FORMAL METHODS FOR LIFE-CRITICAL SOFTWARE

Ricky W. Butler
Sally C. Johnson
NASA Langley Research Center
Hampton, Virginia

Abstract

The use of computer software in life-critical applications, such as for civil air transports, demands the use of rigorous formal mathematical verification procedures. This paper demonstrates how to apply formal methods to the development and verification of software by leading the reader step-by-step through requirements analysis, design, implementation, and verification of an electronic phone book application. The current maturity and limitations of formal methods tools and techniques are then discussed, and a number of examples of the successful use of formal methods by industry are cited.

Introduction

From civil air transports to nuclear power plants, computer software is finding its way into more life-critical applications every year. This paper examines the available methods for avoiding/tolerating design faults in software and makes a case that fault-avoidance techniques such as formal methods are the only intellectually defensible means for producing life-critical software. The characteristics of formal methods and how they may be applied to software are then described and demonstrated on an example application. Finally, the maturity of formal methods for practical use by industry is examined and some limitations of formal methods are discussed.

The validation of an ultra-reliable system must deal with two sources of error:

1. system failure due to physical component failure
2. system failure due to design errors

Presented at the AIAA Computing in Aerospace 9 Conference, San Diego, Ca., Oct. 19-21, 1993, pp 319-329.

There are well-known techniques for handling physical component failure—using redundancy and voting. The reliability assessment problem in the presence of physical faults is based upon Markov modeling techniques and is well understood. The design fault problem is a much greater threat.

There are 3 basic approaches to dealing with the design fault problem.

1. Testing (Lots of it)
2. Design Diversity (i.e., Software Fault-Tolerance: N-Version Programming, Recovery Blocks, etc.)
3. Fault Avoidance (i.e., Formal Specification and Verification, Automatic Program Synthesis, Reusable Modules)

The problem with life testing is that in order to measure the ultra-reliability one must test for exorbitant amounts of time. For example to measure a 10^{-9} probability of failure for a 1-hour mission one must test for more than 10^9 hours (i.e., 114,000 years).

There are many who advocate the use of design diversity to overcome the limitations of testing. The basic idea is to use separate design/implementation teams to produce multiple versions from the same specification. Then, through the use of threshold voters rather than exact-match voters, one can mask the effect of a design error in one of the versions while tolerating minor variations in calculations between versions. The hope is that the design flaws will manifest errors independently or nearly so. By assuming independence one can obtain ultra-high estimates of reliability even though the individual versions have failure rates on the order of $10^{-4}/hour$. When one examines the case for tolerance of physical faults, one finds that the only criterion that enables quantification of ultra-reliability for hardware systems with respect to physical failure is the *independence assumption.* However, the independence assumption has been rejected at the 99% confidence level in several experiments for low reliability software.[1, 2, 3, 4] Furthermore, the independence assumption cannot be validated for high reliability software because of the exorbitant test times required.

If one cannot assume independence one must measure correlations. However, this is infeasible as well. To measure correlations between versions would require as much testing time as life-testing the system because the correlations must be in the ultra-reliable region in order for the system to be ultra-reliable.

It is not possible, within feasible amounts of testing time, to establish that design diversity achieves ultra-reliability.[5] Consequently, design diversity can create an "illusion" of ultra-reliability without actually providing it.

Since we cannot quantify the reliability of ultra-reliable software, we must develop our systems in a manner that eliminates errors in the first place. In other words, we concentrate our efforts on producing a correct design and implementation rather than on the process of quantification. Our confidence in the software is derived from our rigorous analysis rather than by experimentation.

The Characteristics of Formal Methods

Central to formal methods is the use of mathematical logic. Mathematical logic serves the computer system designer in the same way that calculus serves the designer of continuous systems—as a *notation* for describing systems and as an analytical tool for calculating and *predicting* the behavior of systems. In both design domains, computers can provide speed and accuracy for the analysis.

Formal methods involves the specification of a system using languages based on mathematical logic. Formal methods provides a means for rigorous specification of desired properties as well as implementation details. Mathematical proof may be used to establish that an implementation meets the desired abstract properties. The most rigorous application of formal methods is to use semi-automatic theorem provers to ensure the correctness of the proofs. In principle, formal methods can accomplish the equivalent of exhaustive testing, if applied all the way from requirements to implementation. However, this requires a complete verification, which is rarely done in practice.

The reason that correct software is difficult to produce, even with large amounts of testing, is at first surprising—after all, we have been designing complex engineering systems for decades. Table 1 compares computer systems with classical systems and illustrates why the traditional approach to validation is ineffective. Unlike physical systems that are subject to physical failure, in software, there's nothing to go wrong but the *design*. Our intuition and experience is with continuous systems—but software exhibits discontinuous behavior. We are forced to separately reason about or test millions of sequences of discrete state transitions. Most of the design complexity in modern systems is in the software. The problem is that the complexity exceeds our ability to have intellectual control over it.

The term formal in "formal methods" refers to the idea that a proof can be known to be valid based upon its "form." In other words, the validity of a proof can be established by examining the syntax of an argument without regard to its semantics. The following argument:

That animal is a cat
All cats are sneaky
Therefore, that animal is sneaky

is valid independent of the meaning of "animal", "cat" or "sneaky." Thus, the following equivalent argument is also valid:

That ◯ is a □
All □s are ◇
Therefore, that ◯ is a ◇

Since the validity of a formal proof depends upon form only, a computer program can be used to check the validity of a proof without being supplied detailed domain-specific knowledge.

Formal logic provides rules for constructing arguments that are sound because of their form and *independent of their meaning*. Formal logic provides rules for manipulating formulas in such a manner that only valid conclusions are deducible from premises. The manipulations are called a *proof*. If the premises are true statements about the world, then the soundness theorems of logic guarantee that the conclusion is also a true statement about the world. Assumptions about the world are made *explicit*, and are separated from rules of deduction.

Logic provides the foundation for all mathematics. But traditional applications of mathematics have been to continuous systems, where highly developed bodies of theory (e.g., aerodynamics) remove practitioners from having to reason from the elementary logical underpinnings. But computer systems operate in a discrete domain; their operation is essentially a sequence of decisions, and each application is new. Therefore we must develop a specific theory about each one, directly in logic.

Formal methods can be roughly divided into two basic components: specification and verification. Formal specification is the use of notations derived from formal logic to (1) describe the *assumptions* about the world in which a system will operate, (2) the *requirements* that the system is to achieve and (3) a *design* to accomplish those requirements. Formal verification is the use of proof methods from formal logic to (1) analyze specifications for certain forms of consistency and completeness, (2) prove that the design will satisfy the

Classical Systems	Computer Systems
continuous state space	discrete state space
smooth transitions	abrupt transitions
finite testing and interpolation OK	finite testing inadequate, interpolation unsound
mathematical modeling	prototyping and testing
build to withstand additional stress	build to specific assumptions
predictable	surprising

Table 1. Comparison of Classical Engineering Systems with Computer Systems

requirements, given the assumptions, and (3) prove that a more detailed design *implements* a more abstract one. The mathematics of formal methods include (1) predicate calculus (1^{st} order logic), (2) recursive function theory, (3) lambda calculus, (4) programming language semantics and (5) discrete mathematics—number theory, abstract algebra, etc.

The following is a useful (first-order) taxonomy of the degrees of rigor in formal methods:

Level-1: Formal specification of all or part of the system.

Level-2: Paper and pencil proof of correctness.

Level-3: Formal proof checked by mechanical theorem prover.

Level 1 represents the use of mathematical logic or a specification language that has a formal semantics to specify the system. This can be done at several levels of abstraction. For example, one level might enumerate the required abstract properties of the system, while another level describes an implementation, which is algorithmic in style. *Level 2* formal methods goes beyond Level 1 through use of pencil-and-paper proofs that the more concrete levels logically imply the more abstract-property oriented levels. *Level 3* is the most rigorous application of formal methods. Here one uses a semi-automatic theorem prover to ensure that all of the proofs are valid. The Level 3 process of *convincing* a mechanical prover is actually a process of developing an argument for an ultimate skeptic who must be shown every detail. One can also add a Level 0 to refer to software engineering techniques that do not involve mathematical logic in a significant way, such as statically testing for uninitialized variables and V&V activities such as formal inspections. Intuitively, higher levels of rigor provide greater confidence but at greater cost.

It is also important to realize that formal methods is not an all-or-nothing approach. The application of formal methods to the most critical portions of a system is a pragmatic and useful strategy. Although a complete formal verification of a large complex system is impractical at this time, a great increase in confidence in the system can be obtained by the use of formal methods at key locations in the system.

Formal Requirements Analysis

In this section we will explore the process of writing a Level 1 formal specification of requirements. This will be done by way of example.

Suppose we want to develop an electronic telephone book, and we wish to write down the requirements for it using formal methods. We begin with some informal English requirements:

- Phone book shall store the phone numbers of a city
- There shall be a way to retrieve a phone number given a name
- It shall be possible to add and delete entries from the phone book

Mathematical Representation of a Phone Book

The first question that we face is how do we represent the phone book mathematically? There appear to be several possibilities:

1. As a set of ordered pairs of names and numbers. Adding and deleting entries via set addition and deletion.
2. As a function whose domain is all possible names and range is all phone numbers. Adding and deleting entries via modification of function values.
3. As a function whose domain is only names currently in the phone book and range is phone numbers. Adding and deleting entries via modification of the function domain and values. (Z style)

We decide to go with the second approach because it seems the simplest.

In traditional mathematical notation, we would define the phone book as follows:

$$\begin{array}{l}\text{Let } N = \text{set of names}\\ \quad P = \text{set of phone numbers}\\ book : N \longrightarrow P\end{array}$$

The set N represents all possible names, not just those in the city. Similarly the set P represents all possible phone numbers, not just those currently in service. How then do we indicate that we do not have a phone number for all possible names, only for names of real people? One possibility is to to use a special number, that could never really occur in real life, e.g. 000-0000. We don't have to specify the implemented value of this special number we can just give it a name: $p_0 \in P$.

Now we can define an empty phone book. In traditional notation, we would write:

$$\begin{array}{l}emptybook : N \longrightarrow P\\ emptybook(name) \equiv p_0\end{array}$$

Now we need to figure out how to represent English requirement 2: "There shall be a way to retrieve a phone number given a name." We decide to use a function "$FindPhone$."

$$\begin{array}{l}FindPhone : B \times N \longrightarrow P\\ FindPhone(bk, name) = bk(name)\end{array}$$

where B = set of functions : $N \longrightarrow P$. $Findphone$ returns a phone number when given a book and a name. Note that $FindPhone$ is a higher-order function since its first argument is a function (i.e., its type is B).

English requirement 3 stated, "It shall be possible to add and delete entries from the phone book." We decide to model these activities with two functions "$AddPhone$" and "$DelPhone$":

$$\begin{array}{l}AddPhone : B \times N \times P \longrightarrow B\\ AddPhone(bk, name, num)(x) = \begin{cases} bk(x) & \text{if } x \neq name\\ num & \text{if } x = name\end{cases}\end{array}$$

$$\begin{array}{l}DelPhone : B \times N \longrightarrow B\\ DelPhone(bk, name)(x) = \begin{cases} bk(x) & \text{if } x \neq name\\ p_0 & \text{if } x = name\end{cases}\end{array}$$

The specification for $AddPhone$ reads as follows: if you add an entry to the phone book for $name$ and then access entry x, you get the original value $bk(name)$ if $x \neq name$ and num otherwise. Similarly, $DelPhone$ states: if you delete the $name$ entry from the phone book and then access x, you get p_0 if $x = name$ and $bk(x)$ otherwise.

We can now write the complete specification:

$$\begin{array}{l}\text{Let } N = \text{set of names}\\ \quad P = \text{set of phone numbers}\\ \quad book : N \longrightarrow P\\ \quad p_0 \in P\\ \quad B = \text{set of functions} : N \longrightarrow P\\ FindPhone : B \times N \longrightarrow P\\ FindPhone(bk, name) = bk(name)\\ AddPhone : B \times N \times P \longrightarrow B\\ AddPhone(bk, name, num)(x) = \begin{cases} bk(x) & \text{if } x \neq name\\ num & \text{if } x = name\end{cases}\\ \\ DelPhone : B \times N \longrightarrow B\\ DelPhone(bk, name)(x) = \begin{cases} bk(x) & \text{if } x \neq name\\ p_0 & \text{if } x = name\end{cases}\end{array}$$

At this point we realize that our work is not completely satisfactory, for example:

- Our specification does not rule out the possibility of someone having a "p_0" phone number
- We have not allowed multiple phone numbers per name

The first question is an artifact of our particular specification; however, the second question is a result of a deficiency in the English specification.

Overcoming the Deficiencies

The first deficiency is that our requirements do not rule out the possibility of someone having a "p_0" phone number. One way to overcome this problem is to use a flag to indicate when a phone number is valid. However, this would not help us at all with the second deficiency—no way to store multiple phone numbers per name. The most straight-forward solution to the second deficiency is to make the phone book map into a set of phone numbers rather than just a single phone number. This also solves deficiency 1—the emptyset can be used to represent the situation where there is no phone number instead of using a p_0 number. Thus, we have:

$$\begin{array}{l}\text{Let } N = \text{set of names}\\ \quad P = \text{set of phone numbers}\\ \quad book : N \longrightarrow 2^P\\ \quad \mathcal{P} = \text{set of functions} : N \longrightarrow 2^P\\ book : \mathcal{P}\\ emptybook(name) \equiv \phi\end{array}$$

The notation 2^P represents the set of subsets of P. Thus, a book is a function from the set of names into a set of subsets of phone numbers (i.e., given a name it will return a set of phone numbers). The empty set ϕ can be used to represent the lack of a phone number for a name.

The $FindPhone, AddPhone$ and $DelPhone$ functions can be modified as follows:

Let N = set of names
P = set of phone numbers
$book : N \longrightarrow 2^P$
$\mathcal{P}$ = set of functions : $N \longrightarrow 2^P$
$emptybook(name) \equiv \phi$
$FindPhone : B \times N \longrightarrow P$
$FindPhone(bk, name) = bk(name)$
$AddPhone : B \times N \times \mathcal{P} \longrightarrow B$

$$AddPhone(bk, name, num)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ bk(name) \cup \{num\} & \text{if } x = name \end{cases}$$

$DelPhone : B \times N \longrightarrow B$

$$DelPhone(bk, name)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ \phi & \text{if } x = name \end{cases}$$

Notice that the function $DelPhone$ deletes all of the phone numbers associated with a name. Should the system be able to just remove one phone number associated with the name? The English requirements as written do not cover this situation. Clearly, the requirements must be corrected. If the capability to remove one of the phone numbers out of the set is needed, an additional function, say $DelPhoneNum$, must be defined:

$DelPhoneNum : B \times N \times \mathcal{P} \longrightarrow B$

$$DelPhoneNum(bk, name, num)(x) = \begin{cases} bk(x) & \text{if } x \neq name \\ bk(name) \setminus \{num\} & \text{if } x = name \end{cases}$$

Several aspects of the formal specification are significant. First, the specification is abstract and does not resemble program code. For example, the functions are defined over infinite domains. Second, the process of translating the requirements into mathematics has forced us to enumerate many things that are usually left out of English specifications. Third, the formal process exposes ambiguities and deficiencies in the requirements. For example one must chose between $book : N \longrightarrow P$ and $book : N \longrightarrow 2^P$ as the definition of the phone book.

Formal Analysis of Requirements

Although formal analysis can be carried out using pencil and paper, greater confidence in the analysis can be gained through use of a semi-automatic theorem prover, i.e. using Level 3 rigor. In order to use a theorem prover, the specification must be translated into the formal specification language used by the theorem prover. We will illustrate this process, using the PVS (Prototype Verification System) theorem prover.[6, 7, 8]

The specification becomes:

```
names: TYPE
ph_number: TYPE

IMPORTING sets[ph_number]

book: TYPE = [names -> set[ph_number]]

name: VAR names
emptybook(name): set[ph_number] = emptyset
bk: VAR book

FindPhone(bk, name): set[ph_number] = bk(name)

num: VAR ph_number
AddPhone(bk, name, num): book =
           bk WITH [name := add(num,bk(name))]

DelPhone(bk, name): book =
           bk WITH [name := emptyset]
```

A few observations should make the PVS syntax understandable. The first two lines define the types `names` and `ph_number`. These represent the domains of names and phone numbers, respectively. The `IMPORTING` command makes the PVS sets library available to the specification. The notation `[names -> set[ph_number]]` defines a function whose domain is `names` and whose range is `set[ph_number]`. The notation `bk WITH [name := add(num,bk(name))]` defines a new function identical to `bk` except at the point `name`. The value of the new function at `name` is set equal to `add(num,bk(name))`, the original set `bk(name)` with `num` added to it.

We can now analyze our requirements by posing challenges: "If this specification is correct, the following property should be true." For example, if I add a phone number to a name, then the set returned by $FindPhone$ should contain that entry:

$$num \in FindPhone(AddPhone(bk, name, num), name)$$

In PVS notation, we have

```
Find_Add_lem: LEMMA
  member(num,FindPhone(
               AddPhone(bk,name,num),name))
```

We issue the PVS prove command followed by a `TCC` command, a high-level strategy that is often able to automatically prove simple theorems. The system responds:

```
Rewriting AddPhone(bk, name, num) to ...
Rewriting FindPhone ...
Rewriting member(num, bk(name)) to ...
Rewriting add(num, bk(name))(num) to TRUE.
Rewriting member(num, add(num, bk(name))) to TRUE.

Trying repeated skolemization, instantiation,
and if-lifting,
```

```
Q.E.D.

Run time  = 3.80 secs.
Real time = 10.48 secs.
```

The PVS prover displays `Q.E.D.` which informs us that the theorem has been successfully proved. We have verified that our definition of `FindPhone` satisfies our expectation[†]. Encouraged by our success, we try another:

```
Del_Add_lem: LEMMA
    DelPhone(AddPhone(bk,name,num),name) = bk
```

This time our PVS proof effort leaves us with:

```
Del_Add_lem.1 :

[-1]   name!1 = x!1
  |-------
{1}   emptyset = bk!1(x!1)

Rule?
```

This is not provable because `bk!1(x!1)` (which is equal to `bk!1(name!1)`) is not necessarily equal to the empty set. We realize that after `DelPhone` removes `name` from the phone book that `bk(name)` will be equal to the empty set only for the case that there were no phone numbers for `name` before the `AddPhone` function operates on the phone book. Thus, we must change the lemma to:

```
Del_Add_lem: LEMMA emptyset(bk(name)) IMPLIES
  DelPhone(AddPhone(bk,name,num),name) = bk
```

At this point we have gained some additional insight into our requirements. Several questions arise that should be addressed in more detail in our requirements:

- Should we add a "*ChangePhone*" function that alters the phone numbers for an already existing *name*.
- Should we change the definition of *AddPhone* to only operate on non-existing names?
- Should error messages be output from the functions?

We will not pursue these questions further in this paper, but have raised them to illustrate how the putative theorem proving process can lead to a closer investigation of the requirements.

[†]Of course this is merely one of many properties we may wish to verify.

Revising the Informal English Requirements

One important product from the formal specification process is that it enables us to revise our English specification in a way that removes ambiguities. The original specification was

- Phone book shall store the phone numbers of a city
- There shall be a way to retrieve a phone number given a name
- It shall be possible to add and delete entries from the phone book

We now revise them to read:

- For each name in the city, a set of phone numbers shall be stored (Should we limit the number?)
- There shall be way to retrieve the phone numbers given a name
- It shall be possible to add a new name and phone number
- It shall be possible to add new phone numbers to an existing name
- It shall be possible to delete a name
- It shall be possible to delete one of several phone numbers associated with a name
- The user shall be warned if a deletion is requested on a name not in the city
- The user shall be warned if a deletion of a non-existent phone number is requested

There are many different ways to formally specify something. No matter what representation you chose you are making some decisions that bias the implementation. The goal is to minimize this bias and yet be complete. The process of formalizing the requirements can reveal problems and deficiencies and lead to a better English requirements document as well.

Design Verification

In this section we will briefly explore the techniques of design verification. This will be done by continuing with our phone book example. We decide to design our phone book using a hash table. For simplicity we assume that we have a hash function that will return a unique index into a multi-dimensional array for each name in the phone book. This is illustrated in Figure 1: The high-level design of the phone book can be specified in PVS as follows:

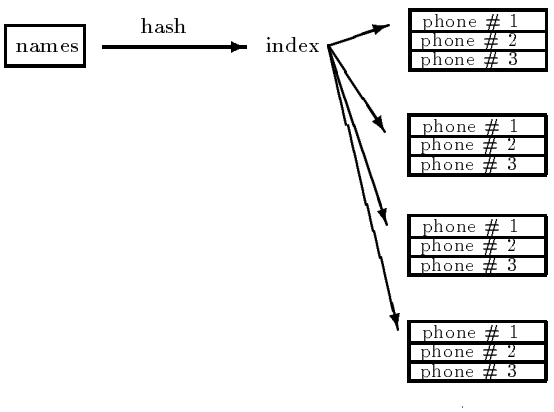

FIG. 1. Data Structure for Phone Book High-Level Design

```
index: TYPE = {i: nat | i < max_names}
nbufidx: TYPE = {i: nat | i <= max_numbers}
nbufloc:  TYPE = {i: posnat | i <= max_numbers}

numbuf: TYPE = ARRAY[nbufloc -> ph_number]
hashf: TYPE = function[names -> index]
numlist: TYPE = [# last: nbufidx, nbuf: numbuf #]

ibook: TYPE = ARRAY [index -> numlist]

hash: hashf

ibk: VAR ibook
name: VAR names
findphone(ibk, name): numlist
   = ibk(hash(name))
```

Similarly the high-level design for the "addphone" and "delphone" functions can be defined:

```
addphone(ibk, name, num): ibook =
  IF last(ibk(hash(name))) >= max_numbers
  THEN ibk
  ELSE % book is not full
    LET nl = ibk(hash(name)) IN
        ibk WITH [(hash(name)) :=
          (# nbuf := nbuf(nl)
             WITH [(last(nl)+1) := num],
               last := last(nl) + 1 #)]
  ENDIF

delphone(ibk, name): ibook =
    LET nl = ibk(hash(name)) IN
        ibk WITH [(hash(name)) := nl
              WITH [last := 0]]
```

One then constructs a mapping function that relates the objects of the high-level design specification to the objects of the requirements-level specification. In this case, we need a function that constructs the requirements-level phone book, a set of entries, from the high-level design data structure. We name this function, "bmap":

```
bmap: function[ibook -> book] =
      (LAMBDA ibk:
          (LAMBDA name:
              (LET nl = ibk(hash(name)) IN
                 smap(nbuf(nl),last(nl)))))
```

This function is defined in terms of the recursive function, "smap":

```
smap: RECURSIVE function[numbuf,nbufidx
        -> set[ph_number]] =
(LAMBDA nb,ii: IF ii = 0 THEN emptyset
               ELSE add(nb(ii),smap(nb,ii-1))
               ENDIF)
```

To show that the high-level design satisfies the requirements, we prove homomorphisms of the form:

```
Verif_condition: THEOREM
  NOT name_full(ibk,name) IMPLIES
    bmap(addphone(ibk, name, num)) =
      AddPhone(bmap(ibk),name,num)
```

In other words, if we start with a phone book `ibk`, add `name` to it, and then map it up to the requirements level with `bmap`, we obtain the same result as first mapping `ibk` up to the requirements level and then executing `AddPhone`. This is illustrated in Figure 2:

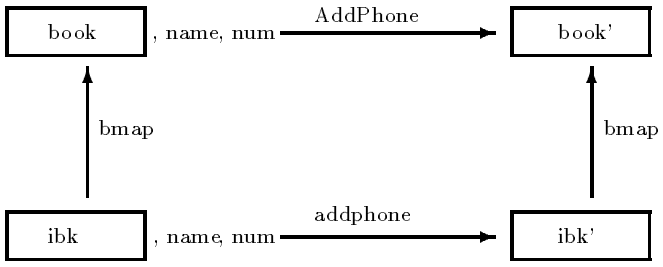

FIG. 2. Design Proof that High-Level Design addphone Function Implements Requirements-Level AddPhone

Introduction to Code-Level Verification

Presentation of the entire code-level specification, implementation and the corresponding formal verification is beyond the scope of this conference paper. However, some of the concepts involved can be introduced by

way of a single procedure that could be used in the implementation of this phone book—an array search function.

Let's begin with an English specification of such a procedure:

> The procedure searches an array "A" of length "N" for a value "X." If it finds the element, then "Y" is equal to the "index" of the array element that is equal to "X" on exit from the return. If there is no element of the array equal to "X" then Y is equal to "0" on exit.

The following is a formal specification of this procedure:

pre-condition: $N > 0$
post-condition:
$\{X = A[Y] \wedge (1 \leq Y \leq N)\} \vee \{(Y = 0)$
$\wedge (\forall k : (1 \leq k \leq N) \supset A[k] \neq X)\}$

The "pre-condition" describes what must be true of the input variables when the procedure is called and the "post-condition" presents a property on the output variables that defines the behavior of the routine. The $\wedge$'s represent logical or and $\vee$ represents logical and.

This specification could be implemented with a variety of different search techniques, e.g. linear search, binary search etc. For simplicity a linear search algorithm is presented here:

function $Lookup$(**var** A:**array**$[1..N]$ **of** $integer$; $x : integer$):
  $1..N$;
  var $i, m, n : 1..N$; **label** 11;
  $\{(1 < N) \wedge sorted(A) \wedge (A[1] \leq x < A[N])\}$
    begin $m := 1$; $n := N$;
    $\{(m < n) \wedge sorted(A) \wedge (A[m] \leq x < A[n])\}$
    while $m + 1 < n$ **do**
      begin $i := (m + n)$ **div** 2;
      if $x < A[i]$ **then** $n := i$
      else if $A[i] < x$ **then** $m := i$
      else begin $Lookup := i$; $\{A[Lookup] = x\}$
      goto 11 **end**
      end;
    $\{(m + 1 = n) \wedge sorted(A) \wedge (A[m] \leq x < A[n])\}$
    if $A[m] \neq x$ **then** $\{\neg\exists k.((1 \leq k \leq N) \wedge (A[k] = x))\}$
     goto 11
    else $Lookup := m$;
  11: **end**
  $\{(A[Lookup] = x) \vee (\neg\exists k.((1 \leq k \leq N) \wedge (A[k] = x)))\}$
  where
    $sorted(A) = \forall i, j.((1 \leq i < j \leq N) \supset (A[i] < A[j]))$

Note that the pre- and post-conditions have been added to the text as comments. In addition a "loop invariant" has been supplied for each loop. This is a property that is true about the loop whenever one reaches that point in the loop. Once these annotations are made a set of "verification conditions" can be automatically generated using a tool such as Penelope.[9, 10] If these verification conditions (VC) can be shown to be theorems, then the program correctly implements the specification.[‡]

For this program and specification, the following are the set of verifications that would be produced:

1. $\{(1 < N) \wedge sorted(A) \wedge (A[1] \leq x < A[N]\}$
   $\supset \{(1 < N) \wedge sorted(A) \wedge (A[1] \leq x < A[N])\}$
2. $\{A[Lookup] = x\} \supset \{A[Lookup] = x\}$
3. $\{(m + 1 = n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (A[m] = x)\}$
   $\supset \{A[m] = x\}$
4. $\{Failure\} \supset \{Failure\}$
5. $\{(m + 1 = n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (A[m] \neq x)\}$
   $\supset \{\neg\exists k.((1 \leq k \leq N) \wedge (A[k] = x))\}$
6. $\{(m < n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (m + 1 < n)$
   $\wedge (A[(m + n)\ div\ 2] \geq x) \wedge (x \geq A[(m + n)\ div\ 2])\}$
   $\supset \{A[(m + n)\ div\ 2] = x\}$
7. $\{(m < n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (m + 1 < n)$
   $\wedge (A[(m + n)\ div\ 2] < x) \wedge (x \geq A[(m + n)\ div\ 2])\}$
   $\supset \{(((m + n)\ div\ 2) + 1 = n) \wedge sorted(A)$
   $\wedge (A[(m + n)\ div\ 2] \leq x < A[n])\}$
8. $\{(m < n) \wedge sorted(A) \wedge (A[m] \leq x < A[n]) \wedge (m + 1 < n)$
   $\wedge (A[(m + n)\ div\ 2] < x) \wedge (x < A[(m + n)\ div\ 2])\}$
   $\supset \{(m + 1 = (m + n)\ div\ 2) \wedge sorted(A)$
   $\wedge (A[m] \leq x < A[(m + n)\ div\ 2])\}$

The overall process is illustrated in Figure 3.

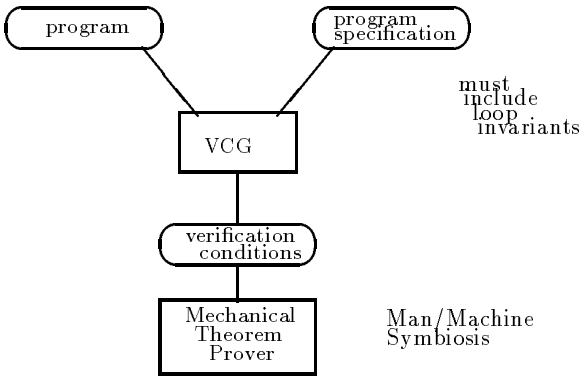

FIG. 3. The VC generation process

One can see that the specification above is very detailed and deals specifically with implementation variables. In fact, code-level verification is usually the most time-consuming of all of the formal methods, because of the amount of detail that must be handled. The formal specification that drives the VC generation process can be connected to the upper-level design specifications to make a formal hierarchy as shown in Figure 4. The upper-level proofs are accomplished using the techniques of design proof described in the previous section.

[‡]Of course this is true in practice only if the semantics of the language used for the VC generation match the actual semantics of the language employed and there are no bugs in the VC generator and compiler.

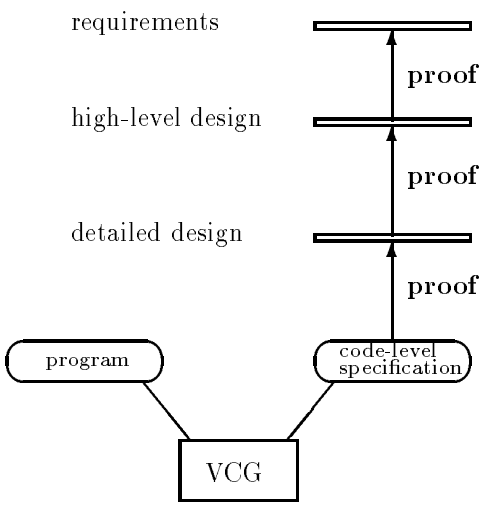

FIG. 4. Hierarchical Specification Used to Prove High-Level Property

Maturity of Formal Methods

The major drawback cited by critics is that formal methods is too expensive and time-consuming to be practically applied. While this criticism was perhaps true twenty years ago, much progress has been made in development of formal methods languages, tools, and techniques.

Most of the commercial application of formal methods has occurred in Europe. Most noteworthy is the IBM CICS Project.[11] This project applied formal methods to an upgrade of a major on-line transaction-processing software package. The size of the upgrade was 13,230 lines of code. The project team claimed that 19 defects were avoided as a result of using formal methods. They also claimed a cost savings of 9% of total or $13 million saved. They used the Z specification language at the Level 1 level of rigor.

Another noteworthy application of formal methods is the Inmos/Oxford T800 Transputer Floating-Point Unit Project. This project involved the application of formal methods to the design of a hardware device. The T800 Transputer Floating-Point Unit Project originally begin with two separate, parallel developments: an informal development, supported by months of testing against other FPUs and a formal development using Z. Because the formal development moved far ahead of the informal team, the informal effort was terminated. Inmos claims a saving of 12 months in the development time. They received the Queen's Award for Technological Achievement 1990.

Another successful application of formal methods is the SACEM Railroad Signalling System.[12] The objective of this project was to increase traffic movement by 25% (800,000 passengers/day). This involved 21,000 lines of Modula-2 code of which 63% was safety-critical. They used Level 2 rigor, performing manual proofs on the VCs. The validation effort for the total system was 100 man-years. The development team believes that the system is safer as a result of the use of formal methods.

Meanwhile in the United States, the National Security Agency and the Defense Advanced Research Projects Agency (DARPA) have quietly funded quite a lot of formal methods research, resulting in significant advances in theorem-proving tools (e.g., Gypsy, EHDM, SDVS) and in the complexity of systems that can be formal verified (e.g., encryption devices, secure operating systems, microprocessors).

NASA Langley Research Center has established a research program aimed at bringing formal methods technology to a sufficiently mature level for practical use on life-critical systems by United States aerospace and other industries and to facilitate the transfer of this technology through carefully orchestrated demonstration projects. Our research efforts are primarily concentrated on the technically challenging areas of digital flight-control systems design that are currently beyond the state of the art. Demonstration projects are focussed on problem domains where current formal methods technologies are deemed adequate but techniques and examples of how to apply them are absent. To overcome the sizeable "learning curve" associated with adoption of formal methods and their application to new problem domains, these demonstration projects are accomplished by establishing cooperative partnerships between industry and the developers of the formal methods tools and techniques.

Our software demonstration projects began with formal verification of some simple utility routines obtained from the NASA Goddard Space Flight Center and the NASA Lewis Research Center. This work was performed by Odyssey Research Associates (ORA) using their Ada verification tool named Penelope.[13] During this project, ORA demonstrated that the use of formal specification alone uncovered several errors in the routines and that the subsequent formal verification effort uncovered additional errors.[10] In a second project, ORA formally specified the mode-control panel logic of a Boeing 737 experimental research aircraft using Larch (the specification language used by Penelope).[14]

We are participating with NASA Johnson Space Center and the Jet Propulsion Laboratory (JPL) to demonstrate the use of formal methods for space applications. In this project, we are working with space application experts from NASA Johnson, JPL, and IBM to

- educate the application experts about the PVS prover and how to apply formal methods,
- work jointly to develop a hierarchical set of formal specifications of the Jet-Select function of the NASA Space Shuttle, ranging from pseudo-code level to detailed-design level to abstract high-level specification,
- demonstrate how to prove that each level specification is a valid implementation of the level above, and
- demonstrate how to prove that the requirements-level specification meets a set of properties that the system is required to hold.

Other demonstration projects related to software include:

- formal specification and verification of floating point software for calculating trajectories of a ballistic missile;
- formal specification of guidance and control system software for a planetary lander;
- design, specification and verification of an operating system for a fault-tolerant, Reliable Computing Platform; and
- development of a formal requirements definition language for flight-control software.

This work along with the rest of NASA Langley's research in formal methods is discussed in an overview paper presented at Compass 91.[15]

Since the Federal Aviation Administration (FAA) must approve any new methodologies for developing life-critical digital systems for civil air transports, their acceptance of formal methods is a necessary precursor to its adoption by industry system designers. Therefore, we have been working with the FAA and other regulatory agencies to incorporate credit for formal methods into the standards they set. We presented a tutorial to the FAA SWAT (SoftWare Advisory Team) at their request, and SRI International is currently writing a chapter for the FAA Digital Systems Validation Handbook on formal methods. We were instrumental in including formal methods as an alternate means of compliance in the DO-178B standard.

Limitations

It is important that the limitations of formal methods be recognized. For many reasons, formal methods do not provide an absolute guarantee of perfection, even if applied with Level 3 rigor. First, formal methods cannot guarantee that the top-level specification is what was *intended*. Second, formal methods cannot guarantee that the mathematical model of a physical device such as a hardware gate is accurate with respect to the physics of the device. The formal verification depends upon the the validity of the models of the primitive elements such as hardware gates. The mathematical model of a gate is merely a representation of the physical device. Some formal models just include logical properties. Other formal models include timing delays, but formal models typically do not include effects of temperature, EMI, manufacturing flaws, etc. Third, often the formal verification process is only applied to part of the system. Finally, there may be errors in the formal verification tools themselves. Nevertheless, formal methods provide a significant capability for discovering/removing errors in large portions of the design space.

Concluding Remarks

This tutorial-style paper describes in simple terms what formal methods is and how it can be applied to software. We believe that formal methods tools and techniques are already sufficiently mature to be practical and cost-effective in the development and analysis of life-critical software systems. Several examples of formally specified and verified systems support our position. The intellectual investment required to adopt formal methods is considerable. However, we see no acceptable alternative; the use of computer software in life-critical applications demands the use of rigorous formal specification and verification procedures.

References

[1] Knight, John C.; and Leveson, Nancy G.: An experimental evaluation of the assumptions of independence in multiversion programming. *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, Jan. 1986, pp. 96–109.

[2] Scott, R. Keith; Gault, James W.; and McAllister, David F.: Fault-Tolerant Software Reliability Modeling. *IEEE Transactions on Software Engineering*, May 1987.

[3] Shimeall, T. J.; and Leveson, N. G.: An Empirical Comparison of Software Fault-Tolerance and Fault Elimination. *IEEE Transactions on Software Engineering*, Feb. 1991, pp. 173–183.

[4] Knight, John. C.; and Leveson, Nancy. G.: A Reply To the Criticisms Of The Knight & Leveson Experi-

ment. *ACM SIGSOFT Software Engineering Notes*, Jan. 1990.

[5] Butler, Ricky W.; and Finelli, George B.: The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Transactions on Software Engineering*, vol. 19, no. 1, Jan. 1993, pp. 3–12.

[6] Shankar, N.; Owre, S.; and Rushby, J. M.: *The PVS Proof Checker: A Reference Manual (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[7] Owre, S.; Shankar, N.; and Rushby, J. M.: *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[8] Owre, S.; Shankar, N.; and Rushby, J. M.: *User Guide for the PVS Specification and Verification System (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[9] Hird, Geoffrey: Formal Specification And Verification Of Ada Software. In *AIAA Computing in Aerospace 8 Conference*, Baltimore, MD., Oct. 1991.

[10] Eichenlaub, Carl T.; Harper, C. Douglas; and Hird, Geoffrey: *Using Penelope to Assess the Correctness of NASA Ada Software: A Demonstration of Formal Methods as a Counterpart to Testing.* NASA Contractor Report 4509, May 1993.

[11] Houston, I.; and King, S.: CICS project report: Experiences and Results From the Use Of Z in IBM. In *VDM '91,* Formal Software Development Methods, vol. LNCS 551, pp. 558–603. Springer-Verlag, 1991.

[12] Guiho, G.; and Hennebert, C.: SACEM Software Validation. In *Proc. 12th International Conference on Software Engineering.* IEEE Computer Society Press, Mar. 1990, pp. 186–191.

[13] Guaspari, David: Penelope, an Ada Verification System. In *Proceedings of Tri-Ada '89*, Pittsburgh, PA, Oct. 1989, pp. 216–224.

[14] Guaspari, David: Formally Specifying the Logic of an Automatic Guidance Controller. In *Ada-Europe Conference*, Athens, Greece, May 1991.

[15] Butler, Ricky W.: NASA Langley's Research Program in Formal Methods. In *6th Annual Conference on Computer Assurance (COMPASS 91)*, Gaithersburg, MD, June 1991.