveness of formal methods for this purpose is generally accepted. Formal methods are especially important in those cases where traditional system testing is not effective. For example, one does not simply take down the Internet for system testing (at least not without getting a lot of attention). Thus, formal methods have their highest payoff at the early stages of product development and in studying global properties of large systems.

Formal methods can never fully replace informal development methods, and can never substitute for real experience. An A1 system makes much more sense as a follow-on to a series of B1 and B2 or B3 systems than as a new system developed from scratch.

NCSC-endorsed verification tools make sense as benchmarks that establish an expected level of rigor. They need not rule out the use of mathematical English or of newer, unendorsed verification tools, if the resulting evidence compares favorably with that of similar efforts based on the use of endorsed tools. The TCSEC wording already conveys this policy on the use of endorsed tools, and it would be counterproductive to strengthen the requirement beyond what is explicitly stated.