Testing Formal Specifications to Detect Design Errors

RICHARD A. KEMMERER, MEMBER, IEEE

Abstract—Formal specification and verification techniques are now used to increase the reliability of software systems. However, these approaches sometimes result in specifying systems that cannot be realized or that are not usable. This paper demonstrates why it is necessary to test specifications early in the software life cycle to guarantee a system that meets its critical requirements and that also provides the desired functionality. Definitions to provide the framework for classifying the validity of a functional requirement with respect to a formal specification are also introduced. Finally, the design of two tools for testing formal specifications is discussed.

Index Terms—Design and development, formal verification, reliable software, requirements, specification, testing.

INTRODUCTION

THE desire to build reliable software has resulted in the use of formal specification and verification techniques to guarantee the correctness of the system being built [1]-[4]. Formal verification demonstrates that an implementation is consistent with its requirements. The problem of demonstrating consistency is approached by decomposing it into a number of easier problems. The requirements, which are usually an English statement of what is desired, are first stated in precise mathematical terms. This is known as the *formal model* or *criteria* for the system. This formal model expresses the critical requirements for the system. For example, for a security system the criteria could be that information at one security level does not flow to a lower security level. Next, a high level formal specification of the system is stated. This specification gives a precise mathematical description of the behavior of the system omitting all implementation details, such as resource limitations. This is followed by zero or more less abstract specifications which implement the next higher level specification with a more detailed level of specification. Finally, the system is coded in a high order language (HOL). This HOL implementation must be shown to be consistent with the original critical requirements.

It should be evident that demonstrating that HOL code is consistent with critical requirements is a difficult process. The process is made tractable by verifying the design at every step (see Fig. 1).

The first step of the verification process is to informally verify that the formal model properly reflects the critical requirements. This is the only informal step in the process. Since the formal model is at a high level of abstraction and contains no unnecessary details, it is usually a simple task to review the formal model with the persons who generated the requirements and determine whether the model properly reflects the critical requirements. Next it is necessary to prove that the highest level specifications are consistent with the formal model. This approach differs based on whether the specifications are presented using a state machine approach [5] or an algebraic approach [6]. The method presented in this paper uses the state machine approach, which specifies the effect of performing each operation based on certain entry conditions being satisfied. That is, for each operation there are entry and exit conditions, and if the state of the machine before the operation is invoked satisfies the entry conditions, then the state after the operation completes execution will satisfy the exit conditions. When using the state machine approach one must verify that the initial state satisfies the formal model and that every operation preserves the model.

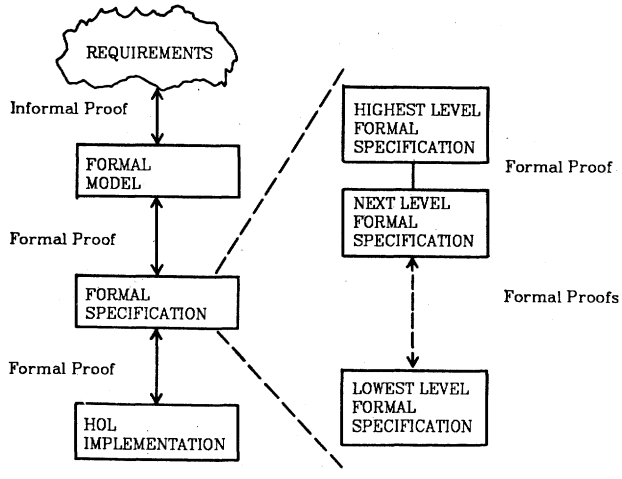


Fig. 1. Formal verification hierarchy.

After the highest level formal specification has been shown to be consistent with the formal model, it is necessary to 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, it is necessary to show that the HOL implementation is consistent with the lowest level specification.

Since each level of specification is shown to be consistent with the level above, and the HOL implementation is shown to

Manuscript received January 11, 1983; revised March 10, 1984. This work was supported in part by the National Science Foundation under Grant ECS81-06688.

The author is with the Department of Computer Science, University of California, Santa Barbara, CA 93106.

0098-5589/85/0100-0032$01.00

be consistent with the lowest level, by induction the HOL implementation is consistent with the highest level specification. In addition, since the highest level specification was shown to satisfy the formal model of the critical requirements, the implementation satisfies the formal model.

There is a problem with this approach: although the specification satisfies the correctness criteria, there may be no implementation that is consistent with the specification and at the same time provides the desired functionality. The real disaster is that this is usually not discovered until the design has gone through several levels of refinement, with each level being formally verified, and the implementation is in progress or completed. The result is a "yo-yo" effect where the designer goes back to the top level and rewrites the specification to allow an implementation that provides the desired functionality while preserving the correctness criteria.

This "yo-yo" effect is costly and time consuming, particularly where proofs have to be redone because the specification has changed [7]. An approach to reducing the "yo-yo" effect is to test the specifications to see if they allow the desired functionality, particularly for special cases. For instance, one might test what the result of performing a particular sequence of operations would be. This can be achieved by executing some test cases to see if the desired results are obtained. The problem is that most specification languages are nonprocedural. This paper considers two approaches to solving this problem. The first is to convert the nonprocedural specifications into a procedural form. This procedural form then serves as a rapid prototype to use for testing. The other approach is to perform a symbolic execution of the sequence of operations and check the resultant symbolic values to see if they define the desired set of resultant states.

In this paper the term functional requirements is used in a nonstandard way. That is, some of the functional requirements may not be known at design time. In fact, some functional requirements may arise during the testing of the rapid prototype. Although this is different from the prescribed software engineering approach, it more accurately reflects the way large software systems are built.

In the next section some definitions for classifying the validity of functional requirements with respect to a formal specification are presented. An example system with critical and functional requirements is then presented. Next, a particular nonprocedural specification language is considered and a specification of the example system is given. Using this formal specification, the functional requirements for the example are both symbolically executed and translated to a procedural form. Finally, the design of two tools for testing specifications are presented and their strengths and weaknesses are discussed.

Validity of Functional Requirements

The advantage of using a nonprocedural specification language is that during the design stages of the software life cycle no commitment is made to the order in which the constituent parts of an operation are to be performed. This allows the implementor to choose the order that is most efficient in terms of time or space. When converting a nonprocedural specification to a procedural form that can be executed, only one possible implementation is being considered. Therefore, if the desired functionality is provided by the resulting implementation, this does not guarantee that all implementations will provide the desired functionality. In the following paragraphs formal definitions make this distinction explicit.

A functional requirement F consists of a start predicate **Fstart,** a sequence of operations **SEQ,** and a resultant predicate **Fresult.** The start predicate defines the set of states of the system from which the sequence of operations can be invoked. Each operation is to be invoked in the order that it appears in the sequence and with the specified actual parameters. The resultant predicate defines the set of states that satisfy the desired result after executing the sequence of operations. The start predicate, sequence of operations, and resultant predicate may contain free identifiers. Thus, the functional requirement is actually a schema representing all uniform value assignments to these identifiers. Note that the start predicate may be identically true, indicating that the sequence of operations may be invoked from any state or it may be so precisely restricted that only one or no state satisfies its constraints. It may be useful to the reader to think of a functional requirement as a test case. The following notation is used in the definitions.

Let S be a formal specification.

Let $I(S)$ be an implementation of specification S.

Let IMP(S) be the set of all possible implementations of S.

Let STATES(I) be the set of all possible states for implementation I.

Each state in STATES(I) is uniquely determined by the values assigned to the state variables of implementation I.

A particular functional requirement is **satisfiable** with respect to a given specification if there is some implementation of the specification that gives the desired functionality.

Definition 1: If F is a functional requirement for a system formally specified by S, then F is **satisfiable** with respect to S iff

$$(\exists I, I \in \text{IMP}(S))(\forall P, P \in \text{STATES}(I))$$

$$(\text{Fstart}(P) \rightarrow \text{Fresult}(\text{SEQ}(P))).$$

A functional requirement is **unsatisfiable** if it is not satisfiable; i.e., if none of the possible implementations of the specification gives the desired functionality.

Definition 2: If F is a functional requirement for the system formally specified by S, then F is **unsatisfiable** with respect to S iff

$$(\forall I, I \in \text{IMP}(S))(\exists P, P \in \text{STATES}(I))$$

$$(\text{Fstart}(P) \& \sim \text{Fresult}(\text{SEQ}(P))).$$

Finally, a functional requirement is **valid** with respect to a given specification if every possible implementation of that specification gives the desired functionality.

Definition 3: If F is a functional requirement for the system formally specified by S, then F is **valid** with respect to S iff

$$(\forall I, I \in \text{IMP}(S))(\forall P, P \in \text{STATES}(I))$$

$$(\text{Fstart}(P) \rightarrow \text{Fresult}(\text{SEQ}(P))).$$

Clearly, it is desirable to have all the functional requirements be *valid* with respect to the specification. This says that no matter how the implementor chooses to implement the specification, if the implementation is consistent with the specification then the desired results will be obtained. One way of guaranteeing this is to have the specifications contain all of the desired functional requirements. However, by taking this approach the specifications get too cumbersome and the verification too costly when dealing with real systems.

A more reasonable approach is to have the specification define the minimal critical requirements and to test whether the desired functional requirements are *satisfiable* by executing the functional requirements on the specifications. Two methods for executing functional requirements are considered in this paper.

Before looking at an example it would be useful to discuss why functional requirements are defined to be satisfiable/unsatisfiable/valid with respect to a specification. Intuitively, a functional requirement is valid if it describes a user need, independent of the formal specification. Following this reasoning it would be more intuitive to say that it is desirable to have the specification be valid for all of the functional requirements. However, functional requirements as used in this paper may not all be known at design time. In fact, when testing a specification, test cases that are not necessarily going to be final requirements for the system may be tried. These are the "what would happen if" type test cases. Therefore, it is more reasonable to classify a functional requirement as being satisfiable/unsatisfiable/valid with respect to a formal specification.

An Example System

The example system considered in this paper is a university library database. There are two types of users of the database: normal borrowers and users with library staff status. The database transactions are as follows.

- Check out a copy of a book.
- Return a copy of a book.
- Add a copy of a book to the library.
- Remove a copy of a book from the library.
- Get a list of titles of books in the library by a particular author.
- Find out what books are currently checked out by a particular borrower.
- Find out what borrower last checked out a particular copy of a book.

These transactions have the following restrictions. A copy of a book may be added or removed from the library only by someone with library staff status. (In actual libraries only cataloging staff have the right to add or remove copies; the restriction here is a simplification.) Library staff status is also required to find out which borrower last checked out a copy of a book. A borrower without library staff status may find out only what books he or she has checked out. However, a user with library staff status may find out what books are checked out by any borrower.

The critical requirements that the database must satisfy at all times are as follows.

- All copies in the library must be either checked out or available for checkout.
- No copy of a book may be both checked out and available for checkout.
- A borrower may not have more than some predetermined number of books checked out at one time.
- A borrower may not have more than one copy of the same book checked out at one time.

Referring to the last two requirements as "critical" is a slight exaggeration since most libraries enforce these only informally.

As an example of a functional requirement consider the case where a particular author, Clyde Wroteit, has published three books, but the library has copies of only two. If a user were to invoke a transaction to get a list of titles of books by Clyde Wroteit presently in the library, then the result returned should be the titles of the two books that the library has in its collection. Otherwise, the functional requirement will not be satisfied although the critical requirements are.

Another more complex functional requirement considered in this paper concerns a borrower that currently has two less than the allowed number of books checked out, and attempts to check out three books which are not copies of any books that he currently has checked out, nor copies of each other. In attempting to check out the third book the transaction should not be successful since the borrower will have reached his book limit. The borrower then decides that he would rather have the third book than the first and returns the first book. He then again attempts to check out the third book. The result of this sequence of operations should be to have the borrower possess the second and third book and for the first book to be available for checkout. The borrower should also have his limit of books checked out.

In the following sections a formal specification for the example system is presented and it is shown how these functional requirements can be tested using the proposed specification testing approaches.

The Specification Language

The formal specification language that is used in this paper is a variant of Ina Jo,® which is a nonprocedural assertion language that is an extension of first-order predicate calculus. The language assumes that the system is modeled as a state machine. The key elements of the language are types, constants, variables, definitions, initial conditions, criteria, and transforms. A criterion is a conjunction of assertions that specify the critical requirements of a good state. A criterion is often referred to as a state invariant since it must hold for all states including the initial state. An Ina Jo language transform is a state transition function; it specifies what the values of the state variables will be after the state transition relative to what their values were before the transition was fired. A complete description of the Ina Jo language can be found in the Ina Jo Reference Manual [2].

Before giving a formal specification of the example system, a brief discussion of some of the notation is necessary. The

®Ina Jo is a trademark of System Development Corporation, a Burroughs Company.

following symbols are used for logical operations:

- & Logical AND
- | Logical OR
- ~ Logical NOT
- → Logical implication.

In addition there is a conditional form

(if A then B else C)

where A is a predicate and B and C are well-formed terms.

The notation for set operation is

- ∈ is a member of
- ∪ set union
- ~~ set difference
- $\{a, b \cdots c\}$ the set consisting of elements $a, b \cdots$ and c
- {set description} the set described by set description.

The language also contains the following quantifier notation:

- ∀ for all
- ∃ there exists.

Three other special Ina Jo symbols that may be used are

- N'' to indicate the new value of a variable (e.g., $N''v1$ is new value of variable $v1$)
- NC'' which indicates no change to the value of a variable
- T'' which defines a subtype of a given type T.

Specification of the Example System

Fig. 2 gives a top level Ina Jo specification for the example system. The state variable Library is the set of books currently in the library. Checked_Out indicates whether a particular book is checked out, and Responsible indicates which user last checked out a particular book. The number of books currently checked out by a particular user is indicated by the state variable Number_Out and the Boolean Never_Out is true if the book was never checked out. The state variables User_Result, Book_Result, and Title_Result are necessary to specify the results of the last three transforms. This is required because Ina Jo transforms are procedures and not functions (i.e., they cannot return a value).

The CRITERION expresses the critical requirements of the system that must hold in every state. The INITIAL section specifies that initially the library is empty and there are no books checked out to any users.

Symbolically Executing the Functional Requirements

When testing functional requirements, first one must determine how to state the restrictions placed on the start state in terms of the state variables. The first functional requirement presented earlier stated that if a particular author has published three books, but the library has copies of only two of these books, then a query for titles of books by the author should return only the titles of the two books in the library. A restatement of the start state for this requirement in terms of state variables might be: Author($b1$) = $a1$, Author($b2$) = $a1$, Author($b3$) = $a1$, $b1 \in$ Library, $b2 \in$ Library, and $\forall\, b5$:

```
Specification Library
LEVEL Top_Level

TYPE
   User,
   Book,
   Book_Title,
   Book_Author,
   Book_Collection = Set Of Book,
   Titles = Set Of Book_Title,
   Natural = T" i:Integer (i≥0)

CONSTANT
   Title(Book):Book_Title,
   Author(Book):Book_Author,
   Library_Staff(User):Boolean,
   Book_Limit:Natural,
   Copy_Of(B1:Book,B2:Book):Boolean =
         Author(B1) = Author(B2)
      & Title(B1) = Title(B2)

VARIABLE
   Library:Book_Collection,
   Checked_Out(Book):Boolean,
   Responsible(Book):User,
   Number_Books(User):Natural,
   Never_Out(Book):Boolean,

   User_Result:User,
   Book_Result:Book_Collection,
   Title_Result:Titles

DEFINE
   Available(B:Book):Boolean =
      B ∈ Library & ~Checked_Out(B),
   Checked_Out_To(U:User,B:Book):Boolean =
         Checked_Out(B)
      & Responsible(B)=U

CRITERION
      ∨b:Book(b ∈ Library →
                  (Checked_Out(b) & ~Available(b)
               |  ~Checked_Out(b) & Available(b)))
   & ∨u:User(Number_Books(u) ≤ Book_Limit)
   & ∨u:User,b1,b2:Book(
         Checked_Out_To(u,b1)
      & Checked_Out_To(u,b2)
      & Copy_Of(b1,b2)
         → b1=b2)
   INITIAL
      Library = Empty
   & ∨u:User (Number_Books(u) = 0)
   & ∨b:Book (~Checked_Out(b))

TRANSFORM Check_Out(U:User,B:Book) External
 Effect
  (if  Available(B)
   & Number_Books(U) < Book_Limit
   & ∨B1:Book (Checked_Out_To(U,B1) → ~Copy_Of(B,B1))
      then ∨U1:User (N"Number_Books(U1) =
            (if U1=U
                then Number_Books(U) + 1
                else Number_Books(U1)))
         & ∨B1:Book (
            (if B1=B
                then  N"Checked_Out(B)
                   & N"Responsible(B)=U
                   & ~N"Never_Out(B)
                else  NC"(Checked_Out(B1), Responsible(B1),
                           Never_Out(B1))))
      else NC"(Number_Books,Checked_Out,Responsible,Never_Out))

TRANSFORM Return(B:Book) External
 Effect
  (if Checked_Out(B)
     then ∨B1:Book (N"Checked_Out(B1) =
           (if B=B1
               then False
               else Checked_Out(B1)))
        & ∨U1:User (N"Number_Books(U1) =
           (if U1=Responsible(B)
               then Number_Books(U1) -1
               else Number_Books(U1)))
     else NC"(Checked_Out,Number_Books))

TRANSFORM Add_A_Book(U:User,B:Book) External
 Effect
  (if  Library_Staff(U)
   & B ∉ Library
     then  N"Library = Library ∪ {B}
        & ∨B1:Book (
            N"Checked_Out(B1) =
              (if B=B1
                  then False
                  else Checked_Out(B1))
           & N"Never_Out(B1) =
              (if B=B1
```

Fig. 2. Ina Jo specification of example system.

```
                then True
                else Never_Out(B1)))
      else NC"(Library,Checked_Out,Responsible,Never_Out))

TRANSFORM Remove_A_Book(U:User,B:Book) External
 Effect
  (if  Library_Staff(U)
   & Available(B)
      then N"Library = Library ~~ {B}
      else NC"(Library))
TRANSFORM Last_Responsible(U:User,B:Book) External
 Effect
  (if  Library_Staff(U)
   & B ∈ Library
   & ~Never_Out(B)
      then N"User_Result = Responsible(B)
      else NC"(User_Result))

TRANSFORM What_Checked_Out(Requester,Whom:User) External
 Effect
  (if (Library_Staff(Requester) | Requester=Whom)
     then ∀B1:Book (
          Checked_Out_To(Whom,B1) & B1 ∈ N"Book_Result
        | ~Checked_Out_To(Whom,B1) & B1 ∉ N"Book_Result)
     else NC"(Book_Result))

TRANSFORM Titles_By_Author(By_Whom:Book_Author) External
 Effect
  N"Title_Result = {T1:Book_Title ( ∃ B1:Book (
                   Author(B1)=By_Whom & Title(B1)=T1))}

END Top_Level
END Library
```

Fig. 2. (*Continued.*)

Book($b5 \in$ Library $\rightarrow \sim$Copy_Of($b5$, $b3$)). Next, a general statement of the desired resultant state in terms of state variables must be formulated. The complete functional requirement is

Fstart: $b1, b2, b3$: Book & $a1$: Author
& ~Copy_Of($b1, b2$) & ~Copy_Of($b1, b3$)
& ~Copy_Of($b2, b3$)
& Author($b1$)=$a1$ & Author($b2$)=$a1$ & Author($b3$)=$a1$
& ∀$b4$: Book (
Author($b4$)=$a1$ →
Copy_Of($b4, b1$)
|Copy_Of($b4, b2$)
|Copy_Of($b4, b3$)
& $b1 \in$ Library & $b2 \in$ Library
& ∀ $b5$: Book ($b5 \in$ Library → ~Copy_Of($b5, b3$))
SEQ: Titles_By_Author($a1$)
Fresult: Title_Result = {Title($b1$), Title($b2$)}

Note the free identifiers $b1$, $b2$, $b3$, and $a1$ that appear in Fstart, SEQ, and Fresult.

Now symbolic execution will be used to check the validity of this functional requirement with respect to the example specification, but first some more notation. Assume that the sequence of operations to be tested is started in some state q, and let var(q) denote the value of state variable var in state q. Furthermore, let succ(q) denote the state reached by applying the proper transform in the sequence to state q. Thus, if the sequence contains three transforms and the start state is p, then the resultant state would be succ(succ(succ(p))).

To symbolically execute a functional requirement on a specification the effect section of the transform being executed must be altered as follows. If a state variable appears in the effects section not preceded by the new value symbol N'', its current value is used to determine the effects of the transform. Therefore, each state variable not preceded by N'' is replaced by the value it has in the state where the transform is fired. Thus, if the sequence contains three transforms and the third transform contains state variable var1 not preceded by an N'', and the sequence was initiated in state p, then var1 is replaced by var1(succ(succ(p)). That is, the first transform resulted in state succ(p) and the second transform resulted in state succ(succ(p)), which is the state in which the third transform was fired.

In addition, each occurrence of a state variable var preceded by N'' is replaced by var(succ(q)), where q is the state in which this transform was fired.

Finally, any variable that does not appear in the effects section preceded by N'' is added to the result of firing the transform as var(succ(q)) = var(q). This occurs because the Ina Jo processor assumes that any variable that is not explicitly mentioned as changing remains unchanged, and it automatically appends N''var = var to the effect of the transform when processing the specification.

The result of symbolically executing transform Titles_By_Author($a1$) starting in state p is

Title_Result(succ(p)) = {$T1$: Book_Title (∃ $B1$: Book (
Author($B1$)=$a1$ & Title($B1$)=$T1$))}

and for all the other state variables their value in state succ(p) is equal to their value in state p.

Since p satisfies Fstart, it is known from the third line of the predicate defining Fstart that books $b1$, $b2$, and $b3$ have author $a1$ and from the fourth, fifth, and sixth lines it is known that the only other books that have $a1$ as their author are copies of $b1$, $b2$, or $b3$. Therefore, the set defined as the new value of Title_Result is

{Title($b1$), Title($b2$), Title($b3$)}.

Now for this resultant state to satisfy the desired resultant state it must be the case that

{Title($b1$), Title($b2$), Title($b3$)} = {Title($b1$), Title($b2$)}.

However, this is true only if Title($b1$) = Title($b3$) or Title($b2$) = Title($b3$). Expanding the second conjunct in the second line defining Fstart yields

~(Author($b1$)=Author($b3$) & Title($b1$)=Title($b3$))

which by moving the not inside is equivalent to

Author($b1$)~=Author($b3$)|Title($b1$)~=Title($b3$)

but by the third line of the predicate defining Fstart

Author($b1$)=$a1$ & Author($b3$)=$a1$

thus, Author($b1$)=Author($b3$). Therefore, it must be the case that

Title($b1$)~=Title($b3$)

Similarly, expanding the third disjunct in the second line defining Fstart yields

Title($b2$)~=Title($b3$).

Therefore, the two sets are not equal and the resultant state does not satisfy the desired result. That is, the functional requirement is not *valid* with respect to the example specification.

When symbolically executing a functional requirement, no assumptions are made about a particular implementation. Therefore, if the functional requirement is satisfied, then all implementations that are consistent with the specification will satisfy the requirement. Thus, the functional requirement is *valid* with respect to the specification. If the functional requirement is not satisfied by the specification, then the functional specification may be either *satisfiable* or *unsatisfiable* with respect to the specification. To show that it is *unsatisfiable* requires showing that Fstart can be true and that the resultant state implies ~Fresult, for then no implementation that is consistent with the specification can satisfy the functional requirement. The first functional requirement is not *valid* with respect to the sample specification because the restriction that $B1$ be an element of the library was not included. The correct effect for the Titles_By_Author transform is

N''Title_Result = {$T1$:Book_Title ($\exists$ $B1$:Book(
$B1 \in$ Library & Author($B1$)
= By_Whom & Title($B1$)=$T1$))}.

Now, the second functional requirement is considered. To demonstrate the symbolic execution of this functional requirement the succ notation is extended. If p is a predicate defining a set of states, then succ(p) defines the set of states that can be reached by applying the proper transform in the sequence to the set of states defined by p. The second functional requirement is expressed as follows.

Fstart: bk1, bk2, bk3:Book & us1:User
& ~Copy_Of(bk1, bk2) & ~Copy_Of(bk1, bk3)
& ~Copy_Of(bk2, bk3)
& Available(bk1) & Available(bk2) & Available(bk3)
& $\forall$ $b4$:Book (
Checked_Out_To(us1, $b4$) $\rightarrow$
~Copy_Of($b4$, bk1) & ~Copy_Of($b4$, bk2)
& ~Copy_Of($b4$, bk3))
& Number_Books(us1)=Book_Limit - 2

SEQ: Check_Out(us1, bk1), Check_Out(us1, bk2),
Check_Out(us1, bk3), Return(bk1),
Check_Out(us1, bk3)

Fresult: Number_Books(us1)=Book_Limit
& Checked_Out_To(us1, bk2)
& Checked_Out_To(us1, bk3)
& Available(bk1)

Symbolic execution is now used to check the validity of this functional requirement with respect to the example specification. The effect of symbolically executing Check_Out(us1, bk1) is

(if Available(bk1)
& Number_Books(us1) < Book_Limit
& $\forall$ $B1$:Book (
Checked_Out_To(us1, $B1$) $\rightarrow$ ~Copy_of(bk1, $B1$))
then
$\forall$ $U1$: User (N''Number_Books($U1$) =
(if $U1$=us1
then Number_Books(us1) + 1
else Number_Books($U1$)))
& $\forall$ $B1$: Book (
(if $B1$=bk1
then N''Checked_Out(bk1)
& N''Responsible(bk1)=us1
& ~N''Never_Out(bk1)
else NC''(Checked_Out($B1$), Responsible($B1$),
Never_Out($B1$))))
else NC''(Number_Books, Checked_Out, Responsible,
Never_Out))

In Fstart bk1 is available and none of the copies that us1 has checked out are copies of bk1. Furthermore, us1 has two less than the limit of books checked out. Therefore, the conditional is satisfied and the effect of symbolically executing Check_Out(us1, bk1) in state Fstart is specified by the then clause.

Thus, succ(Fstart) is specified by

$\forall$ $U1$:User (Number_Books($U1$)(succ(Fstart)) =
(if $U1$=us1
then Number_Books(us1)(Fstart) + 1
else Number_Books($U1$)(Fstart)))
& $\forall$ $B1$:Book (
(if $B1$=bk1
then Checked_Out(bk1)(succ(Fstart))
& Responsible(bk1)(succ(Fstart))=us1
& ~Never_Out(bk1)(succ(Fstart))
else Checked_Out($B1$)(succ(Fstart))
=Checked_Out($B1$)(Fstart)
& Responsible($B1$)(succ(Fstart))
=Responsible($B1$)(Fstart)
& Never_Out($B1$)(succ(Fstart))
=Never_Out($B1$)(Fstart))))

and all other state variables have the same value that they had in state Fstart.

In particular, the predicate defining succ(Fstart) is

1 bk1, bk2, bk3:Book & us1:User
2 & ~Copy_Of(bk1, bk2) & ~Copy_Of(bk1, bk3) & ~Copy_Of(bk2, bk3)
3 & Available(bk2) & Available(bk3)
4 & bk1 $\in$ Library
5 & Checked_Out(bk1)
6 & Responsible(bk1)=us1
7 & ~Never_Out(bk1)
8 & $\forall$ $b4$:Book (
Checked_Out_To(us1, $b4$) $\rightarrow$
~Copy_of($b4$, bk2) & ~Copy_Of($b4$, bk3))
9 & Number_Books(us1)=Book_Limit - 1

Lines 1 and 2 of the predicate follow directly from Fstart since they are type and constant declarations. Lines 5, 6, and 7 are a result of the effect of the transform. Line 9 is derived by substituting Book_Limit - 2 for Number_Books(us1)(Fstart). The derivations of lines 3, 4, and 8 are the most interesting. In Fstart Available(bk1) was true, and by expanding the definition of Available one derives

bk1 $\in$ Library & ~Checked_Out(bk1)

but the effect of the transform specifies Checked_Out(bk1) in the new state. Therefore, Available(bk1) is no longer true. However, bk1 is still an element of the library. This results in lines 3 and 4. The effect of the transform specifies that Checked_Out(bk1) & Responsible(bk1)=us1, but this is the definition of Checked_Out_To(us1, bk1). Now, by the definition of Copy_Of one can derive Copy_Of(bk1, bk1). That is, in succ(Fstart)

Checked_Out_To(us1, bk1)
& Copy_Of(bk1, bk1)

Therefore, ~Copy_Of($b4$, bk1) is simplified out of the consequent of the universally quantified expression that partially defined Fstart. The resulting expression is line 8.

Next Check_Out(us1, bk2) is symbolically executed from state succ(Fstart). As was the case with the previous transform bk2 is available, none of the books checked out by us1 are copies of bk2, and us1 has less than the limit of books checked out (i.e., Book_Limit - 1). Therefore, the condition is satisfied and the then clause specifies the state succ(succ(Fstart)) based on succ(Fstart).

The predicate defining succ(succ(Fstart))) is

1 bk1, bk2, bk3:Book & us1:User
2 & ~Copy_Of(bk1, bk2) & ~Copy_Of(bk1, bk3)
  & ~Copy_Of(bk2, bk3)
3 & Available(bk3)
4 & bk1 ∈ Library & bk2 ∈ Library
5 & Checked_Out(bk1) & Checked_Out(bk2)
6 & Responsible(bk1)=us1 & Responsible(bk2)=us1
7 & ~Never_Out(bk1) & ~Never_Out(bk2)
8 & ∀ $b4$:Book (
    Checked_Out_To(us1, $b4$) →
      ~Copy_Of($b4$, bk3))
9 & Number_Books(us1)=Book_Limit

The reasoning for deriving this state definition is identical to the reasoning for deriving state succ(Fstart).

The transform Check_Out(us1, bk3) is symbolically executed next starting in state succ(succ(Fstart)). Since Number_Books(us1) = Book_Limit, Number_Books(us1) < Book_Limit is not satisfied in this state. Since the conditional is false, the else clause defines the new state. However, the else clause specifies that there is no change to any state variable; therefore, succ(succ(succ(Fstart))) = succ(succ(Fstart)).

The next state transform to be symbolically executed is Return(bk1) starting in state succ(succ(succ(Fstart))). Since bk1 is checked out the conditional of the effect is satisfied and the then clause defines the new state as follows.

∀ $B1$:Book(Checked_Out($B1$)(succ(q)) =
  (if B=$B1$
    then False
    else Checked_Out($B1$)(q)))
& ∀ $U1$:User(Number_Books($U1$)(succ(q)) =
  (if $U1$=Responsible(bk1)(q)
    then Number_Books($U1$)(q) - 1
    else Number_Books($U1$)(q)))

where state q is succ(succ(succ(Fstart))).

The predicate defining state succ(succ(succ(succ(Fstart)))) is

1 bk1, bk2, bk3:Book & us1:User
2 & ~Copy_Of(bk1, bk2) & ~Copy_Of(bk1, bk3)
  & ~Copy_Of(bk2, bk3)
3 & Available(bk1) & Available(bk3)
4 & bk2 ∈ Library
5 & Checked_Out(bk2)
6 & Responsible(bk1)=us1 & Responsible(bk2)=us1
7 & ~Never_Out(bk1) & ~Never_Out(bk2)
8 & ∀ $b4$:Book (
    Checked_Out_To(us1, $b4$) →
      ~Copy_Of($b4$, bk3))
9 & Number_Books(us1=Book_Limit - 1

Since the effect of Return specifies ~Checked_Out(bk1) and because bk1 is an element of the library, Available(bk1) is true in the new state. Notice that Responsible(bk1)= us1 and ~Never_Out(bk1) are still present since Return does not modify these state variables.

Finally, Checked_Out(us1, bk3) is symbolically executed in state succ(succ(succ(succ(Fstart)))). Since bk3 is available, none of the books checked out to us1 are copies of bk3, and us1 has less than the limit of books checked out, the final state is defined by the then clause of the effect section of the Check_Out transform.

The final state is defined by

1 bk1, bk2, bk3:Book & us1:User
2 & ~Copy_Of(bk1, bk2) & ~Copy_Of(bk1, bk3)
  & ~Copy_Of(bk2, bk3)
3 & Available(bk1)
4 & bk2 ∈ Library & bk3 ∈ Library
5 & Checked_Out(bk2) & Checked_Out(bk3)
6 & Responsible(bk1)=us1 & Responsible(bk2)=us1
  & Responsible(bk3)=us1
7 & ~Never_Out(bk1) & ~Never_Out(bk2)
  & ~Never_Out(bk3)
8 & Number_Books(us1)=Book_Limit

The reasoning for arriving at this final state is the same as was used when discussing the first two transforms so it will not be repeated again.

Now it is necessary to see if the predicate defining the final state is strong enough to imply Fresult. The first conjunct of Fresult follows directly since it is identical to line 8 of the predicate defining the final state. To prove the second conjunct Checked_Out_To(us1, bk2) is expanded to yield

Checked_Out(bk2) & Responsible(bk2)=us1

but these also follow directly from lines 4 and 5 of the predicate defining the final state. The third conjunct of Fresult is proved similarly. Finally, the fourth conjunct is identical to the third line of the predicate defining the final state. Thus, the conjuncts of Fresult are true in the final state and the second functional requirement is *valid* with respect to the sample specification.

The simplifications outlined above are to be performed automatically by the symbolic execution tool that is described in a later section of this paper. Care must be taken when deriving the predicates that define the new states to assure that

the predicates are not weakened, resulting in information being lost. For instance, when manually deriving state succ(succ(succ(succ(Fstart)))) while symbolically executing the second functional requirement there was not enough information in the predicate defining the previous state to add the conjunct ~Copy_Of($b4$, bk1) to the conclusion of the quantified statement in line 8. The information needed was lost when deriving the predicate to define succ(Fstart). If line 8 of this predicate had been

$\forall\, b4$:Book (
  Checked_Out_To(us1, $b4$) & $b4 \sim= b1 \rightarrow$
    ~Copy_Of($b4$, bk1) & ~Copy_Of($b4$, bk2)
      & ~Copy_Of($b4$, bk3))

the information would not have been lost.

Using a Rapid Prototype to Test Functional Requirements

The rapid prototype approach to checking the satisfiability of the functional requirement converts the nonprocedural specification into a procedural program and runs test cases on that program. However, since the resulting program represents only one of many possible consistent implementations, a successful test shows only satisfiability. In the same manner, an unsuccessful test shows that the functional requirement tested is not *valid*, but does not indicate if it is *satisfiable* or not.

To convert a nonprocedural Ina Jo specification into a procedural Pascal-like program it is necessary to determine for each specification structure how it should be translated into code structure. For instance, equalities expressions not involving an N'' (Number_Books(U) < Book_Limit) become conditionals which must hold for any of the changes to occur. If the equalities expression contains a single state variable preceded by an N'', then it becomes an assignment statement. Thus, N''Responsible(B) = U becomes Responsible[B] := U. Since Ina Jo allows conditional expressions a target language that allows conditional expressions would make translation easier, but is not necessary. If the language does not allow conditional expressions, then the conditionals move out to an if statement. All definitions must also be expanded or declared as functions. The Appendix contains a Pascal translation of the Check_Out and Return transforms as well as the type, constant, and variable declarations that are necessary to execute these procedures.

The Check_Out transform is translated to

```
procedure checkout(u:user; b:book);
begin
  if (available(b) and (numbooks[u]<booklimit)
      and notinset(b, u))
    then
      begin
        numbooks[u] := numbooks[u] + 1;
        checkedout[b] := true;
        responsible[b] := u;
        neverout[b] := false
      end
end
```

Notice that none of the no change specifications are reflected in the executable code because no code is needed to implement a no change.

This example points out one problem with converting a formal specification to executable code. The condition

$\forall\, B1$:Book (Checked_Out_To($U, B1$) $\rightarrow$
  ~Copy_Of($B, B1$))

is defined on a possibly infinite (it is an unspecified type) domain Books. Since it is undesirable to have an infinite domain, the user must assign finite sizes to these domains. This may be accomplished by converting unspecified types into enumerated types (scalar types in Pascal). These enumerated types range from 1 to some maximum value which is specified by the user. Thus, the Ina Jo type Book becomes the Pascal enumerated type

```
const bookmax = 40;
type book = 1..bookmax;
```

In the executable code translation in the Appendix the function notinset implements the predicate discussed above. The universally quantified statement becomes a for loop which sets a Boolean false if the conditions being checked are not true for any value of book.

Type declarations other than unspecified types are translated to Pascal in a straightforward manner. Ina Jo Booleans and integers become Pascal Booleans and integers. Ina Jo enumerated types become Pascal enumerated types, and sets become sets. For example

```
type bookcollection = set of book;
```

An Ina Jo state variable that has a single parameter which is an unspecified type becomes a Pascal variable array indexed by the enumerated type corresponding to the unspecified type. Thus

```
VARIABLE Checked_Out(Book):Boolean
```

becomes

```
var checkedout[1..bookmax] of boolean;
```

State variables that have more than one parameter may become multidimensional variable arrays.

Ina Jo definitions may become Pascal functions. For instance,

```
DEFINE Available(B:Book): Boolean == B ∈ Library
    & ~Checked_Out(B)
```

becomes

```
function available(b:book): boolean;
begin
  available := ((b in Library) and not checkedout[b])
end
```

Functional requirements are tested on the executable code by trying different test cases that are instances of the functional requirement. An example of a specific instance of the second functional requirement to be tested is

Fstart: bookmax=40, usermax=20, booklimit=4,
copyof[1,2]=false, copyof[1,3]=false,
copyof[2,3]=false, copyof[2,1]=false,
copyof[3,1]=false, copyof[3,2]=false,
1 **in** Library, 2 **in** Library, 3 **in** Library,
checkedout[1]=false, checkedout[2]=false,
checkedout[3]=false,
checkedout[29]=true, checkedout[30]=true,
29 **in** Library, 30 **in** Library,
responsible[29]=5, responsible[30]=5,
copyof[1,29]=false, copyof[2,29]=false,
copyof[3,29]=false, copyof[29,1]=false,
copyof[29,2]=false, copyof[29,3]=false,
copyof[1,30]=false, copyof[2,30]=false,
copyof[3,30]=false, copyof[30,1]=false,
copyof[30,2]=false, copyof[30,3]=false,
numbooks[5]=2
SEQ: checkout(5, 1), checkout(5, 2), checkout(5, 3),
return(1), checkout(5,3)
Fresult: numbooks[5]=4,
checkedout[2]=true, responsible[2]=5,
checkedout[3]=true, responsible[3]=5,
checkedout[1]=false, 1 **in** Library

In this instance the three books to be checked out are 1, 2, and 3 and the user is 5. User 5 already has two books checked out and they are books 29 and 30.

The program in the Appendix initializes the variables as defined by Fstart and executes SEQ. The resulting values of the variables agree with Fresult. Therefore, this instance of the second functional requirement is *satisfiable* with respect to the specification.

Rapid Prototype Tool

The proposed rapid prototype tool is more than just a compiler, for in addition to translating the nonprocedural specification into a procedural target language that can be executed, it provides a testing environment. This testing environment allows the user to use several modes of operation to test his specification. One mode is to submit the specification to the tool which then compiles it into executable code which can be used as a rapid prototype to test the functionality. Another mode is to submit the specification along with the start state, sequence of operations, and resultant state and let the tool determine whether the functional requirement submitted is satisfiable with respect to the specification.

The user inputs to the rapid prototyping tool are

1) a formal system specification written in Ina Jo (or some other suitable specification language);

2) the start state of the system (i.e., values for the state variables) or the keyword DEFAULT;

3) A sequence of transforms to be executed or the keyword INTERACTIVE;

4) the expected resultant state of the system, a list of variables to be output, or the keyword ALL.

The tool then performs as follows.

1) Check the initial values to see that they are consistent with the criterion (i.e., is the system starting in a correct state?).

If the keyword DEFAULT is used, then the tool will assign initial values to each of the state variables such that the initial condition is satisfied.

In either case the initial values of all of the state variables are printed.

2) If a sequence of transforms is input the tool executes the procedures that correspond to each of the transforms in sequence on the compiled specification starting from the initial state of part 1.

If the keyword INTERACTIVE was input, the tool enters an interactive mode and requests directions from the user. The user may ask to have the current state variable values displayed, to change the values of state variables, to execute a procedure, to check whether the resultant condition is satisfied by the current state, or to quit testing. If the user changes the current values of the state variables, then before executing the next procedure a check is made to see if the new values of the state variables are consistent with the criterion.

After executing each procedure, whether in the interactive mode or executing an input sequence, the tool automatically checks whether the current values of the state variables satisfy the criterion. If they do not, no more procedures may be executed, and the current value of each of the state variables is printed.

3) If an expected resultant state is input the tool checks whether the final state satisfies the resultant predicate, by substituting the current values for each of the state variables in the predicate.

If a list of variables is input the names and values of each of these variables is printed.

If the keyword ALL is input all of the state variables and their current values are printed.

Note that if the tool gets a runtime error while executing a procedure, either input as part of a sequence or while in the interactive mode, it will immediately terminate execution and print the current values of all variables as well as the runtime error.

Symbolic Execution Tool

The proposed symbolic execution tool is much like the rapid prototyping tool. It too provides a testing environment allowing the user to use several modes of operation to test his specifications. Unlike the rapid prototyping tool, however, the specification is not compiled into executable code.

The user inputs to the symbolic execution tool are the same as for the rapid prototyping tool, except states are described by predicates not just by listing their contents.

The tool performs as follows.

1) Check the initial predicate to see that it is consistent with the criterion (i.e., is the system starting in a correct state?).

If the keyword DEFAULT is used, then the start state is assumed to be defined by the initial state of the specification.

In either case the predicate defining the start state is printed.

2) If a sequence of transforms is input the tool symbolically executes each of the transforms in sequence starting from the initial state of part 1.

If the keyword INTERACTIVE was input, the tool enters an interactive mode and requests directions from the user. The user may ask to have the predicate defining the current state displayed, to change the value of the predicate defining the current state, to symbolically execute a transform, to check whether the resultant predicate is satisfied by the current state, or to quit testing. If the user changes the predicate defining the current state, a check is made to see if the newly defined state is consistent with the criterion.

After each transform is symbolically executed the tool automatically checks whether the current state is consistent with the criterion. If it is not, no more transforms may be executed and the predicate defining the current state is printed.

3) If a predicate defining the expected resultant state is input the tool checks to verify whether the final state satisfies the resultant predicate.

If the keyword DEFAULT is input, then the predicate which represents the final state is printed.

Advantages/Disadvantages of the Rapid Prototype Approach

There are several problem areas in translating Ina Jo to executable code. The problem with translating universal and existential quantifiers was discussed in an earlier section: Ina Jo specifications often deal with infinite domains, and a likely part of a specification might be

$$\forall\, i\text{:Pos\_Integer}(N''\text{var}.i = \text{var}.i + i).$$

The resulting infinite for loop, however, would be difficult to test. The solution proposed for the example is to have the tester specify the range of the domain, thus converting the infinite domain to a finite one.

Another problem area arises because Ina Jo allows nondeterministic specifications. When using the Ina Jo language if a variable is to be changed under certain circumstances, but not others, all circumstances must be described explicitly. This is not enforced by the specification processor; therefore, the effect section of an Ina Jo transform may not be deterministic. For instance, a possible specification for the Remove_A_Book transform, which is incorrect, is

(if Library_Staff(U)
  & Available(B)
    then N''Library=Library $\sim\sim$ {B}).

The interpretation of this specification is that if user U has library staff privileges and book B is available, then the new value of the set library is equal to its old value with book B removed. However, it does not specify what the new value of the library will be if the condition is not met. That is, this specification is equivalent to the following specification.

(if Library_Staff(U)
  & Available(B)
    then N''Library=Library $\sim\sim$ {B}
    else N''Library=N''Library).

In the Ina Jo language the meaning of N''var=N''var is that the variable var can assume any value in the new state.

Two immediate solutions to this nondeterministic problem are: treat the variables as unchanged or treat it as getting a special undefined value. If the first solution is picked, this part of the specification will not show up in the executable code. That is, the particular implementation that does not change the value of the variable in question is used. The second solution more accurately reflects the specification; however, this solution requires a special undefined value for each type, which must be treated differently than all other values. In particular, two variables of the same type with undefined values should not be considered equal.

In addition to the translation problems, the rapid prototype approach suffers from all of the disadvantages of testing. For example, only one test case can be tested at a time, and unless all instances of a functional requirement are tested there is no guarantee that the functional requirement will be *satisfiable* for all instances. Also, if the right test case is not tried, an error will not be revealed.

Another problem that was discussed earlier is that using the rapid prototype approach at best guarantees satisfiability since it is only checking one implementation of the specification.

The previous paragraphs have outlined some of the disadvantages with the rapid prototype approach. There are two important advantages that can outweigh the disadvantages. First, a rapid prototype helps the specification writer to debug the specification. It also helps a potential user experience the capabilities of the system. It is often only through this type of experience that the necessary functional requirements can be discovered. Furthermore, it is better to have the user discover needs early in the software life cycle, not after the system has been completely implemented and delivered.

Another reason for having executable code is that although the result of symbolically executing a sequence of transforms contains the information that is necessary to determine whether a desired result is satisfied, it is often difficult to glean the pertinent information from the remainder.

Advantages/Disadvantages of Symbolic Execution

An advantage of symbolically executing a functional requirement is that it allows one to deal with infinite domains. Thus, universal and existential quantifiers cause no problems. It also lets the user test a large number of cases at one time.

The biggest problem with symbolic execution is that the predicates defining the current state can get unmanageable after executing a sequence of transforms, particularly when the transforms contain conditional expressions that can not be readily resolved. Consider three transforms $T1$, $T2$, and $T3$ defined as follows.

$T1$: (if A then B else C)
$T2$: (if D then E else F)
$T3$: (if G then H else I)

The effect of executing the sequence $T1$, $T2$, $T3$ is

(if G(if D(if A then B else C)
      then E(if A then B else C)
      else F(if A then B else C))
  then H(if D(if A then B else C)
      then E(if A then B else C)

else F(if A then B else C))
else I(if D(if A then B else C)
then E(if A then B else C)
else F(if A then B else C)))

This effect has been abbreviated, for if s is the start state, then every occurrence of

(if A then B else C)

should be replaced by

(if $A(s)$ then $B(s)$ else $C(s)$).

As was seen in the earlier examples the result can be simplified, and the proposed symbolic execution tool will do this automatically whenever possible. However, the examples that have been presented in this paper are simple examples used only for pedagogical purposes. Specifications for real systems tend to be more complex. Thus, the predicate that defines the resultant state after executing a sequence of these transforms is likely to be complex.

Conclusions

This paper has attempted to demonstrate the need for testing formal specifications early in the software life cycle to guarantee a reliable system that also provides the desired functionality. Formal definitions to classify the implementability of a functional requirement with respect to a specification were also presented. In addition, two tools for carrying out the process of testing specifications were proposed: a rapid prototype tool and a symbolic execution tool. The rapid prototype tool is the more difficult to implement, but it provides the user with a prototype which he can exercise to see if it meets his (sometimes fuzzy) functional requirements.

A preliminary symbolic execution tool, which accepts Ina Jo specifications and tests them under user direction has been built on a UNIX® operating system running on a VAX/750. The goal of this preliminary system was to get it working unintelligently; therefore, it performs very little simplification on the generated formulas. The rapid prototype tool has been designed but not implemented. Currently work on this project is concentrating on designing and implementing efficient decision procedures for automatically simplifying the complicated formulas generated by the symbolic executer. These simplification procedures will be integrated into the testing tool.

Appendix

Pascal Translation of Example Specification

```
program examplesys (input, output);
const bookmax = 40;     {number of books}
   usermax = 20;     {number of users}
   booklimit = 4;
type book = 1..bookmax;
   user = 1..usermax;
   bookcollection = set of book;
var library: bookcollection;
   checkedout, neverout: array[1..bookmax] of boolean;
   responsible: array[1..bookmax] of user;
   numbooks: array[1..usermax] of integer;
   copyof: array[1..bookmax, 1..bookmax] of boolean;
function available(b: book): boolean;
begin
   available := (b in library) and not checkedout[b]
end;
function notinset(b:book; u:user): boolean;
var
   i: integer;
   nocopy: boolean;
begin
   nocopy := true;
   for i:= 1 to bookmax do
      if checkedout[i] and (responsible[i] =u)
            and copyof[b, i]
         then nocopy := false;
   notinset := nocopy
end;
procedure checkout(u:user; b:book);
begin
   if available(b) and (numbooks[u] < booklimit)
            and notinset(b, u)
      then
         begin
            numbooks[u] := numbooks[u] + 1;
            checkedout[b] := true;
            responsible[b] :=u;
            neverout[b] := false
         end
end;
procedure return(b:book);
begin
   if checkedout[b]
      then
         begin
            checkedout[b] := false;
            numbooks[responsible[b]]
               := numbooks[responsible[b]] - 1
         end
end;
begin
   library:=[1··30];
   checkedout[1]:=false; checkedout[2]:=false;
            checkedout[3]:=false;
   checkedout[29]:=true; checkedout[30]:=true;
   responsible[29]:=5; responsible[30]:=5;
   copyof[1,2]:=false; copyof[1,3]:=false;
            copyof[2,3]:=false;
   copyof[2,1]:=false; copyof[3,1]:=false;
            copyof[3,2]:=false;
   copyof[1,29]:=false; copy[2,29]:=false;
            copyof[3,29]:=false;
   copyof[29,1]:=false; copyof[29,2]:=false;
            copyof[29,3]:=false;
   copyof[1,30]:=false; copyof[2,30]:=false;
            copyof[3,30]:=false;
```

®UNIX is a trademark of Bell Laboratories.

```
    copyof[30,1] := false; copyof[30,2] := false;
                copyof[30,3] := false;
    numbooks[5] := 2;
    checkout(5,1);
    checkout(5,2);
    checkout(5,3);
    return(1);
    checkout(5,3);
    write(numbooks[5], checkedout[2], responsible[2],
                checkedout[3],
            responsible[3], available(1))
end.
```


Acknowledgment

The author would like to thank J. L. Bruno and P. R. Eggert for their many useful criticisms and suggestions on an earlier version of this paper, and also S. T. Eckmann who implemented the preliminary symbolic execution tool.

References

[1] D. I. Good, "The proof of a distributed system in Gypsy," Inst. Comput. Sci., Univ. Texas at Austin, Austin, TX, Tech. Rep. 30, Sept. 1982.

[2] R. Locasso, J. Scheid, V. Schorre, and P. Eggert, *The Ina Jo Specification Language Reference Manual,* System Development Corp., Santa Monica, CA, SDC document TM-6889/000/01, Nov. 1980.

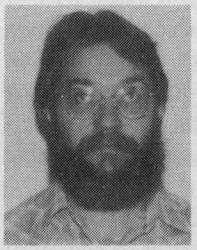
[3] L. Robinson, *The HDM Handbook, Vol. I: Foundations of HDM,* Comput. Sci. Lab., SRI International, Menlo Park, CA, June, 1979.

[4] *AFFIRM Reference Manual,* D. Thompson and R. Erickson, Eds., Univ. Southern California Information Sciences Inst., Marina Del Rey, CA, Feb. 1981.

[5] C. A. R. Hoare, "Proof of correctness of data representations," *Acta Inform.*, vol. 1, pp. 271–281, 1972.

[6] J. Guttag, E. Horowitz, and D. Musser, "Abstract data types and software validation," *Commun. ACM,* vol. 21, pp. 1048–1064, Dec. 1978.

[7] R. Kemmerer, "Status report on SDC's formal development methodology," in *Proc. 2nd Verification Workshop,* Gaithersburg, MD, Apr. 1981.



Richard A. Kemmerer (M'81) was born in Allentown, PA, in 1943. He received the B.S. degree in mathematics from the Pennsylvania State University, University Park, in 1966, and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles, in 1976 and 1979, respectively.

He is currently an Assistant Professor at the University of California, Santa Barbara. From 1966 to 1974 he worked as a programmer and systems consultant for North American Rockwell and the Institute of Transportation and Traffic Engineering at UCLA. His research interests include formal specification and verification, reliable software, and secure systems. He is author of the book *Formal Specification and Verification of an Operating System Security Kernel.*

Dr. Kemmerer is a member of the IEEE Computer Society and the Association for Computing Machinery, and he is currently the Vice Chairman of the IEEE Technical Committee on Security and Privacy.

The Eden System: A Technical Review

GUY T. ALMES, MEMBER, IEEE, ANDREW P. BLACK, EDWARD D. LAZOWSKA, AND JERRE D. NOE, SENIOR MEMBER, IEEE

Abstract—**The Eden project is a five year experiment in designing, building, and using an "integrated distributed" computing system. We are attempting to combine the benefits of integration and distribution by supporting an object based style of programming on top of a node machine/local network hardware base. Our experimental hypothesis is that such an architecture will provide an environment conducive to building distributed applications.**

This technical review is written three years into the project. We begin by summarizing the Eden system: its concepts, history, status, and context. We next discuss the way in which the task of supporting the Eden architecture is divided between the kernel, the programming language, and user-level (library) code; we describe the experiences with paper designs and prototype implementations that led us to this division of labor. We then show how distributed applications make use of various aspects of the Eden architecture. We conclude by providing some preliminary evaluations based on our experiences to date.

The objective of our research is to assess the benefits (in terms of programmability) and the costs (in terms of necessary support) of our system architecture. We feel we have gained insights on a number of questions of relevance well beyond Eden or Eden-like systems.

- **How should the job of supporting a system such as Eden be divided between the kernel, the programming language, and user-level (library) code?**
- **How do distributed applications make use of various aspects of the Eden architecture (location independence, concurrency, etc.)?**
- **Of the many design and implementation choices we have made, which are apparently good or apparently bad, based on our experience to date?**

Index Terms—**Capability, Concurrent Euclid, concurrent programming, distributed electronic mail, distributed program, distributed system, Eden, object-oriented system, remote procedure call.**

Manuscript received October 26, 1983; revised June 14, 1984. This work was supported in part by the National Science Foundation under Grant MCS-8004111. Computing equipment was provided in part under a cooperative research agreement with Digital Equipment Corporation.

The authors are with the Department of Computer Science, University of Washington, Seattle, WA 98195.