# Integrating Formal Methods into the Development Process

**Richard A. Kemmerer**, *University of California at Santa Barbara*

***Integrating formal specification and verification with development is faster and more cost-effective than doing the steps separately or in parallel, as this effort showed.***

As information-processing systems are used in increasingly sensitive or life-critical environments, inadvertent errors in design or implementation can have far-reaching effects. The need for these systems to be highly reliable is self-evident. Thus, it is important that developers use those techniques and methods that can offer a high degree of assurance that a system will perform as desired. Developers seek assurance that a system's design accurately captures the system's critical requirements and that an implementation in software or hardware is an accurate realization of the system design. By integrating formal specification and verification techniques into the software-engineering process, you can gain this added assurance.

**After-the-fact verification.** The use of formal methods to increase assurance of a critical system's reliability is not a new idea. It has been presented in the literature often, particularly for secure systems. However, most of these have been after-the-fact efforts: The system is built using a standard development approach, and, after it is completed, a formal specification for the system is written and properties are proved about the specification, and/or the code that implements the system is formally verified to be consistent with the formal specification.

This approach increases confidence in the system, but it is costly in both time and money. It is not uncommon to spend an additional 30 to 50 percent of the development cost for the formal specification and verification effort when using the after-the-fact approach. Also, the time for the formal activity occurs after product development is complete, so errors discovered by the formal verification cost more to fix.

**Verification in parallel.** By performing the formal specification and verification effort in parallel with the development,



0740-7459/90/0900/0037/$01.00 © 1990 IEEE

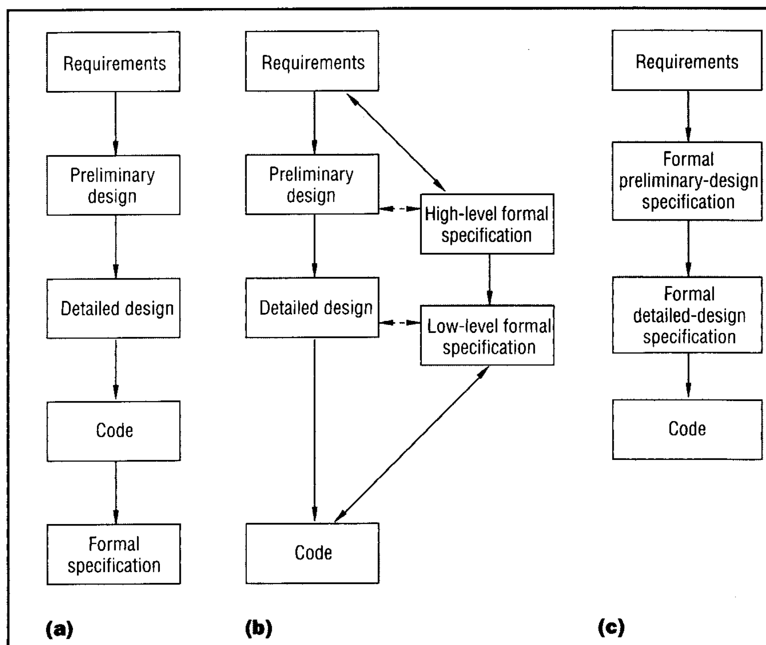


**Figure 1.** Three formal development processes: the **(a)** after-the-fact, **(b)** parallel, and **(c)** integrated approaches.

you can reduce the time to product release from what it is when using the after-the-fact approach. With the parallel approach, there are two teams: the development team and the formal-verification team. The development team uses the standard practices of good development, but at the same time the formal-verification team writes formal specifications for the system and verifies them.

This approach requires constant communication between the two teams so the formal specification and the actual system do not proceed in different directions. Normally, the goal is to formally verify that the code developed by the development team is consistent with the formal specifications developed by the formal-verification team. However, if there is not sufficient communication between the two teams, there is often a large gap between the system developed and its formal specification — you in effect again have an after-the-fact approach.

With the parallel approach, the time penalty is usually not as high as with the after-the-fact approach (especially if there is adequate communication between the two teams), but the monetary cost is about the same.

**Integrated verification.** This article presents another approach: completely integrating the formal methods into the development process. The development team and the formal-verification team should be one. There should not be two separate processes, but rather a single integrated process where the developers use formal specifications as their design notation. With the integrated approach, the time penalty is less than with the after-the-fact approach and the cost is less than either of the other approaches.

Many problems that occur in development projects are the result of developers rushing off and generating code without thinking about the design. By using formal notation for the design documents, you can reason rigorously about the design *before* writing any code. The key is integration.

Figure 1 shows the three verification processes.

**Secure Release Terminal.** The security community deals with critical systems, which contain highly sensitive data; therefore, members of that community are willing to pay the extra cost in time and money for added assurance that their systems will perform as desired. My colleagues and I used the integrated approach in developing the Secure Release Terminal, an example problem from the security domain.

The SRT's purpose is to move appropriately classified machine-readable data from a processing environment at one security level to a processing environment at another level. The SRT design was carried out using the Formal Development Methodology.[1] (The box on p. 44 describes some of the notation used in this article from the Ina Jo specification language that we used.) The SRT design team viewed formal specification and verification as an integral part of the design process.

Although the justification for using a formal verification methodology for this project was to generate the certification needed by the US Defense Dept. to evaluate and approve the SRT for use in a multilevel operational environment, the design team also wanted to ensure from the start that the SRT design could meet the desired security requirements. Thus, the formal specification was tightly integrated with the design to the extent that the evolving design specifications were the formal specifications themselves.

## Verification process

Formal verification originally dealt with formally verifying that programs written in a high-level language satisfied their specifications, which were expressed in a formal mathematical notation. This is commonly called program verification or code verification. As people tried to verify larger and more complex programs, the detailed specifications for the program became harder to understand and were more susceptible to error. It also became more difficult to associate these complex program specifications with the system's original requirements.

To address this problem, formal methods were introduced into the early phases of the life cycle. In this approach, you express the preliminary design in a formal notation, analyze the design, and prove formal statements about the preliminary design. Furthermore, as you develop a more detailed design, you formally verify it to be consistent with the more abstract preliminary design, which you have already shown to satisfy the more abstract desired properties, like safety or security. This process is called design verification or specification verification.

The higher level verification of more abstract objects is necessary to derive the ap-

propriate program-level formal specifications, but code-level verification must also be part of the process for the process to be complete. In fact, some people have argued that the process is not complete without also verifying the hardware on which the code runs.[2]

**Formal specifications.** The detailed specifications for large and complex programs are hard to understand and are susceptible to error. Without a hierarchical approach to develop formal specifications, it is extremely difficult to associate the complex program specifications with the system's original requirements.

To construct formal specifications for highly reliable software, you decompose it into several easier problems:

First, you state the critical requirements, which are usually an English statement of what is desired, in precise mathematical terms. For example, for a banking system, the critical requirements could be that the total deposits to an individual's account over time exceed the total withdrawals from the same account.

Next, you produce a high-level formal specification of the system. This specification gives a precise mathematical description of the system's behavior, omitting many implementation details like resource limits.

You may follow this by less-abstract specifications that implement the next higher level specification, but with more detail.

Finally, you code the system in a high-level language. This high-level-language implementation must be shown to satisfy the original critical requirements.

It becomes readily evident that it is difficult to demonstrate that the high-level-language code is consistent with the critical requirements. However, it can be done by verifying the design at every step. By using formal notation for every level of the design, you can reason rigorously about the design (with or without the aid of a verification system) before writing any code. When the code is finally produced, you can formally verify that the code is consistent with its specifications.

**Formal verification.** The first step of the verification process is to *informally* verify that the formal critical requirements properly reflect the customer's critical requirements. This step is necessarily informal because the customer's requirements are not formal. If the customer's requirements were written in a formal notation, this step could also be formal.

Because the critical requirements are at a high level of abstraction and contain no unnecessary details, it is usually straightforward to review the formal statement of the critical requirements with the customer who generated the requirements.

Next, you must prove that the highest-level specifications are consistent with the formal critical requirements. This approach differs based on whether the specifications are presented using a state-machine approach [3] or an algebraic approach.[4] In the SRT effort, we used the state-machine approach.

In the state-machine approach, you specify the effect of performing each operation based on certain conditions being satisfied when the operation is invoked: For each operation, there are entry and exit conditions, and if the system state before the operation is invoked satisfies the entry conditions, the state after the operation completes execution will satisfy the exit conditions. When using the state-machine approach, you must verify that the initial state satisfies the formal critical requirements and that every operation preserves the critical requirements.

After showing that the highest-level formal specification is consistent with the formal critical requirements, you must show that the next lower level specification, if one exists, is consistent with the level above it. This process continues from level to level until the lowest-level specification is shown to be consistent with the level above it.

Finally, you must show that the high-level-language implementation is consistent with the lowest-level specification.

Because each level of specification is shown to be consistent with the level above, and the high-level-language implementation is shown to be consistent with the lowest level, by transitivity the high-level-language implementation is consistent with the highest-level specification. In addition, because the highest-level specification was shown to satisfy the formal model of the critical requirements, the implementation satisfies the formal model.

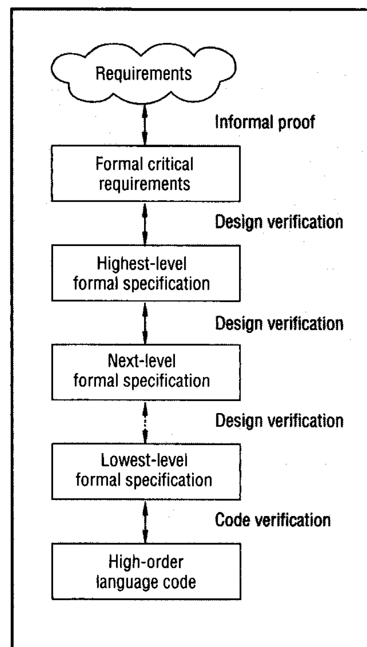


**Figure 2.** Formal-verification hierarchy.

The formal verification that the highest-level specification is consistent with the formal critical requirements, as well as the verification that each formal specification is consistent with the more abstract specification that immediately precedes it, is a form of *design verification.* The formal verification that the high-level-language code is consistent with the lowest-level formal specification is *code verification.* Figure 2 shows the formal-verification hierarchy.

## Secure Release Terminal

The SRT was conceived as a project to solve a problem frequently faced in classified applications: how to move appropriately classified data from a high-security-level processing environment to a lower-security-level processing environment.[5] An example of such data would be a report prepared on the high-security-level system using data from a sensitive document but that includes only information that is not sensitive — the high-security data items have been deleted. Thus, you can view the report as *sanitized* information because it includes only information appropriately classified for release to the lower-security-level system.

If nonsensitive information is to be moved from the high-security system to the low-security system, some way must exist to ensure that sensitive data is not carried with the supposedly sanitized re-

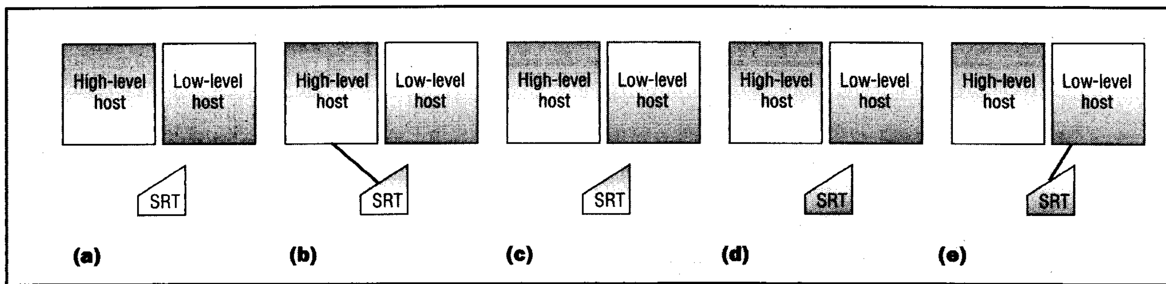


**Figure 3.** Example Secure Release Terminal scenarios: **(a)** sanitized, **(b)** connected to a high-security-level host, **(c)** in review mode, **(d)** after review, accept, and level change, and **(e)** connected to a low-security-level host.

port. The previous approach to this problem was to print the report, have a security officer review it, and then manually reenter the report into the low-security system. Clearly, it is more desirable to move the information in machine-readable form. This requires a way to ensure that all the information moved from a high-security to a low-security system is reviewed before releasing the information to the low-security system. In addition, reviewed information should not be changed after review and before release. This is the purpose of the SRT.

The SRT provides the link between the high-security and low-security systems. It provides the means for a security officer to review all data, including normally nonprinting ASCII characters. The SRT serves as the security reference monitor (it is always invoked, tamper-proof, and correctly enforces the desired security policy) for data flowing between the high-security and low-security systems.

The SRT, in concert with the security officer, is the sole security barrier to prevent high-security data flow into the low-security system. Therefore, it must be designed to provide a high level of confidence that it correctly performs its security function.

**Design decisions.** The availability of inexpensive hardware as well as the desire to minimize the complexity of the SRT design resulted in designing the SRT as a single-user system in which all the software was to be subjected to a formal development methodology. This decision led to an implementation where the SRT trusted code is smaller than many operating-system kernels.

The second simplifying design decision was to limit the functionality of the terminal to only those functions required from a security viewpoint. Thus, although it was to use a smart terminal, the SRT contains no text editor. All text editing is performed on the attached host, with the SRT receiving the data only after the necessary editing to prepare a file (or message) for release has been performed on the remote host. The role of the SRT is thus reduced to that of a text-review and -release station.

**Using the SRT.** The SRT has two primary modes of use: a connected mode where it acts as a dumb terminal and a disconnected mode where it performs a secure-review function. When the SRT is connected to a remote host, all the standard operations available to a user connected using a standard terminal are available to the SRT user. In addition, the SRT user may send data, receive data, and disconnect. The operations available when disconnected are reviewing data, accepting data, changing level, sanitizing, and connecting.

The connected mode is initiated by connecting to a high- or low-security host. In this mode, the user interacts with the host, using the host's text editor and preparing text for release. To connect to a host, the SRT's security level must be the same as that of the host to which it is connecting. Furthermore, the SRT's security level cannot change while it is connected to a host.

(The SRT was originally perceived as moving data only from a high-security host to a low-security host. However, early in the design process the design team realized that a more general system that let data be moved from a high to low host or from a low to high host would be better and would not complicate the design.)

When the user has prepared the text for release to the other system, the user initiates a receive-data function, which is a reliable file transfer of the data from a transfer file in the attached host to a holding buffer in the SRT. When the transfer is complete, the user disconnects the SRT from the host.

Because the data in the terminal originated in the host, the SRT's security level is still equivalent to that of the host to which it was just connected. The user can now use the text-review functions and command the SRT to display the first or last screen of the text buffer. The user can also page through the text buffer to display the next screen of data or to move the screen down or up several lines.

The user can also choose between two display modes. The first mode lets the user view just the printing ASCII characters. In the second mode, the user can also view the normally nonprinting ASCII control characters.

Once the user has reviewed all the data in either display mode, he can accept the text for a level change. The SRT will not let the data be accepted for a level change until every line has been displayed on the screen.

After reviewing and accepting the data, the user can request the SRT to change the data's level. He does this by changing the terminal's level.

If he has reviewed and accepted the data and changed its level, the user can then connect the SRT to the destination host. First, the user connects to the high or low host as appropriate and then performs a send-data function, which is a reliable file transfer from the SRT to the destination host.

If during the review the user finds that the data is unacceptable for transfer or if for other reasons (like it is the end of a work shift or a higher priority transfer was requested) the user wants to abort the transfer, he can purge the terminal's contents by executing the sanitize operation. In addition to purging the terminal's buffer, this changes the terminal's level to reflect the sanitization. Because a sanitized terminal contains no data, it may be connected to a host of any security level. Figure 3 shows some of the steps in moving data from one host to another.

**Security requirements.** In this system,

the critical security properties that had to be preserved were that

- the SRT can be connected only to a host at the same level;
- for an SRT to change levels, its text buffer must first be sanitized or else the data it contains must be reviewed and accepted for a level change before the change can take place;
- the data in the SRT must previously be marked as reviewed before it can be marked as accepted (these markings must be performed in sequence);
- if the data in the SRT is marked as accepted, it must be also marked as reviewed (data cannot be accepted but not reviewed); and
- if the SRT is connected, it cannot be marked as reviewed or sanitized.

## Formal design specifications

When using the Formal Development Methodology and the Ina Jo specification language, you specify the design with increasingly detailed levels of abstraction. Thus, at the highest level, you specify the complete system, although at a high abstraction level. In the SRT effort, this high-level view provided the design team with the opportunity to discuss and document at a high abstraction level the functions that the SRT should provide without being bogged down by implementation details. This first level corresponds to the preliminary design.

At the second level, you add additional detail. In the SRT effort, the lower level states were a refinement of the higher level states, and we used a mapping function to specify how each Ina Jo type, constant, variable, and transform is implemented in the lower level.

Also, the second level let us clarify the notion of what constitutes proper review: What had been viewed as a single review function that would be applied many times to complete the review of the data in the buffer was refined into seven review functions in the detailed design. This was possible because more structure was added to the state variables as the appropriate design decisions were made. At this stage, the design team also recognized that text files contain both printing and nonprinting (control) ASCII characters, so the team included two display modes (normal and control) toggled by a change-mode function.

The SRT code was written directly from the lowest-level formal specification. There were not two separate development and formal-specification processes. Instead, there was a single integrated process where the formal specifications *were* the design.

```
N"Terminal_Level≠Terminal_Level →
    ~Connected
  & (Terminal_Level=Sanitized&N"Connected
    |Terminal_Level=High&N"Terminal_Level=Low&Accepted&NC"(Connected)
    |Terminal_Level=Low&N"Terminal_Level=High&Accepted&NC"(Connected)
    |N"Terminal_Level=Sanitized&NC"(Connected))
```

**Figure 4.** Constraint to express the requirement that the SRT can change level only when it is disconnected from a host and its text buffer is sanitized or when the data in its text buffer is accepted for level change.

**Critical requirements.** As outlined earlier, the first step of the formal-specification process is to formally express the system's critical requirements. When using the Formal Development Methodology, you express the formal critical requirements with state invariants (called criteria in Ina Jo) and constraints. State invariants are relationships between the state variables that must hold at all times (in all reachable states), and constraints are relationships that must hold between two successive states (they express the relationship between the old state and the new state).

The invariants for the example system defined what constitutes a secure state for the SRT. For example, the SRT can be connected only to a host at the same security level as the SRT. But invariants alone are not sufficient to define all critical requirements because the relationships between consecutive states also have security ramifications that must be controlled. For example, when there is a change in the SRT's security level during a state transition, certain relationships must hold for the system to maintain its security. One example is that the SRT must be prevented from going from the "connected to high-level host" state to the "connected to low-level host" state without going through either sanitization or review and acceptance. Because this requirement deals with successive states, in Ina Jo it is expressed as a constraint.

The SRT's five critical security requirements (described earlier) were specified by the following invariants and constraints. The security requirement that the SRT can be connected only to a host at the same security level as the SRT was expressed in the invariant

```
Connected → Terminal_Level=Active_Host
```

where Active_Host is the security level of the connected (active) host; it represents the hardware switch that connects a communication line to the host.

The requirement that the SRT can change level only when it is disconnected from a host and its text buffer is sanitized or when the data in its text buffer is accepted for level change was expressed in the constraint shown in Figure 4. This constraint specifies that if the SRT's level is different in the new state than it was in the old state, the terminal was not connected and one of four things occurred:

- The SRT was already sanitized and is now connected at the host's level.
- The SRT had already accepted the data and the SRT's level was changed from high to low.
- The SRT had already accepted the data and the SRT's level was changed from low to high.
- The SRT became sanitized.

For the last three possibilities, the terminal remains disconnected.

The requirement that the text buffer be reviewed before it is accepted was expressed by the constraint:

```
N"Accepted → Reviewed
```

and the requirement that it must be marked as reviewed when it is marked as accepted was expressed by the invariant

```
Accepted → Reviewed
```

Finally, the fifth security requirement (that the SRT cannot be marked as reviewed or sanitized if it is connected) was expressed by the invariants

```
Reviewed → ~Connected
```

and

```
Terminal_Level=Sanitized → ~Connected
```

After the invariants and constraints were written, both the design team and the customer reviewed them to ensure that they contained all known critical requirements (in this case, security ones).

This review process was aided by the fact that the invariants and constraints were unambiguous, thanks to the formal notation used. A potential danger of using English specifications to express the critical requirements is that the customer and the developers may interpret the same specification differently because of the language's ambiguity, so they may think that they agree on some point when in reality they do not. Of course, there is still some potential for misinterpretation when using formal notation, because how the values of some of the variables, like Reviewed, are set have not yet been fully defined.

It is these critical requirements against which the formal specifications constituting the preliminary design must be verified. The Ina Jo processor will automatically generate proof obligations that guarantee that the formal specification's transitions preserve the criteria and that the values of the variables in their old and new states satisfy the constraints.

## Preliminary design

The SRT's preliminary design consisted of a state description expressed in Ina Jo. The description contains state variables and state transitions. Changes to the SRT's state are produced as a result of user-initiated actions, which are performed at the SRT's user interface (at the terminal keyboard). These actions are:

- Connect To(Lev: Connect_Level), which connects the SRT to the level Lev host.
- Disconnect, which disconnects the terminal from the host.
- Receive Data, which initiates a file transfer from the connected host to the SRT.
- Send Data, which initiates a file transfer from the SRT to the connected host.
- Sanitize, which sanitizes the data-storage buffer in the SRT.
- Review Data, which reviews the data-storage buffer and may mark the buffer as reviewed.
- Accept Data, which accepts the data in the buffer for a level change. The data must have already been reviewed.
- Change Level, which toggles the SRT's security level from high to low or from low to high. The data must already have been accepted.
- Process Normal, which models normal data processing at the remote host. The SRT must be connected, and it acts as a dumb terminal.

**Specification.** In the preliminary-design specifications, the design team expressed these operations as Ina Jo transforms. The Ina Jo transform for connecting to a host is

```
Transform
Connect_To(Lev:Connect_Level)
```

***This review process was aided by the fact that the invariants and constraints were unambiguous, thanks to the formal notation used.***

```
Refcond
  ~Connected
  & (Terminal_Level=Lev |
     Terminal_Level=Sanitized)
Effect
  N"Connected
  & N"Terminal_Level=Lev
  & N"Active_Host=Lev
  & ~N"Reviewed
  & ~N"Accepted
```

The Refcond part of the transform expresses the conditions that must hold for the transition to take place. For the Connect_To transform, these conditions are that the terminal must be disconnected (~Connected) and the terminal must either be at the level of the host it is connecting to or be sanitized.

The Effect part of the transform expresses the resulting state after the state transition occurs (after the transform fires). After the Connect_To transform fires, the terminal's level is equal to the level of the host it connected to; the value of the active host is also equal to this level; and the terminal is marked as connected, as not reviewed, and as not accepted. The terminal is marked as not reviewed and as not accepted because as soon as it connects to a remote host it is assumed that the terminal's contents can be contaminated by the data at that host.

Any state variables that do not appear in the Effect section are assumed to not change. That is, the Ina Jo processor will automatically conjoin the expression

```
N"x = x
```

to the Effect expression for any variable $x$ not explicitly mentioned as changing in the Effect section of the transform. Therefore, for the Connect_To transform, the Ina Jo processor conjoins

```
N"Terminal_Buffer=Terminal_Buffer
```

to the Effect expression whenever it uses the Effect section to generate a proof obligation. The same is true for the variable Host_Buffer.

In this highest-level specification, the buffer has no structure. It could be a list of lines, a list of characters, a list of bits, or some other structure. At this stage, the team was not ready to commit to a specific structure. What was important was that a buffer could be reviewed and be accepted. Thus, there are two Boolean state variables in the specification: Reviewed and Accepted.

The Review_Data transform provides an example of the use of the Review state variable, as well as an example of one form of nondeterminism in Ina Jo.

```
Transform Review_Data
  Refcond
    ~Connected
  Effect
    N"Reviewed | NC"(Reviewed)
```

For this transform to fire, the terminal must be disconnected. The Effect expression indicates that the new value of the state variable Reviewed is either true (N"Reviewed) or unchanged (NC"(Reviewed)) — the result of this transform is nondeterministic.

At this stage, the design team had not decided what the criteria should be for marking the terminal buffer as being reviewed. It also did not know exactly what the review functions provided would be.

## Partial preliminary-design specification

```
Specification Secure_Release_Terminal
LEVEL Preliminary_Design
  TYPE
    Level = (High,Low,Sanitized),
    Connect_Level = T"L:Level (L=High | L=Low),
    Buffer

  CONSTANT
    Sanitized_Buffer:Buffer

  VARIABLE
    Terminal_Level:Level,
    Connected:Boolean,
    Reviewed:Boolean,
    Accepted:Boolean,
    Host_Buffer(Connect_Level):Buffer,
    Terminal_Buffer:Buffer,
    Active_Host:Connect_Level

  INITIAL
      Terminal_Level=Sanitized
      & ~Connected
      & ~Reviewed
      & ~Accepted
      & Terminal_Buffer=Sanitized_Buffer
```

*The criterion, constraints, and transforms are given in the main article*

```
  CRITERION
  CONSTRAINT

  Transform Connect_To(Lev:Connect_Level)
  Transform Disconnect
  Transform Receive_Data
  Transform Send_Data
  Transform Review_Data
  Transform Accept_Data
  Transform Change_Level
  Transform Process_Normal

END Preliminary Design
END Secure_Terminal
```

This was partially the result of not knowing what the buffer's structure would be. For example, the customer might want the entire buffer to be displayed or might want the entire buffer to be displayed with nonprinting characters both expanded and suppressed.

At one point in the design, the team considered checking for particular encodings that would let information be covertly signaled in a file being downgraded. However, because the number of possible encoding schemes that should be checked was large and, more important, because this type of check was not being performed by the security officer using the manual system, the team soon abandoned this idea. The team decided that at a minimum every part of the buffer should be displayed at some time during the review process.

It was also evident that the system should determine when this condition has been satisfied rather than having the user decide. (The user has the option to accept the data or not, but not until the system determines that the buffer has been properly reviewed.)

Rather than force a decision about what was meant by Reviewed and what review functions were to be provided, the design team chose to have a single review transform that would eventually be refined into the desired functions later in the design process. At first, it may seem that it makes no sense to have a formal requirement in terms of variables (like Reviewed) that are not fully defined. However, Reviewed, which is declared to be a Boolean variable, is initially false. It can become true only as a result of firing the Review_Data transform; it becomes false whenever the Sanitize or Connect_To transforms are fired; and it is unchanged by all other transforms. Thus, although Reviewed is not fully defined, it is constrained at the appropriate level of detail for a preliminary-design specification.

Furthermore, at this stage, there had been no agreement on what it meant for a buffer to be reviewed; therefore, the formal specification is an accurate documentation of the design at this stage. Because the setting of the variable Reviewed was not fully defined, the design team had to carefully analyze the mapping function to see how Reviewed later got implemented in the detailed-design specification, when that specification was written.

The design team also commented the preliminary-design specification to indicate what the competing ideas were for the definition of Reviewed, so this information was not lost.

What we knew at this stage was that by executing the appropriate review transitions, we would eventually have reviewed the buffer (N"Reviewed) and that no review function would make an already-reviewed buffer not reviewed. Thus, the result is nondeterministic. The Ina Jo expression that captures this is

```
(~Reviewed & N"Reviewed)
| (~Reviewed & ~N"Reviewed)
| (Reviewed & N"Reviewed)
```

This is logically equivalent to the Effect expression for the Review_Data transform presented earlier.

The other transforms of the preliminary-design specification can be interpreted similarly.

The final part of the preliminary-design specification presented here is the Initial clause, which defines the system's initial state. For the SRT, the initial clause specifies that the terminal level will be sanitized, the terminal will be disconnected and marked both as not reviewed and as not accepted, and its buffer contents will be a sanitized buffer.

The box above shows part of the preliminary-design specification. The details of the criteria, constraints, and transforms are not shown.

The design team viewed formal specifications as a convenient design notation that provided a rigorous statement of the evolving design. The use of formal specifications forced detailed technical discussions early in the design process, with attendant analysis of the security ramifications of design decisions. The results of such discussions were captured in the formal specifications and their effects analyzed by the entire design team.

**Verification.** After the preliminary-design specifications were completed, they were input to the Ina Jo processor to produce the necessary proof obligations to guarantee that the transforms specified would satisfy the formal critical requirements.

## Formal specification language

The formal method we used in developing the SRT is the Formal Development Methodology, which is an example of the state-machine approach to formal specification. When using the state-machine approach, you view a system as being in various states. One state is differentiated from another by the values of the state variables, and the values of these variables can be changed only via well-defined state transitions.

The Formal Development Methodology's formal specification language is Ina Jo, a nonprocedural assertion language developed by Unisys that is an extension of first-order predicate calculus. The key elements of Ina Jo are types, constants, variables, definitions, initial conditions, criteria, and transforms.

A criterion is a conjunction of assertions that specify the critical requirements for a good state (like a secure state). A criterion is often called a state invariant, because it must hold for all reachable states, including the initial state.

An Ina Jo transform is a state-transition function. It specifies what the values of the state variables will be after the state transition relative to their values before the transition. It also specifies any conditions that must hold for the transition to occur. The system being specified can change state only as described by one of the state transforms.

In Ina Jo, the following symbols are used for logical operations:

- & for logical And,
- | for logical Or,
- ~ for logical Not, and
- → for logical implication.

There is also a conditional form (if *A* then *B* else *C*) where *A* is a predicate and *B* and *C* are well-formed terms.

The language also has the following quantifier notation:

- ∀ for "for all" and
- ∃ for "there exists."

Three other special Ina Jo symbols used in the main article are

- N" to indicate the new value of a variable (N"*v1*) is the new value of variable *v1*),
- NC" to indicate no change to a variable's value (NC"(*x*) is equivalent to N"*x*=*x*), and
- T" to define a subtype of a given type *T*.

The Formal Development Methodology uses an inductive approach to generate the necessary proof obligations to assure that the critical requirements are preserved. In this approach, you must first show that the criteria hold in the initial state. Next, for every transform, you must show that if the transform fires in a state where the criteria hold, the resulting state also satisfies the criteria and the previous and new states satisfy the relationships expressed by the constraints. (The initial state is the basis case and the induction is on the transforms.) Thus, the transforms can be fired in any order and, by induction, any reachable state will satisfy the criteria and any two consecutive states will satisfy the constraints.

The first proof obligation generated by the Ina Jo processor is the Initial Conditions Theorem:

INIT → CR

where INIT is the Initial clause and CR is the criterion in the specification's Criterion clause.

In addition, for each transform the Ina Jo processor generates a transform theorem:

CR & R & E → N"CR & CO

where *R* and *E* are the Refcond and Effect, respectively, for the transform and CO is the Constraint clause.

We used the Ina Jo processor to generate the necessary proof obligations for the preliminary-design specification. These were all proved using the Formal Development Methodology's Interactive Theorem Prover.[6]

In addition — or as an alternative — to verifying the formal specifications at each design stage, you can test the formal specifications to see if they specify the desired functionality. Several approaches have been proposed;[7,8] one deals specifically with Ina Jo specifications.[8] Some of these approaches serve as rapid prototypes for resolving design decisions early in the life cycle. They are all useful for detecting errors early in the development process.

## Detailed design

As more design decisions were made, the next level of specification was developed to document these decisions. The box on p. 45 shows part of the resulting formal specification, which corresponds to a detailed-design specification for the SRT.

**Specification.** A review of this document shows that at this stage buffers have taken on some structure. A buffer is shown as a list of lines, but lines are still not explicitly defined. This specification also introduces the idea of the SRT having a screen on which lines can be displayed:

Screen: Screen_Buffer

where

Screen_Buffer: Loline

and

Loline = List of Line

The idea of being able to display a line with nonprinting characters expanded or suppressed is also introduced at this stage.

To rigorously associate the entities of the lower level specification with the refined entities of the parent specification, the Formal Development Methodology requires a Map section in each lower level specification. All types, constants, state variables, and transforms that appear in the parent specification must be mapped to their corresponding representations in the lower level specification.

By introducing a display screen into the design, the team could then refine the Review_Data transform into seven transforms that corresponded closely to the review functions to be provided to the user. These transforms are

- First Page, which displays the SRT data buffer's first screen,
- Last Page, which displays the last screen,
- Plus Page, which displays the next screen,
- Minus Page, which displays the previous screen,
- Plus Lines, which displays the next *N* lines,
- Minus Lines, which displays the previous *N* lines, and
- Change Mode, which toggles the SRT between normal and control-character display modes.

The mapping expression for the

## Partial detailed-design specification

```
Specification Secure_Release_Terminal
LEVEL Detailed_Design UNDER Preliminary Design
  TYPE
    Level = (High,Low,Sanitized),
    Connect_Level = T"L:Level (L=High | L=Low),
    Line,
    Loline = List of Line,
    Buffer = Loline,
    Screen_Buffer = Loline,
    Pos_Integer = T"i:Integer (i>0),
    Mode = (Normal,Show_Control)

  CONSTANT
    Max_Buffer: Pos_Integer,
    Screen_Size:Pos_Integer,
    Sanitized_Line:Line,
    Scroll_Size:Pos_Integer,
    Expanded(Line):Line

  TYPE
    Screen_Number = T"i:Pos_Integer (i≤Screen_Size),
    File_Number = T"i:Integer (i≥0 & i≤Max_Buffer),
    Buffer_Number = T"i:Pos_Integer (i≤Max_Buffer)

  CONSTANT
    Sanitized_Buffer:Buffer

  VARIABLE
    Terminal_Level:Level,
    Terminal_Mode:Mode,
    Connected:Boolean,
    Reviewed(Buffer_Number):Boolean,
    Accepted:Boolean,
    Host_Buffer(Connect_Level):Buffer,
    Terminal_Buffer:Buffer,
    Active_Host:Connect_Level,
    Screen:Screen_Buffer,
    File_Size:File_Number,
    Buffer_Size(Connect_Level):File_Number,
    Top_Line:File_Number

  DEFINE
    Definitions given in the main article

  INITIAL
      Terminal_Level=Sanitized
    & ~Connected
    & ∀ i:Buffer_Number(~Reviewed(i))
    & ~Accepted
    & ∀i:Buffer_Number (Terminal_Buffer.i=Sanitized_Line)
    & File_Size=0
    & Terminal_Mode=Normal
    & ∀s:Screen_Number (Screen.s=Sanitized_Line)

  Transform Connect_To_High
  Transform Connect_To_Low
  Transform Disconnect
  Transform Receive_Data
  Transform Send_Data
  Transform Sanitize
  Transform Accept_Data
  Transform Change_Mode
  Transform First_Page
  Transform Last_Page
  Transform Plus_Page
  Transform Minus_Page
  Transform Plus_Lines
  Transform Minus_Lines
  Transform Change_Level
  Transform Process_Normal

  MAP
    Buffer == Buffer,
    Sanitized_Buffer == Sanitized_Buffer,

    Terminal_Level == Terminal_Level,
    Connected == Connected,
    Reviewed == ∀i:Buffer_Number (
        i≤File_Size → Reviewed(i))
        & File_Size ≠ 0,
    Accepted == Accepted,
    Host_Buffer(Lev) == Host_Buffer(Lev),
    Terminal_Buffer == Terminal_Buffer,
    Active_Host == Active_Host,

    Connect_To(Lev) == ( Lev=High & Connect_To_High
        | Lev=Low & Connect_To_Low),
    Disconnect == Disconnect,
    Receive_Data == Receive_Data,
    Send_Data == Send_Data,
    Sanitize == Sanitize,
    Accept_Data == Accept_Data,
    Review_Data ==
          Change_Mode
        | First_Page
        | Last_Page
        | Plus_Page
        | Minus_Page
        | Plus_Lines
        | Minus_Lines,
    Change_Level == Change_Level,
    Process_Normal == Process_Normal

END Detailed_Design

END Secure_Terminal
```

Review_Data transform is

```
Review_Data ==
  Change_Mode
  | First_Page
  | Last_Page
  | Plus_Page
  | Minus_Page
  | Plus_Lines
  | Minus_Lines.
```

What precedes the == sign in a mapping expression is an entity from the upper level (the preliminary-design specification), and what follows the == is an expression using entities from the lower level (detailed-design specification). This mapping indicates that the Review_Data transform is nondeterministically implemented by the seven lower level transforms. (An execution of any of the seven lower level transforms corresponds to an execution of the Review_Data transform.)

The Change Mode transform introduces the idea of being able to display control characters. To accomplish this, the specification used the constant function Expanded:

```
Expanded(Line): Line
```

Because a line's structure is undefined at this stage, the Expanded function is also undefined, but it represents the line that results when control and other nonprinting characters are expanded to a printable form. For example the sequence <a, Ctrl-H, b> would normally be displayed as "b" because the Ctrl-H backspaces over the "a." In expanded mode, this might be displayed as "a^Hb". (In the SRT's initial implementation, control characters were displayed in reverse video.) The Change_Mode transform is

```
Transform Change_Mode
  Refcond
    ~Connected
  Effect
    ( Terminal_Mode=Normal
    & N"Terminal_Mode=Show_Control
    & ∀s:Screen_Number( N"Screen.s=
        Expanded(Terminal_Buffer.
                (s+Top_Line-1)))
    | Terminal_Mode=Show_Control
    & N"Terminal_Mode=Normal
    & ∀s:Screen_Number( N"Screen.s=
        Terminal_Buffer.(s+Top_Line-1)))
```

The Change_Mode transform's only requirement to be invoked is that the SRT be disconnected. This reflects the decision that the display mode was to be used only during the review process and not when the SRT is connected to a remote host and used as a dumb terminal.

The Effect section is a disjunction corresponding to two cases: The SRT is in normal mode or it is in show-control mode. If the SRT is in normal mode when the transform is fired, it is put in show-control mode. Similarly, if it is in show-control mode, it is put in normal mode. The lines displayed on the screen must also be updated accordingly.

The remaining six transforms that are a refinement of Review_Data determine what lines will be displayed on the screen. All have a Refcond that requires the terminal to be disconnected for the transform to fire. Consider the Minus_Page transform. The Effect section for this transform is

```
N"Top_Line =
  (if Top_Line - Screen_Size > 0
     then Top_Line - Screen_Size
     else 1)
& ∀i:Buffer_Number(
  N"Reviewed(i) =
    (if i ≥ N"Top_Line & i ≤ N"Bottom_Line
      then TRUE
      else Reviewed(i)))
& ∀s:Screen_Number( N"Screen.s=
    Display(Terminal_Buffer.
        (s+N"Top_Line-1)))
```

The Top_Line state variable keeps track of which line in the buffer is displayed at the top of the screen. The constant Screen_Size indicates how many lines the screen can display. Bottom_Line is an example of the use of an Ina Jo definition; it makes the specification more readable. The definition is

```
Bottom_Line = Top_Line + Screen_Size - 1
```

The other definition used in this transform expression is Display. It is defined as

**When developing the detailed design, the design team decided that it wanted all of the SRT's operations to be single keystrokes (to correspond to function keys on the terminal).**

```
Display(li:Line) ==
  (if Terminal_Mode=Normal
    then li
    else Expanded (li))
```

Without the Display definition, whenever the effect of a transform changes what is displayed on the screen, it would be necessary to use a conditional statement in the specification like the one used in the definition of Display. It is more elegant to define the concept once and then use the Display definition whenever it is needed in the specification.

The effect of the Minus_Page transform is to display from the buffer the Screen_Size number of lines that immediately precede the lines being displayed. The only complexity in this Effect section is determining whether there are enough lines preceding to fill up the screen. This is the condition

```
Top_Line - Screen_Size > 0
```

If there are, Top_Line is set to Top_Line - Screen_Size. Otherwise, Top_Line is set to the first line in the buffer, and the first page of the buffer will be displayed. Using the new value of Top_Line to determine what lines are to be displayed, the second conjunct specifies that these lines of the buffer are to be marked as reviewed (N"Reviewed(i) = True). The final conjunct specifies exactly which line will be displayed at each position on the screen. The Display definition specifies whether the lines are displayed with nonprinting characters expanded.

The other five review transforms are defined similarly.

When developing the detailed design, the design team decided that it wanted all of the SRT's operations to be single keystrokes (to correspond to function keys on the terminal). As a result, none of the transforms should be parameterized. Because the Connect_To transform in the preliminary-design specification was parameterized by connection level, this raised the question of whether the preliminary-design specification should be modified to change the Connect_To transform to two separate transforms.

When using the Formal Development Methodology approach, you write code directly from the lowest-level specification. In addition, the Formal Development Methodology lets an upper level transform be implemented by one or more lower level transforms, and this correspondence can be based on the value of a parameter of the upper level transform. That is, upper level transforms are abstract representations of one or more lower level transforms, and they themselves may not directly correspond to a user interface operation. (This differs from a pure abstract-data-type approach where the highest-level specification defines the user interface.) Therefore, the team decided that there was no need to redo the preliminary-design specification.

Thus, in the detailed-design specification, there are two connection transforms. The mapping for the Connect_To transform reflects this:

```
Connect_To(Lev) ==
  (Lev=High & Connect_To_High
  | Lev=Low & Connect_To_Low)
```

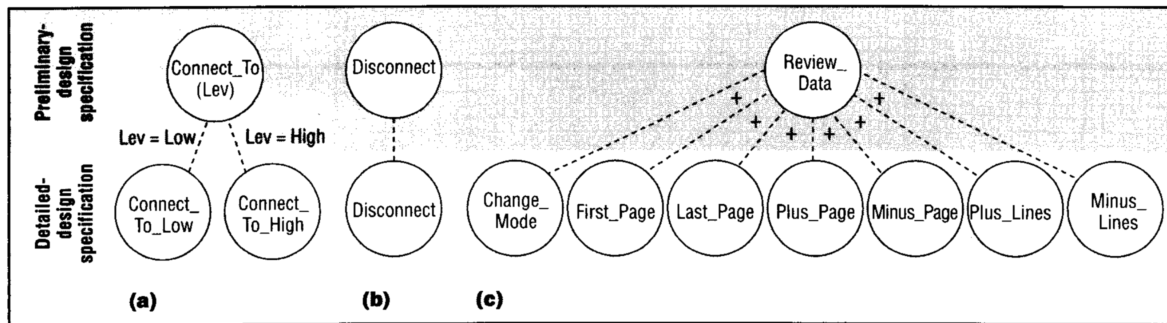


**Figure 5. (a)** Conditional, **(b)** one-to-one, and **(c)** disjunctive-transform mappings of preliminary-design specifications (top) to detailed-design specifications (bottom).

The mapping for the Connect_To transform of the preliminary design states that this transform with its parameter equal to High is refined to the Connect_High transform in the detailed-design specification and, when its parameter is equal to Low, it refines to the Connect_Low transform.

Figure 5 shows three forms of transform mapping used in the detailed-design specification.

The idea of what it means for the buffer to be reviewed was also clarified at this stage and was reflected in the detailed-design specification. The team considered different strategies, but eventually we determined "display all lines on the screen before an accept decision can be permitted" to be the most acceptable criterion for a buffer to be considered as reviewed. This decision was documented in the mapping of the state variable Reviewed:

```
Reviewed ==
  ∀i:Buffer_Number (
    i ≤ File_Size → Reviewed(i))
  & File_Size ≠ 0
```

which specifies that the buffer must be nonempty and that all of the buffer's lines must be marked as reviewed. This is also reflected in the Refcond section of the Accept transform:

```
~Connected
& ∀ i:Buffer_Number (
    i ≤ File_Size → Reviewed (i))
& File_Size > 0
```

Because Reviewed is a key concept in the formal critical requirements, this mapping had to be carefully analyzed to assure that it was defined accurately.

An example of another form of nondeterminism in Ina Jo is the result of wanting to include a nonsecurity requirement in the detailed-design specification. The requirement was for the data to be viewable during a receive operation. This was not a security requirement but a functional requirement. What was desired was to let an implementation display the file being transferred. Because the design team had not decided whether the screen would be changed and, if it were to be changed, how it would change (and because this is not relevant to security), the team thought that the value of the screen should not appear in the transform specification.

However, in Ina Jo, if a state variable is not mentioned in a transform specification's Effect section, it is assumed to be unchanged. Because there was to be the possibility of change, this did not suffice. The result was to formally specify the effect on the screen:

```
N"Screen = N"Screen.
```

This expression specifies that the new value of the screen is equal to its new value. That is, the new value of the screen could be equal to any value, thus letting the implementation change the value of the screen or to leave it unchanged during the receive operation.

At first, it may appear that the need to express N"Screen=N"Screen this way indicates that it was a bad Formal Development Methodology design decision to assume that variables that are not mentioned in a transform are unchanged. However, the alternative approach of having to explicitly express in every transform all variables that don't change quickly convinced us that the decision was a wise one. For example, for the complete detailed-design specification, there are 12 state variables and 16 transforms, and the average number of variables changed in each transform is less than four.

**Verification.** The appropriate proof obligations for the SRT detailed-design specification were also generated by the Ina Jo processor. These proof obligations guaranteed that the detailed-design specification was a correct refinement of the preliminary-design specification.

In the Formal Development Methodology, the proof obligations between any two levels of specification always have the same form. First, a proof obligation is generated that guarantees that the initial condition specified in the lower level specification implies the initial condition of the parent specification. In addition, for each transform in the upper level specification, a proof obligation is generated to guarantee that it is correctly refined in the child specification.

Because the lower level specification contains more details (since it is less abstract) than the higher level specification, you must use the lower level specification's mapping specification to rigorously transform the upper level types, constants, and variables to the entities of the lower level specification.

The interlevel proof obligations generated by the Ina Jo processor are an initial-condition mapping theorem and transform-mapping theorems.

The initial-condition mapping theorem is

$$\mathrm{INIT}_{low} \rightarrow \mathrm{map}(\mathrm{INIT}_{high})$$

where $\mathrm{INIT}_{low}$ is the Initial clause of the lower level specification and $\mathrm{INIT}_{high}$ is the Initial clause of the higher level specification. The Map function in this formula denotes the application of the mappings defined in the lower level specification to the types, constants, and variables that appear in the expression to which it is applied.

The transform-mapping theorem is

$$R_{low} \,\&\, E_{low} \rightarrow \mathrm{map}(R_{high}) \,\&\, \mathrm{map}(E_{high})$$

where $R_{low}$ and $E_{low}$ are the Refcond and

Effect of the low-level transform and $R_{high}$ and $E_{high}$ are the Refcond and Effect of the parent transform.

If the mapping for the parent transform is mapped to several transforms, this mapping is reflected in the mapping theorem generated. For example, in the SRT's detailed-design specification, the disjunctive mapping for the Connect_To transform generates the proof obligation

$$\begin{aligned}&(\text{Lev=High} \mathbin{\&} R_{\text{Connect_High}} \mathbin{\&} E_{\text{Connect_High}}\\ &\mid \text{Lev=Low} \mathbin{\&} R_{\text{Connect_Low}} \mathbin{\&} E_{\text{Connect_Low}})\\ &\quad\rightarrow \text{map}(R_{\text{Connect_To}}) \mathbin{\&} \text{map}(E_{\text{Connect_To}})\end{aligned}$$

This is equivalent to the proof obligations

$$\begin{aligned}&\text{Lev=High} \mathbin{\&} R_{\text{Connect_High}} \mathbin{\&} E_{\text{Connect_High}}\\ &\quad\rightarrow \text{map}(R_{\text{Connect_To}}) \mathbin{\&} \text{map}(E_{\text{Connect_To}})\end{aligned}$$

and

$$\begin{aligned}&\text{Lev=Low} \mathbin{\&} R_{\text{Connect_Low}} \mathbin{\&} E_{\text{Connect_Low}}\\ &\quad\rightarrow \text{map}(R_{\text{Connect_To}}) \mathbin{\&} \text{map}(E_{\text{Connect_To}})\end{aligned}$$

By showing that the lower level initial conditions correctly implement the higher level initial conditions and that the lower level transforms correctly implement the higher level transforms, it follows that the lower level specification preserves the criteria and constraints. Because the lower level specification is consistent with the level above it and the level above is consistent with the formal critical requirements, you can conclude that the lower level specification is consistent with the formal critical requirements.

For the SRT, the higher level specification is the preliminary-design specification and the lower level specification is the detailed-design specification. These specifications were input to the Ina Jo processor to generate the necessary proof obligations to assure that the detailed-design specification was consistent with the preliminary-design specification. The resulting theorems were proved with the Interactive Theorem Prover.

The theorems generated by the Ina Jo processor have been simplified here for clearer presentation.

## Code-level verification

After formally verifying the detailed-design specification, the design team decided to implement the SRT directly from the detailed-design specification. The alternative would have been to develop an even more detailed design specification. This specification would probably have added the concept of characters, possibly with a line being a list of characters and a buffer being a list of lists of characters.

At one point, the team considered allowing files with lines longer than the width of the screen. If this had been done, it would have been desirable to have SRT commands to move the screen right and left. This would be best handled by working at the character level of detail, and a third level of design specification would be reasonable. However, when the team decided not to include the horizontal-movement commands, it concluded that a third level of formal specification was not necessary.

***After formally verifying the detailed-design specification, the design team decided to implement the SRT directly from it. The alternative would have been to develop an even more detailed design specification.***

The prospect of code-level verification posed additional problems for the team. Among these was the choice of an appropriate implementation language.

At the time the SRT was being developed, a verification-condition generator for the Modula-1 language had been started and was being completed. Modula was initially chosen for another project because it was well structured, permitted high-level-language access to specific registers and memory locations for Modula-level I/O, and had a scoping approach that facilitated the formal verification.

Another advantage inherent in Modula but not in the more widely available Pascal was the small runtime package. The design team felt that the verification of the estimated 1,000 lines of high-level-language SRT code made little sense if thousands of bytes of runtime package were to be accepted as part of the language package but not subjected to any formal development. With Modula, the runtime package is a few hundred bytes of code. This was sufficiently small for manual generation and proof of the verification conditions. However, the choice of Modula or even Modula-2 for any microprocessor other than a select few was limited by the lack of available compilers.

For the SRT, the target microprocessor was the Intel 8086, for which we could get no Modula compiler. A further concern was the desire to avoid writing new compilers for future projects that also required code-level verification. While this project targeted an 8086, the next might target a Z8000, and the next some other processor. The design team wanted to establish tools that could be used by future projects and not be discarded at the completion of the SRT project.

The solution chosen was the construction of a Modula-to-C translator. C is widely available on various processors and is a good system-programming language, although not particularly well oriented to formal verification. The Modula language provided the necessary structuring and modularization to C. Because the team contemplated only higher-level-language verification, it felt that there was little security disadvantage in going from Modula to C to 8086 assembly code compared to going directly from Modula to 8086 assembly.

In neither case was compiler correctness to be assured in any way other than through test and observation. Thus, the team viewed this approach as logically equivalent to a Modula-to-8086 compiler with the added advantage that it provided a transportable approach to carry higher-level-language verification to a wide variety of processors.

For the formal verification process to be complete, you should formally verify the code against its formal specification. Unfortunately, the project funding the Modula verification-condition generator was cancelled, so an appropriate generator was no longer available. Therefore, we had to find an alternative assurance method.

What we finally used was a manual comparison of the specification to the code. The correspondence method used was a

rigorous technique developed at System Development Corp. for other security projects.[9] The SRT was particularly amenable to this approach because the code was written directly from the lowest-level formal specification, rather than being developed as a parallel but separate effort.

The formal specifications provided a more coherent design, which was reflected in the production of less error-prone code. The only major problem experienced in code debugging was in completely understanding the hardware interfaces.

## Effects of verification

The formal specification process placed additional constraints on the design because verification was the objective.

During the SRT's preliminary design, the design team made some calculations to determine the worst-case time delay to read in a file from a host. The time was about five minutes, which the team agreed was too long to require a security officer to wait if it was determined that the wrong file was being read in or that higher priority work should take precedence over the ongoing activity. As a result, the team decided that the security officer should be able to view the data as it is being received and that he should be able to interrupt the receive operation in the middle of a transfer.

Formally specifying this meant formally stating that the result of the receive operation was to have the new value of the SRT's buffer equal to the old value of the host's buffer up to some point. (This could be any one of many values, depending on when the security officer chose to interrupt and how frequently interrupts were polled.) It also meant having the rest of the SRT contain its previous value. Formally stating the possible combinations was difficult and formally verifying would have been even worse.

For some time, the design team argued that having to specify what the resulting value of a receive operation would be was unnecessary because the buffer was to be reviewed before release. This was a question of what the system's security perimeter was — what parts of the system are relevant to security. However, although this requirement was not absolutely necessary for specifying the SRT, the team decided it would be useful and even necessary for future secure-release terminals or if the total system using trusted hosts was to be formally specified and verified.

For example, a system using trusted hosts might have some preliminary security checks performed in the trusted host, marking the text to be transferred appropriately. The security officer would then use these markings to determine whether to release the text. Thus, it is relevant to security that the integrity of the markings be preserved. By guaranteeing that what was received by the SRT terminal is identical to what was in the host buffer, the integrity of the security markings is guaranteed.

The solution that the team chose was to have the SRT sanitize its entire buffer if the security officer interrupted file transfer. Thus, the preliminary-design specification stated that either the new value of the SRT's buffer was equal to the old value of the host's buffer or it was sanitized. Therefore, there were only two possible results rather than an infinite number of possibilities. This solution was both simple to implement and to specify and verify.

It also provided a convenient and consistent approach to handling persistent transmission failures when a reliable protocol was later designed: After several unsuccessful retransmission attempts, the receive fails and the buffer is sanitized.

Furthermore, the implementation is efficient and causes no long delays because the sanitization is a local operation requiring no I/O.

**Although this is a small system, the design team felt that the results would scale up if the formal design process were tightly integrated with the overall system design and implementation process, as was true in the SRT project.**

The SRT is a real system. The SRT was originally implemented in Modula targeting a Digital Equipment Corp. PDT/11-110 intelligent terminal and in Pascal targeting a Burroughs B20 workstation to demonstrate the user interface. Full functionality was provided. The workstation version was first implemented in Pascal to avoid waiting for completion of the Modula-to-C translator; it ran under CTOS (although the code made no operating-system calls). This code was hand-translated to C and a downline loader was implemented to let the output of the C cross-compiler overwrite the operating system and to run as a stand-alone process.[10]

The SRT was successfully demonstrated transferring files between various architectures and operating systems, including a DEC PDP-10 running Tenex and several versions of Unix running on DEC and Burroughs architectures. A later version of the SRT was implemented in PL/M on a Burroughs B25.

Three questions often asked about a project like this are:

- How does the approach scale up?
- What kind of experience is necessary to use the approach?
- How useful was the tool support?

The SRT is an admittedly small coding effort (about 1,000 lines of code). The preliminary-design specification is 130 lines of Ina Jo specification, and the Interactive Theorem Prover produced 33 pages of proof log output. The detailed design is 270 lines of specification, and the theorem prover produced 88 pages of output.

Although this is a small system, the design team felt that the results would scale up if the formal design process were tightly integrated with the overall system design and implementation process, as was true in the SRT project. Since the SRT project, others have used the Formal Development Methodology on several larger projects, including a multilevel secure distributed network with end-to-end encryption, a secure operating system, and a multilevel secure LAN.

The three members of the design team had varying levels of experience with for-

mal methods before the SRT project. I had the primary responsibility for writing the Ina Jo formal-design specifications, since I had extensive experience writing formal specifications before the SRT project and had taught classes on the use of the Formal Development Methodology. Another team member had taken a class on formal methods that included the Formal Development Methodology and had also worked with Ina Jo specifications. The third member had had only limited exposure to Ina Jo specifications.

Clearly, the SRT design team had more experience with formal specification and verification technology than is normally true for developers. This does not mean that the integrated approach we propose is inappropriate for the average development team. But it does suggest that developers should be trained in the use of formal methods.

Because the formal specifications were carefully analyzed by the design team before any proofs were tried, very few logical errors were discovered during the proof process. In fact, most of the proof obligations were proved the first time. By contrast, the Ina Jo processor was very useful for detecting syntax errors in the specifications. These were usually missing type declarations or simple typos, and Ina Jo's strong typing revealed them.

As is true with most existing formal-verification tools, the Formal Development Methodology tool suite was developed as a research tool and is not production-quality. However, even without production-quality tool support, the integration of formal methods into the development process is beneficial because, by using a formal notation for the design documents, developers can reason rigorously about their designs.

The primary benefit of formally specifying and verifying software comes in the form of more reliable systems. The programs developed using this approach perform the desired functions with fewer errors and may be trusted to operate correctly in critical environments. In addition, because errors that go undetected until the software is operational are usually vastly more expensive to fix compared to those revealed during the design phase, the cost of development is reduced by locating and eliminating errors early in the development process.

Formal methods should be integrated into the development process to increase assurance that critical systems will perform as desired. With this approach, the formal specification and verification effort does not occur after the system has been built. Neither does it take place as a parallel effort performed by a separate team. There is only one team — the development team — and it uses formal specifications as its design notation.

By using formal specifications for the design notation, the team can reason rigorously about its designs. In addition, because critical system requirements are captured in a mathematical notation, the system specification can serve as an unambiguous arbitrator of the system's desired properties. Furthermore, properties can be proved about the design, which gives assurance early in the development process that the system being developed will satisfy its critical requirements. ❖


## Acknowledgments

It is a pleasure to acknowledge the contributions of Tom Hinke and Jose Althouse, who were the other members of the SRT design team. Thanks also to Clark Weissman of Unisys, who originally suggested this project as a reasonable first application of the Formal Development Methodology to the entire development process.


## References


1. J. Scheid and S. Holtsberg, "Ina Jo Specification Language Reference Manual," tech. report, Formal Methods Group, Unisys, Culver City, Calif., May 1989.
2. W.R. Bevier, W.A. Hunt, Jr., and W.D. Young, "Toward Verified Execution Environments," *Proc. 1987 IEEE Symp. Security and Privacy*, CS Press, Los Alamitos, Calif., 1987, pp. 106-115.
3. C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, No. 4, 1972, pp. 271-281.
4. J. Guttag, E. Horowitz, and D. Musser, "Abstract Data Types and Software Validation," *Comm. ACM*, Dec. 1978, pp. 1,048-1,064.
5. T.A. Berson, R.J. Feiertag, and R.K. Bauer, "Processor-per-Domain Guard Architecture," *Proc. 1983 IEEE Symp. Security and Privacy*, CS Press, Los Alamitos, Calif., 1983, p. 120.
6. D.V. Schorre et al., "The Interactive Theorem Prover (ITP) Reference Manual," tech. report, Formal Methods Group, Unisys, Culver City, Calif., Nov. 1988.
7. J. Guttag and J.J. Horning, "Formal Specifications as a Design Tool," *Proc. Seventh Ann. ACM Symp. Princ. Programming Languages*, ACM, New York, 1980, pp. 251-261.

**Richard A. Kemmerer** is a professor in the Computer Science Dept. at the University of California at Santa Barbara. He has been a visitor at the Massachusetts Institute of Technology, Wang Institute, and the Politecnico di Milano. His research interests include formal specification and verification, reliable software, and secure systems.

Kemmerer received a BS in mathematics from Pennsylvania State University and an MS and PhD in computer science from the University of California at Los Angeles. He is a senior member of the IEEE Computer Society and a member of the ACM and the International Association for Cryptologic Research.

Address questions about this article to the author at Computer Science Dept., University of California, Santa Barbara, CA 93106.