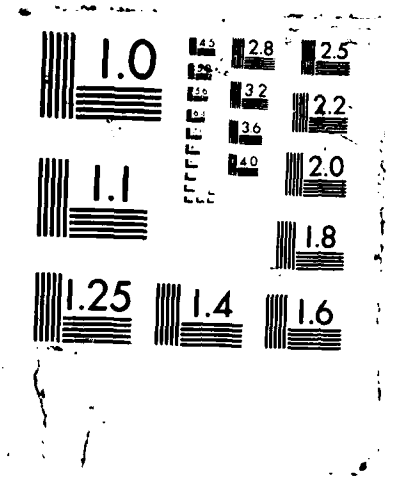
AD-A190 570 SOFTWARE VERIFICATION USING PARTITION ANALYSIS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING R P GRAHAM MAR 88 AFIT/GCS/MA/88M-02 1/1

UNCLASSIFIED F/G 12/5 NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

AFIT/GCS/MA/88M-02

SOFTWARE VERIFICATION USING PARTITION ANALYSIS

THESIS

Robert P. Graham, Jr.
Second Lieutenant, USAF

AFIT/GCS/MA/88M-02

AFIT/GCS/MA/88M-02

SOFTWARE VERIFICATION USING PARTITION ANALYSIS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

Robert P. Graham, Jr., B.S.

Second Lieutenant, USAF

March 1988

Acknowledgements

There are a great many people who contributed in one way or another toward making this thesis a reality, more than can be mentioned here. Greatest thanks go to Dr. Panna Nagarsenker, my thesis advisor, and Captain David Umphress, committee member, for their patience, confidence and guidance during difficult times. I also thank Dr. Henry Potozcny, the third member of my committee.

I am grateful to Will Bralick for long talks and even longer Ping-Pong matches that helped keep me motivated, and to my friends Steve, Kay, and others for many fun times that helped keep me sane. Finally, I thank my parents for their support, love, and understanding.

Table of Contents

List of Figures

Abstract

Software verification is the process of determining whether a piece of software is reliable--whether it performs as it is supposed to. As traditionally performed, program verification can account for 40 percent or more of the development time and cost of a software product. In spite of this fact, released software is notorious for its unreliability. These two facts, the expense of our attempts at program verification and our limited success, have sustained a great deal of research interest directed at finding more effective methods.

This thesis develops extensions to a promising new verification technique called Partition Analysis, developed by Debra J. Richardson (1981). Partition Analysis appears to be a powerful approach for identifying program faults, but in its current state can only be applied to single program modules that produce no side effects, including input or output. This thesis extends the applicability of Partition Analysis by permitting the use of procedure and function calls, thereby allowing complete programs to be analyzed. The result is a set of techniques for handling regular, non-recursive procedure and function calls, separate methods for the analysis of recursive procedures and functions, and an approach to the larger problem of analyzing entire programs.

SOFTWARE VERIFICATION USING PARTITION ANALYSIS

I. Introduction

Program verification is the process of determining whether a piece of software is reliable--whether it performs as it is supposed to. As traditionally performed, program verification can account for 40 percent or more of the development time and cost of a software product (Pressman, 1987: 467). In spite of this fact, released software is notorious for its unreliability (Pressman, 1987: 13-14). These two facts, the expense of our attempts at program verification and our limited success, have sustained a great deal of research interest directed at finding more effective methods. This thesis is a study of a relatively new technique for program verification called Partition Analysis (Richardson, 1981). The Partition Analysis technique shows great promise, but in its current form is severely limited in its scope of applicability. The contribution this thesis makes is to expand that scope.

Background

General Approaches to Program Verification. In the best of all possible worlds, the generation of software would be a fully automated procedure. The user or other interested party would prepare a specification detailing the desired functional and operational characteristics of the software and would present it to a system that would automatically generate a program guaranteed to match the specification. Automatic programming, as this procedure is called, is one of the long-standing goals of the Artificial Intelligence (AI) field but, not surprisingly, this ideal is far from realization: programming is a complex, knowledge-intensive, incompletely understood and error prone process even for human experts. Efforts toward automating the process have been categorized into three subareas:

1. AI programming environments;
2. studies of the software design process; and
3. knowledge-based software assistants.

(Mostow, 1985: 1253)

Automatic programming remains a long-range goal, but is not expected to solve the verification problem in the foreseeable future.

If automated programming is currently beyond our capabilities, the next best thing would be to write a program and then apply some method (again, preferably an automated one) to show that it meets its specification. In other words, we would like to develop a proof of the correctness of our program, with full mathematical rigor and certainty.

An early and still active approach proceeds by defining a formal semantics for a programming language in the form of a set of axioms. Programs in that language can then be translated into assertions in the predicate calculus, and the correctness of the program--its correspondence with a specification likewise expressed in the predicate calculus--becomes a theorem to be proved (Hoare, 1969). Unfortunately, both the translation and the proof are tedious and error prone, and not easily automated. It has also been noted by several researchers that the proper way to apply formal proof techniques is to develop the program and the proof concurrently, or even to derive the program from the proof (Dunn, 1984: 159). This further complicates attempts at automating the approach, which as a manual technique remains impractical for all but the smallest programs (Dunn, 1984: 159).

Given the current impossibility of automatic programming and the extreme difficulty of applying formal techniques, a third approach toward achieving at least some confidence in the reliability of our software is testing: running a program with some subset of the data it is supposed to handle and checking the result. As Dijkstra has pointed out, "testing can only reveal the presence of errors, never their absence" (Dijkstra, 1972). It has been shown that anything short of exhaustive testing (running the program with every possible set of inputs) leaves open the possibility of an incorrect program escaping detection by working correctly on the subset of data tested (Weyuker and Ostrand, 1980). In spite of this rather dismal fact, a carefully chosen set of test data can reveal many of the errors in a program, and a test run resulting in no errors detected can greatly increase confidence in the reliability of software.

Software testing, done informally, was the first technique applied to verifying software. Over the years a large number of techniques have been developed trying to make it more effective and efficient. Many of these have also been automated, at least in part.

Symbolic execution is a verification technique that combines formal verification and testing. In symbolic execution input values are represented by symbols instead of literal values and statements are executed symbolically to produce formulas for the values of the program variables. These formulas can then be analyzed for correctness or to guide the selection of ordinary test data. Several automated systems for symbolic execution have been developed.

Partition Analysis

Partition Analysis is a technique developed by Debra Richardson in her doctoral research that combines both formal verification techniques and testing in order to acquire confidence in a program's reliability (Richardson, 1981). It is also distinctive in that it makes extensive use of both the specification and implementation of a design.

As currently developed, Partition Analysis can be applied to single modules that do not produce side effects (including I/O). It is also directed primarily at numerical algorithms and has not been tried on programs that perform symbolic processing.

The Partition Analysis method consists of three steps. First, symbolic evaluation and other analysis techniques are used to produce the procedure partition of the module. Each element of this partition defines a subdomain of the module's input and describes the computation to be performed on this domain according to the specification and also according to the implementation. Second, proof techniques are applied to the two computation descriptions to demonstrate their equivalence (or nonequivalence, in which case a fault has been found). Third, the subdomain and computation descriptions are used to guide the selection of test data to exercise the functional behavior of the module.

Problem and Approach

Evaluation of Partition Analysis by Richardson and Clarke (1985) shows it to be very effective at finding program faults. The restriction to single modules with no side effects, even I/O, however, drastically limits its usefulness. As currently developed it remains an academic exercise, but one with great potential. This thesis attempts to realize some of that potential by enhancing Partition Analysis to make it applicable to a larger class of programs. In particular, the objective chosen was to devise methods for handling procedure and function calls. Since most programming languages permit recursive calls, the special case of recursion was specifically include in the objective.

Procedures and functions are the building blocks of all practical programs. Expanding Partition Analysis to permit their use make the technique applicable to entire programs rather than just isolated modules. It was felt that this extension would therefore be more useful than, say, the inclusion of I/O or the elaboration of any of the methods already used internally by Partition Analysis.

The approach followed was first of course to understand as fully as possible the Partition Analysis method itself. Since Partition Analysis used many other verification techniques, an extensive review of these techniques was necessary. The basic method for handling procedure and function calls was devised by a close examination of symbolic execution, the particular needs of Partition Analysis, and some hints in Richardson's dissertation (1981) itself. Separate methods for recursive calls were developed by analogy with Richardson's methods for analyzing program loops. The new methods focus on step one of Partition Analysis, forming the procedure partition when procedure and function calls are present. The formal verification and testing steps are also affected, but to a lesser degree.

The result is a set of techniques for handling nonrecursive procedure and function calls; the choice of which technique to use depending on the program to be analyzed.

Recursive calls are handled by their own technique, again with variations. Several examples of recursive and nonrecursive routines were analyzed to verify that the proposed techniques do work.

Overview of the Rest of the Thesis

The remainder of the thesis consists of four chapters. Chapter 2 is a close look at many of the techniques that have been developed for software testing, and a look at symbolic execution. Chapter 3 explains the Partition Analysis method in detail, with a detailed example. Chapter 4 presents the extensions that were made to the method. Chapter 5 offers some conclusions about the method and points to several directions for future work.

II. Literature Review

Numerous techniques for carrying out software verification have been developed. This chapter reviews some of the work done in program testing and symbolic execution. Since Partition Analysis uses many of these techniques itself, this chapter also provides further background on the method.

Software Testing

The effort to replace ad hoc testing practices with systematic methods has produced a number of basic techniques that reflect early attempts to address the testing problem or that address particular special cases. The basic techniques are typically categorized as being either "black box" or "glass (sometimes white) box" techniques. Black box testing relies strictly on the specification to describe the intended function of the software and to guide the selection of sample data to test whether the software implements the function correctly. Glass box testing augments the information provided by the specification with structural information about how the program works. Both kinds of techniques are used in practice; the two approaches are complementary in that they tend to detect different classes of errors (Pressman, 1987: 484). Glass box techniques are more mathematically tractable and more generally effective but can get unwieldy, while black box techniques are easier to apply but are more likely to miss certain types of errors.

Recently these basic ideas have begun to be combined into comprehensive approaches to disciplined testing in-the-large. Two such approaches are Howden's Functional Testing approach (Howden, 1986) and Partition Analysis, discussed in detail in the following chapter.

Before discussing specific techniques, some terminology is needed. A program failure is an discrepancy between the observed behavior of a program and its intended behavior. A program fault is an error in a program that causes it to fail. Failures have been classified as either domain errors or computation errors. Domain errors are concerned with which execution path is followed in the processing of data. If the wrong path is followed, a path selection error has occurred. If the data falls into a special case that the program fails to recognize altogether, a missing path error has occurred. Computation errors occur when the correct path is followed, but the path processes the data incorrectly. In practice, one program fault can cause many failures, and some failures can be of more than one type.

Black Box Testing. Black-box methods focus on exercising all functional requirements of a program without considering its implementation. The basic idea is to break down the specification and identify all of the individual functions performed, and then test each one (Dunn, 1984: 233).

Equivalence partitioning (Myers, 1979) is a method for partitioning the input domain into classes that we can "reasonably" assume are processed equivalently. Testing any value in a class then provides confidence that all values in the class are processed correctly. The technique of breaking up the input domain into subdomains is called domain analysis and is used in other black-box methods as well. Domain analysis is also a glass-box technique when applied to an implementation. In this case each path through the program is considered, and the conditions that the input must meet in order for that path to be followed constitute the input domain.

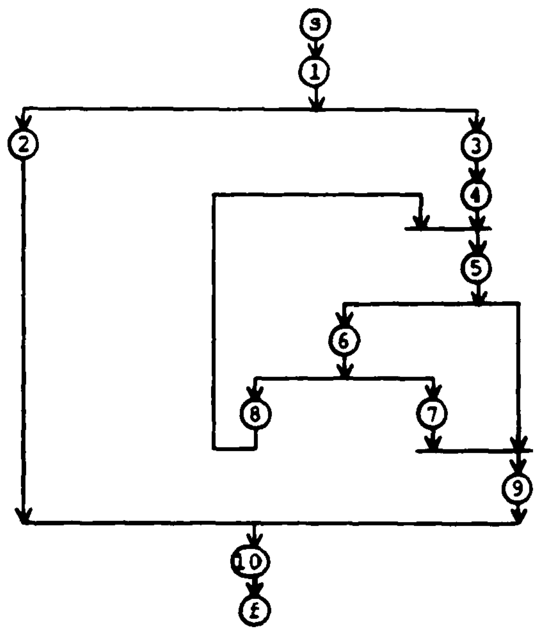
Empirical data has shown that more errors tend to occur at the boundaries of an input domain, so these boundaries should be exercised more fully. Once equivalence partitioning is done, test cases are chosen that lie just inside each class, just outside each class, and somewhere in the "middle" (Pressman, 1987: 486). Extensive work has been done showing how close to the boundaries to get and how effective this technique is, particularly the glass-box version (White and Cohen, 1980; Clarke, Hassell, and Richardson, 1982).

Fault seeding is a black-box technique that deliberately puts errors into a program in order to judge the effectiveness of the testing being done. In this technique a number of errors are deliberately introduced and then testing by some other tecnnique is done. If n faults were seeded, m total faults were exposed, and c of the faults found were seeded faults, then an estimate of the total number of faults N in the program is $N = (n * m) / c$, and so the number remaining undetected is $N - m$. In practice, if the ratio $(m - n) / (N - n)$ is less than 0.9, more testing is called for; values close to 1 provide high confidence in the effectiveness of the tests.

Glass Box Testing. Glass-box testing uses the details of the implementation to guide the selection of test data (Dunn, 1984 : 199). One important tool used by most glass-box techniques is the control flow graph of a program. In this graph the nodes represent statements of the program and edges represent the flow of control from one statement to another. Associated with each edge is a condition that must be true for the transfer to take place. For sequential flow or an unconditional branch the condition has the constant value true and is generally not shown; sequences of statements with no branches are sometimes collapsed into a single node. The common if-then-else and loop constructs typically use boolean decisions to choose one of two possible edges to follow: the condition for one edge is simply the negation of the other. More complex control structures such as the case statement or the select statement in Ada have more arbitrary sets of conditions. Figure 2-1 shows a program for determining whether a positive integer is a prime, and gives the corresponding control flow graph.

```
function Prime (N : POSITIVE) return BOOLEAN is
      Factor : INTEGER;
      IsPrime : BOOLEAN;
s  begin
1     if (N mod 2 = 0) or (N mod 3 = 0) then
2        IsPrime := (N < 4);
      else
3        IsPrime := TRUE;
4        Factor := 5;
5        while Factor ** 2 <= N loop
6           if (N mod Factor = 0)
                  or (N mod (Factor + 2) = 0) then
7              IsPrime := False;
               exit;
            else
8              Factor := Factor + 6;
            end if;
9        end loop;
      end if;
10    return IsPrime;
f  end Prime;
```

a. Ada source code for PRIME
Adapted from Richardson, 1981: 24.



b. Control flow graph for PRIME

Figure 2-1. Example Control Flow Graph

A technique called basis path testing is a simple way to use the control flow graph to guide test data selection (Pressman, 1987: 472-482). The graph is used to find a minimal set of linearly independent paths through the graph such that each edge appears in at least one path. The number of paths required to do this is $E - N + 2$, where E is the number of edges in the graph, and N is the number of nodes. Test data is selected that will cause the program execution to follow each of these paths.

Loop constructs are very common in programs and have special problems associated with them, so special methods for testing loops have been devised (Pressman, 1987: 483-4). Problems associated with loops are initialization errors, indexing or incrementing errors, and bounding errors at loop limits. For a simple loop, test cases are devised that skip the loop entirely, pass through the loop exactly once, exactly twice, some "large" number of times m, and if there is a maximum number of times the loop is allowed to be executed, n, then the loop is tested for $n - 1$, n, and $n + 1$ (if possible). Nested loops and concatenated loops lead to an inordinate amount of testing, so some simplifications are specified to control this.

Mutation testing (Dunn, 1984: 218-220) is a glass-box technique somewhat like fault seeding that is intended to increase confidence in the ability of the chosen test cases to detect errors. Small changes are made one at a time to the program to introduce errors deliberately. Each "mutant" program is then tested to see if a failure occurs. If so, then that test case has been shown effective at finding errors and our confidence that the original program is correct increases. If the mutant passes the test, however, it is analyzed to see if it is in fact equivalent to the original program. If it is not, then the test has failed to detect an error, and our confidence that the original program does not contain a similar error decreases (in fact, it could be that the mutant is correct and the original wrong). Mutation testing works best on programs that are believed to be basically sound except for relatively simple errors (Richardson, 1981: 54).

The glass-box techniques discussed so far address both domain and computation errors, but computation errors in particular are prone to coincidental correctness when only a few test cases are used, and so special techniques for testing for certain kinds of computation errors have been developed (Howden, 1980). For instance, expressions that are polynomials can be shown correct by testing them with a number of test cases one greater than the degree of the polynomial. Other types of expressions have known special values (extrema, discontinuities, etc) that need to be checked for. The value zero in particular is a very common source of error and so should always be tested.

Comparison of black- and glass-box techniques. Black-box techniques are best at finding certain types of errors: incorrect or missing functions; interface errors; errors in data structures; performance errors; and initialization and termination errors (Pressman, 1987: 485). Black box techniques tend to miss many other errors because the implementation is rarely exercised fully: a single function in the specification may be implemented as several special cases, and a black box approach has no way of knowing this; we are forced in effect to guess at what values will be processed the same way by the program, without looking at it. "The disadvantage of the black box testing approach is that it ignores important functional properties of programs which are part of its design and implementation and which are not described in the requirements" (Howden, 1980: 162). Thus the black and glass box approaches must be used together for maximum effectiveness.

Functional Testing (Howden, 1980; Howden, 1986). Howden's functional testing and analysis is an attempt to reintegrate the primary alternative approaches to software testing: static versus dynamic testing, black versus glass box testing, and practical versus theoretical considerations. It is an outgrowth of earlier empirical studies of software testing, and is intended to "provide a framework for the discussion of testing, to provide practical theoretical results, to derive new results, and to indicate directions for future research" (Howden, 1986: 997).

The theoretical basis is provided by a combined view of the behavioral and structural properties of programs. At the behavior level a program defines a mapping from its input domain to its output domain; the specification of a program is intended to provide a complete and accurate description of this mapping, and is needed during testing as an "oracle" for determining if a program is producing the correct results. Structurally, a program is seen as a collection of functions and data types. Functions act to convert one type of data into another type, and the overall transformation of the input types to the output types defines the behavior of the program. Thus there is a duality between types and functions. Program design methodologies (e.g., functional decomposition, data flow analysis, Structured Design or object-oriented design) typically choose to emphasize one aspect over the other; Howden's approach to testing likewise emphasizes the identification and analysis of the functional components of a program, as the name suggests. He provides an analysis of how the functions of a program are structured, in terms of what he calls forms and structures.

In practice, Functional Testing consists of two steps: a static analysis of the program and its specification to identify its functional subunits and the domains and ranges of each, and then test data selection and the dynamic analysis of the results of executing the program on that data. The static analysis step proceeds by identifying three types of units: functionally important classes of input and output data, much as is done in the black box techniques; data design structures within the program, which are subsets of declared data structures that are functionally related; and program design structures, which are identifiable functions used to design and implement the program. Program design structures may or may not correspond to contiguous pieces of code, and Howden suggests that any available design documentation (e.g., data flow diagrams, SADT charts) can help in the identification of those functions whose implementation corresponds to collections of paths in the program, or to scattered pieces of code.

In the test data selection and execution step, Functional Testing combines black and glass box techniques by partitioning the input domains not only of the program as a whole but also of the individual component functions identified in the program's structure. Howden defines a number of rules for identifying these partitions and for devising fault-revealing test cases for them. Execution of these test cases exercises the boundaries and internal regions of the input and output domains of all the functional units in the program, as well as the interfaces between units, and hence is claimed to be very effective at detecting faults in the program. Howden compares the coverage achieved by his method and that of other glass box techniques, and finds his is more demanding, and hence more effective at finding faults. He reports empirical studies to support this conclusion.

In effect, Howden claims that techniques like branch analysis and basis path testing function as approximations to Functional Testing: the validating assumption needed to make such techniques work is that paths in programs correspond to functions. These techniques are weak because this assumption is not true in general. The true structure of a program is the functions that comprise it, regardless of its textual structure, and this is what Functional Testing focuses on. Conversely, it is precisely the loose correspondence between functional structure and textual structure that makes Howden's method much more difficult to automate than the other glass box techniques; this is the major disadvantage of the Functional Testing approach. Without automation, applying the technique to any sizable program remains impractical. Howden discusses some possible approaches to automating parts of his scheme, but the area remains open to future research.

Symbolic Execution

Somewhere between formal verification and ordinary testing lies symbolic execution (Dunn, 1984: 136-137; Darringer and King, 1978; Howden, 1977; King, 1976). Symbolic execution is a generalization of the usual execution model for computer programs in that program variables may be given symbolic representations of their values instead of the value themselves. The convention used in this thesis is to capitalize the first letter of program variable names and to use all lower case for symbolic values. For example, one might represent the value of variable A by a and the value of B by b. Then after execution of the statements

```
A := A * A;
B := B * B;
C := A + B;
```

the values of the variables would be

```
A = a * a
B = b * b
C = a * a + b * b
```

Normal execution is the special case where A and B have actual numerical values and so the computations can be carried out. The relationship between symbolic execution and regular execution has been likened to that between algebra and arithmetic.

If a program contains branches or loops, then merely representing the values of all program variables is not enough, since these values depend on which program path was followed. During symbolic execution a special variable called PC (path condition) is maintained to indicate all of the conditions that had to be true in order for the executed path to have been followed. PC is initially set to "true," and then each time a conditional branch is executed PC is

AND-ed with the condition corresponding to the branch taken. For example, let the value of PC be "true," the value of N be n, and consider executing the following statement:

```
if N = 1 then
   X := 5;
else
   X := 10;
end if;
```

There are two possible paths here. If the first is chosen, then the result is

```
PC = true and (n = 1)
   = (n = 1)
X  = 5
```

If the alternative is chosen, the values would be

```
PC = true and (n /= 1)
   = (n /= 1)
X  = 10
```

Sometimes the current value of PC is enough to determine which way a branch should go. For instance, if in the above example some previous branch had set PC to include the condition (n < 0), then the second path is the only possible one and PC could be left unchanged: the information that (n /= 1) is redundant. Thus only "unresolved" branches are recorded in PC.

Numerous automated systems have been developed to do symbolic execution. One of the first is EFFIGY (King, 1976). It supports a simple (but nontrivial) language with a PL/I-style syntax. It functions much like an interactive debugger, proving trace and breakpoint facilities. It also allows the user to specify variable values, either as literals or symbolically, allowing ordinary execution, pure symbolic execution, or any combination. When an unresolved conditional is encountered during execution, the user can specify "go true" or "go false" to choose which path to follow, or can specify "assume (P)" to add the predicate P to PC, which may (or may not) resolve the condition. Finally, one can request that an "execution tree" be generated of all possible paths through a program. Since any program with loops potentially has arbitrarily long paths, this tree may in fact be infinite. EFFIGY allows the user to specify a bound on the height of the tree generated. Other systems, DISSECT (Howden, 1977), ATTEST (Clarke, 1976), and others typically provide comparable facilities.

The application of symbolic execution to program verification is as follows. First, one can look at the formulas associated with program variables and check that they are correct. One can also detect cases where a variable is used before it is assigned a value, and other such computation errors (Howden, 1977: 267).

Second, symbolic execution can sometimes reveal that a particular path is not executable because the PC for that path evaluates to false (i.e., is a contradiction), or it may reveal that some condition tested on a path is in fact redundant. The general problem of proving that an arbitrary predicate is a contradiction, or that one predicate implies another, is unsolvable; in practice, however, it is frequently possible to do this, ideally using an automated theorem prover.

A third use of symbolic execution is as an aid to domain analysis and test data selection (Clarke, 1976). Each path through a program defines an input subdomain, and symbolic execution of that path will generate the predicate PC defining that domain. Finally, symbolic execution can aid in the development of a formal proof of correctness (King, 1976: 391).

Symbolic evaluation has its advantages and disadvantages. It appears to be superior to ordinary testing overall and in particular is good at detecting computation and domain errors, although it still fails to detect most missing path errors (Howden, 1977: 277). The use of formulas instead of input/output pairs for some verification helps to guard against coincidental correctness. As an automated aid to other kinds of analysis (domain analysis and formal proofs), it has proved valuable.

On the negative side, symbolic execution involves substantial overhead to maintain and manipulate large and unwieldy symbolic formulae. The value of any automated system in particular will depend heavily on the sophistication of its formula manipulation and theorem-proving capabilities. Symbolic execution also has the disadvantages inherent to any abstract model of execution: execution of programs on real machines rarely conforms perfectly to the model (e.g., finite precision of machine computation). For this reason, the execution of programs with actual data remains a necessity. The ability of symbolic execution to aid in the selection of test data considerably mitigates this disadvantage. Finally, even executed symbolically a program can have an infinite number of paths and path domains, making any attempt at an exhaustive demonstration of correctness as futile as in the the usual testing paradigm.

III. Partition Analysis

Partition analysis is a technique that combines both formal verification techniques and testing in order to acquire confidence in a program's reliability. It is also distinctive in that it makes extensive use of both the specification and the implementation of a design, combining black and glass box techniques to a greater degree even than Howden's Functional Testing. Partition analysis can also be applied to two specifications at different levels of abstraction (e.g., a high-level design and a detailed design), making the technique applicable in earlier phases of software development.

Overview of Partition Analysis (Richardson, 1981: 25-50; Richardson and Clarke, 1985)

Partition Analysis consists of three steps: forming the procedure partition, partition analysis (formal) verification, and partition analysis testing. Forming the procedure partition is likewise a three step process.

First, symbolic execution is applied to the implementation to get a static representation of it. This differs from the more usual use of symbolic execution as an interpretive technique. The result is a set of input domains $D[P_J]$ and corresponding computations $C[P_J]$, one such pair for each path P_J, $J = 1, 2, \ldots, N$. This is called the implementation partition.

Next the specification is similarly analyzed to produce the specification partition, a set of input domains $D[S_I]$ and corresponding computations $C[S_I]$, one for each "subspecification" S_I, $I = 1, 2, \ldots, M$. It is assumed that a formal specification is available for this step, in some high-level notation. Richardson uses a specification language/PDL of her own design called SPA and extended the symbolic evaluation techniques to handle this language. Thus a subspecification is simply a path through the "code" of the specification.

Once the implementation and specification partitions have been derived, the procedure partition is formed by intersecting all implementation domains $D[P_J]$ and specification domains $D[S_I]$ to find all nonempty (overlapping) pairs, D_{IJ}. Corresponding to each pair is a "computation difference" C_{IJ} that represents the disparity (hopefully null) between the computations specified by $C[S_I]$ and $C[P_J]$. Domains in one partition that do not overlap any domain in the other partition indicate discrepancies between the implementation and the specification, and hence probable errors (possibly missing path). Analysis can continue by including these domains in the procedure partition as elements D_{I0} (for unpaired specification domains) and D_{0J} (for unpaired implementation domains).

Partition analysis verification takes the procedure partition and tries to prove for each domain D_{IJ} using standard proof techniques that C_{IJ} is null, that is, that $C[S_I]$ and $C[P_J]$ have the same effects on all elements of D_{IJ}. The equivalence problem is undecidable in general, so this step may or may not succeed fully. A proof of equivalence, however, is a very strong argument for the correctness of the implementation. A proof of nonequivalence identifies a fault in the program, and any counterexamples found provide fault-revealing test data.

The last step is partition analysis testing.

> The partition analysis method complements verification with testing. When the verification process is unsuccessful, testing may uncover errors or may increase confidence in the unproven equality relationships. When the verification process is successful, testing challenges or supports the conclusions drawn in the postulated environment of partition analysis verification (Richardson and Clarke, 1985: 1483).

In this step, the procedure partition subdomains and computations are used to generate test data. Glass box techniques for testing arithmetic manipulations, special, extremal and nonextremal cases are applied to the computations, while black box techniques such as domain testing are applied to the domains. Because both implementation and specification domains are being considered, domain testing will not only pick up path selection errors but also some missing path errors as well.

Loop Analysis

In order to derive static expressions for program paths, Richardson had to develop techniques for analyzing and representing loops. The description that follows is taken from Richardson's dissertation (Richardson, 1981), but has been modified to make certain aspects clearer.

The most general loop structure has the form

```
loop
   loop-body;
end loop;
```

where loop-body is a set of paths, some of which exit and some of which do not. Those that do all branch to just past the end of the loop; those that do not all branch back to the top. Thus the loop is a single-entry, single-exit construct (whether this general form is a structured construct can be argued either way). The standard while-loop can be written in this form as

```
loop
   if not while-condition then exit;
   loop-body;
end loop;
```

Richardson's loop technique depends upon knowing a priori the complete sequence of paths followed through the loop. This is a very strong condition that is frequently not met; thus the technique is not at all general. In many cases this condition is met, however, such as when a loop has only one path through it that stays in the loop and all the other paths exit it. Another workable case is when, say, the first iteration through the loop follows one path and all subsequent iterations follow another.

Figure 3-1 shows an example of a loop that cannot be analyzed this way. Procedure SEARCH performs a binary search for element X in a sorted array A and returns its index if found. On each iteration of the loop, X may be found or it may be determined not to be present; in either case the loop is exited. Two paths stay in the loop, corresponding to X being less than or greater than the current element being examined. One cannot determine the sequence of these two paths that will be followed without knowing at least some of the data values; hence, this loop cannot be analyzed by Richardson's technique.

```
procedure Search (A : in ARRAY_TYPE;
                  X : in ELEMENT_TYPE;
                  Found : out BOOLEAN;
                  Where : out INDEX_TYPE) is
      LB : INDEX_TYPE := A'First;
      UB : INDEX_TYPE := A'Last;
      Index : INDEX_TYPE;
   begin
      Found := FALSE;
      while LB  = UB loop
         Index := (LB + UB) / 2;
         if X < A(Index) then
            UB := Index - 1;
         elsif X > A(Index) then
            LB := Index + 1;
         else
            Found := TRUE;
            Where := Index;
            exit;
         end if;
      end loop;
   end Search;
```

Figure 3-1. An un-analyzable loop
Adapted from Darringer and King, 1978: 53

Assuming that a loop is analyzable, the first step is to associate with it an iteration counter k to count the number of loop iterations. Each path of the body of the loop is then executed one time to get a representation for the effects of one iteration of the loop, in the form of a set of recurrence relations

$$VarI_k = f\ (VarI_{k-1}, \ldots , VarN_{k-1});$$

for each VarI occurring in the loop, for each path. Each path also has associated with it a loop exit condition (lec) that is either true or false.

If the recurrence relations can be solved, then the result is a closed-form representation of the effects of the loop in terms of the iteration counter k and the initial values of the variables, $VarI_0$ through $VarN_0$. The loop can now be treated as a simple node in the program for each path that it appears on. In the case of nested loops, analysis starts with the innermost loop and proceeds outward. An example of the loop analysis procedure is provided below.

Empirically it has been found that it is wise to treat the case where the loop exits with k = 1 as a special case. Note that if a path exits the loop without changing any variables (as in the modified while loop above), then the k = 1 case also encompasses the traditional "fall through" case where the body of the loop is not executed at all.

A Partition Analysis Example

The following is an example of Partition Analysis [illegible] should clarify much of the preceding discussion [illegible] from Richardson, 1981: 279-303; Richardson and [illegible], 1985). The module to be analyzed is PRIME, [illegible] tion that takes a positive integer and returns TRUE if it is a prime and FALSE otherwise. Figure 3-2 is a specification for PRIME, written in SPA. It is based on the standard definition of primeness, that a number N is prime if and only if it is divisible only by 1 and N (and hence not by any number in the range 2 through N - 1). Figure 3-3 is an Ada implementation for PRIME which takes advantage of several additional properties, including the fact that if there is a factor at all, there must be one less than the square root of N. Figure 3-4 gives control flow graphs for both.

```
 procedure PRIME (N : in integer inset {1 ...})
                      return boolean =
s  begin
       return case
1                N = 1 =>
                    false;
2                N = 2 =>
3                   true;
4                otherwise ->
5                   forall {i: integer inset {2..N-1}|
6                         (N mod i /= 0)};
                 endcase;
     end PRIME;
```

Figure 3-2. Specification of PRIME
(Richardson and Clarke, 1985: 1478)

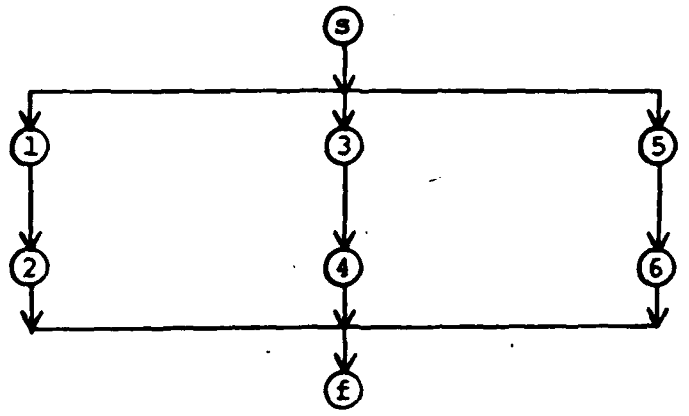
Figure 3-5 is the specification partition for PRIME. There are three paths through the specification, corresponding to the three conditions of the case statement. Note that the case statement is evaluated as if it were a series of if-then-elsif clauses, so that all the preceding conditions must be false in order to try the next. The condi-

```
    function Prime (N : INTEGER range 2..INTEGER'LAST)
            return BOOLEAN is
         Factor : INTEGER;
         IsPrime : BOOLEAN;
s     begin
1        if (N mod 2 = 0) or (N mod 3 = 0) then
2           IsPrime := (N < 4);
         else
3           IsPrime := TRUE;
4           Factor := 5;
5           while Factor ** 2 <= N loop
6              if (N mod Factor = 0)
                     or (N mod (Factor + 2) = 0) then
7                 IsPrime := FALSE;
                  exit;
               else
8                 Factor := Factor + 6;
               end if;
9           end loop;
         end if;
10       return IsPrime;
f     end Prime;
```

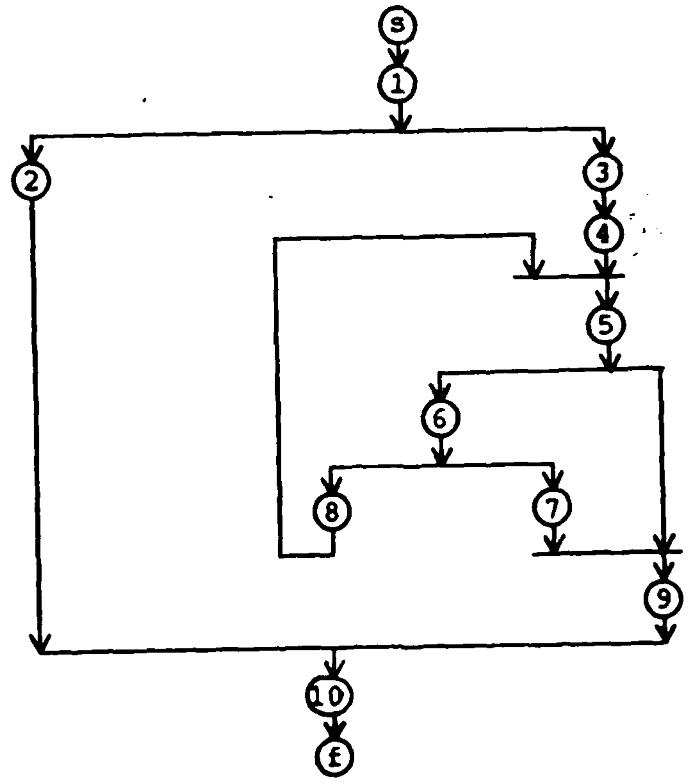
Figure 3-3. Implementation of PRIME
Adapted from Richardson, 1981: 24

tions attached to the three paths reflect this semantics. The only output variable is the value of the function, PRIME. In the first two cases this value is given explicitly. In the last case, it is represented by a formula.

The implementation of PRIME contains a loop, and Figures 3-6 and 3-7 give intermediate steps of the loop analysis required to get an expression for it. There are three paths through the loop, corresponding to the while condition evaluating to false, the if condition being true, and the else condition being true. The first two conditions result in the loop being exited, while the last one stays in the loop. Figure 3-6 gives symbolic representations for the



a. Specification Flow Graph



b. Implementation Flow Graph

Figure 3-4. Flow Graphs for PRIME

```
S1    : (s, 1, 2, f)
D[S1]: (n = 1)
C[S1]: Prime = false

S2    : (s, 1, 3, 4, f)
D[S2]: (n = 2)
C[S2]: Prime = true

S3    : (s, 1, 3, 5, 6, f)
D[S3]: (n >= 3)
C[S3]: Prime = forall {i := 2..n-1 | n mod i /= 0}
```

Figure 3-5. Specification Partition of PRIME

-- path (5, 9)

PC_K = PC_{K-1} and (Factor$_{K-1}$ ** 2 > N)

$IsPrime_K$ = $IsPrime_{K-1}$

$Factor_K$ = $Factor_{K-1}$

lec_K = true

-- path (5, 6, 7, 9)

PC_K = PC_{K-1} and (Factor$_{K-1}$ ** 2 <= N) and ((N mod Factor$_{K-1}$ = 0) or (N mod (Factor$_{K-1}$ + 2) = 0))

$IsPrime_K$ = false

$Factor_K$ = $Factor_{K-1}$

lec_K = true

-- path (5, 6, 8)

PC_K = PC_{K-1} and (Factor$_{K-1}$ ** 2 <= N) and (N mod Factor$_{K-1}$ /= 0) and (N mod (Factor$_{K-1}$ + 2) /= 0)

$IsPrime_K$ = $IsPrime_{K-1}$

$Factor_K$ = $Factor_{K-1}$ + 6

lec_K = false

Figure 3-6. Symbolic Execution of Loop in PRIME

path condition and variable values after some iteration k, in terms of the values from the previous iteration. Figure 3-7 then solves these recurrence relations in closed form for the special case where k = 1 and the general case where k >= 1. Once the loop analysis is complete, the full implementation partition can be constructed. This is given in Figure 3-8.

The last part of the first step of Partition Analysis is to form the procedure partition. The domain of each subspecification is compared with the domain of each implementation path, and any region of overlap defines a subdomain of the procedure partition. For each subdomain, a computation difference is computed by comparing the computations specified by the specification and the implementation for elements in that subdomain. Figure 3-9 gives the procedure partition for PRIME. Note that subspecification S_1 does not match any path domain (element D_{10} in Figure 3-9). This immediately reveals that the implementation fails to consider the case where N = 1. Note also that subspecification S_3 overlaps no less than five different path domains (elements D_{31}, D_{32}, D_{33}, D_{34}, and D_{35}).

Step two of Partition Analysis is formal verification that all computation differences in the procedure partition are null. Other than D_{10}, this can be done for PRIME. For D_{21}, for example, the domain is defined by (n = 2) and the

```
-- path (5,9)
    PC = PC and Factor$_0$ ** 2 > N
    Factor = Factor$_0$
    IsPrime = IsPrime$_0$

-- path (5, 6, 7, 9)
    PC = PC and (Factor$_0$ ** 2 <= N) and ((N mod
            Factor$_0$ = 0) or (N mod (Factor$_0$ + 2) = 0))
    Factor = Factor$_0$
    IsPrime = false

-- path ((5, 6, 8)+, 5, 9)
    PC = PC and exists {k := 2 ... | ((Factor$_0$ + 6*k
         - 6) ** 2 > N) and forall {i := 0 .. k-2 |
         ((Factor$_0$ + 6*i) ** 2 <= N) and (N mod
         (Factor$_0$ + 6*i) /= 0) and (N mod (Factor$_0$ +
         6*i + 2) /= 0)}}
    Factor  = Factor$_0$ + 6*k - 6
    IsPrime = IsPrime$_0$

-- path ((5, 6, 8)+, 5, 6, 7, 9)
    PC = PC and exists {k := 2 ... | ((Factor$_0$ + 6*k
       - 6) ** 2 <= N) and ((N mod (Factor$_0$ + 6*k - 6)
       = 0) or (N mod (Factor$_0$ + 6*k - 4) = 0)) and
       forall {i := 0 .. k-2 | (N mod (Factor$_0$ + 6*i)
       /= 0) and (N mod (Factor$_0$ + 6*i + 2) /= 0)}}
    Factor  = Factor$_0$ + 6*k - 6
    IsPrime = false
```

Figure 3-7. Loop Expression in PRIME

```
P1    : (s, 1, 2, 10, f)
D[P1]: (n >= 2) and (n mod 2 = 0 or n mod 3 = 0)
C[P1]: Prime = (n   4)

P2    : (s, 1, 3, 4, 5, 9, 10, f)
D[P2]: (n >= 2) and (n < 25) and (n mod 2 /= 0)
       and (n mod 3 /= 0)
C[P2]: Prime = true

P3    : (s, 1, 3, 4, 5, 6, 7, 9, 10, f)
D[P3]: (n >= 25) and (n mod 2 /= 0)
       and (n mod 3 /= 0) and ((n mod 5 = 0)
       or (n mod 7 = 0))
C[P3]: Prime = false

P4    : (s, 1, 3, 4, (5, 6, 8)+, 5, 9, 10, f)
D[P4]: (n >= 25) and (n mod 2 /= 0)
       and (n mod 3 /= 0) and exists {k := 2 ... |
       (6*k - 1) ** 2 > n and forall{i := 1..k-1 |
       (6*i - 1) ** 2 <= n)
       and (n mod (6*i - 1) /= 0)
       and (n mod (6*i + 1) /= 0)
C[P4]: Prime = true

P5    : (s, 1, 3, 4, (5, 6, 8)+, 5, 6, 7, 9, 10, f)
D[P5]: (n >= 121) and (n mod 2 /= 0)
       and (n mod 3 /= 0) and exists {k := 2 ... |
       (6*k - 1) ** 2 <= n)
       and ((n mod (6*k - 1) = 0)
           or (n mod (6*k + 1) = 0))
       and forall {i := 1 .. k-1 |
       (n mod (6*i - 1) /= 0)
       and (n mod (6*i + 1) /= 0)
C[P5]: Prime = false
```

Figure 3-8. Implementation Partition of PRIME

two computations are the constant value (true) and the predicate (n < 4), which evaluates to true within the domain. Proofs for the other domains are more complicated, and are not presented here.

Step 3 of Partition Analysis is test data selection, to further reinforce confidence in the correspondence between

```
D10: (n = 1)
C10: (false) vs. nothing

D21: (n = 2)
C21: (true) vs. (n   4)

D31: (n >= 3) and ((n mod 2 = 0) or (n mod 3 = 0))
C31: (forall {i := 2..n-1 | n mod i /= 0}
     vs. (n < 4)

D32: (n >= 3) and (n < 25) and (n mod 2 /= 0)
     and (n mod 3 /= 0)
C32: (forall {i := 2..n-1 | n mod i /= 0}
     vs. (true)

D33: (n >= 25) and (n mod 2 /= 0)
     and (n mod 3 /= 0) and ((n mod 5 = 0)
     or (n mod 7 = 0))
C33: (forall {i := 2..n-1 | n mod i /= 0}
     vs. (false)

D34: (n >= 25) and (n mod 2 /= 0)
     and (n mod 3 /= 0) and exists {k := 2 ... |
     (6*k - 1) ** 2 > n and forall{i := 1..k-1 |
     (6*i - 1) ** 2 <= n)
     and (n mod (6*i - 1) /= 0)
     and (n mod (6*i + 1) /= 0)

C34: (forall {i := 2..n-1 | n mod i /= 0}
     vs. (true)

D35: (n >= 121) and (n mod 2 /= 0)
     and (n mod 3 /= 0) and exists {k := 2 ... |
     (6*k - 1) ** 2 <= n)
     and ((n mod (6*k - 1) = 0)
         or (n mod (6*k + 1) = 0))
     and forall {i := 1 .. k-1 |
     (n mod (6*i - 1) /= 0)
     and (n mod (6*i + 1) /= 0)
C35: (forall {i := 2..n-1 | n mod i /= 0}
     vs. (false)
```

Figure 3-9. Procedure Partition of PRIME

the specification and implementation (especially in the case of a failure in the formal verification step), and also to demonstrate the run-time behavior of the program. Two criteria for selection are used, one for domain testing and one for computation testing. Domain testing focuses on the boundaries between subdomains defined in the procedure partition and chooses test data for each domain that is both ON a boundary (and hence in the domain) and also data that is OFF the boundary and not in the domain. OFF points are chosen to be as close to the boundary as possible, to minimize the maximum boundary displacement that would go undetected. Since PRIME deals with integer values only, OFF points can be selected that are immediately adjacent to each boundary.

Computation testing criteria focus on the computations performed within each domain and help to verify that the computation difference for each domain is null, even if the formal verification step failed. Details of the proof (or attempted proof) often provide guidance for finding good test points. The specific algebraic properties of the computations will also dictate which test data will be selected. Figure 3-10 gives some examples of the test data that would be selected for the subdomains of PRIME.

Performance of Partition Analysis

To get some idea for the effectiveness of Partition Analysis, Richardson used the technique on a set of 34 modules from the programming literature and textbooks, providing specifications for them as needed (Richardson and Clarke, 1985: 1486-1488). Most of the modules were correct or had only a few errors, so mutation analysis was used to

```
D10: Domain Testing Criterion:
     N = 1 (on), N = 0, N = 2 (off)
     Computation Testing Criterion:
     N = 1

D21: Domain Testing Criterion:
     N = 2 (on), N = 1, N = 3 (off)
     Computation Testing Criterion:
     N = 2

D31: Domain Testing Criterion:
     N = 2 (off), N = 3, N = 4 (on)
     N = 5, N = 7 (off), N = 6, N = 9 (on)
     Computation Testing Criterion:
     N = 3, N = 4, N = 1000
```

Figure 3-10. Sample Test Data for PRIME

generate large numbers of "mutant" variations of each module, each with one seeded error. Partition analysis successfully detected all of the errors that led to incorrect programs. There were a few mutants that correctly executed all the test data generated by the Partition Analysis procedure, and in each case it was shown that the mutant program was in fact equivalent to the correct one. Richardson admits that this evaluation is neither as rigorous nor as complete as it could be, but her results argue favorably for the effectiveness of the technique.

The next chapter explores extensions to Partition Analysis that expand its applicability by allowing its use on programs containing procedure and function calls. The case of recursive procedures and functions is also considered.

IV. Extensions to Partition Analysis

This chapter describes how Partition Analysis can be extended to apply to programs that use procedure and function calls. The first section presents several approaches to the general problem, while the second section addresses the special case of recursive procedures and functions. It is assumed that all procedures and functions have a single entry point and a single exit. Returns are treated as branches to a dummy node at the end of the routine to enforce this convention.

Procedure and Function Calls

During ordinary symbolic execution as described in Chapter 2, procedure and function calls do not present any special problems. When a call is encountered, arguments are bound to parameters, space for local variables allocated, and control transferred to the start of the called routine, just as in normal execution. At the end, output values are passed back to the calling routine and execution continues. The fact that some or all of the values being passed around are represented by symbolic expressions does not interfere with this process.

In Partition Analysis, however, the need to derive a static expression for a program causes some problems, and also presents some opportunities, in the handling of procedure and function calls. The most direct approach is to start with the bottom-level routines and derive expressions for them using the methods described in Chapter 3. Then in places where the bottom-level routines are called one can substitute these expressions, making parameter substitutions and any simplifications possible. This process can be repeated until the entire program has been analyzed.

Figures 4-1 through 4-4 illustrate this approach. Figure 4-1 is a function that returns the number of days in a calendar month. This and later examples make use of the predefined Ada package CALENDAR and the type declaration

```
type DATE_TYPE is record
     Day : DAY_NUMBER;
     Month : MONTH_NUMBER;
  end record;
```

Function DAYS_IN calls a boolean function LEAP_YEAR when it is determining the number of days in February. Figure 4-2 gives an implementation for this function. Analysis of LEAP_YEAR results in the expression of Figure 4-3. This expression is then used during the analysis of DAYS_IN to get the expression of Figure 4-4.

This approach is not without its disadvantages. First of all, during top-down development one might want to begin analysis and testing of routines before all of the lower ones are complete. Second, the implementation of a low-level routine may include details that are not relevant to the routine being tested. For example, numerical routines like SIN or SQRT are frequently implemented at iterations

```
function Days_in (Month : MONTH_NUMBER;
                   Year : YEAR_NUMBER)
                  return INTEGER is
s  begin
      case Month is
1        when 4 | 6 | 9 | 11 =>
2           return 30;
3        when 2 =>
4           if Leap_Year (Year) then
5              return 29;
            else
6              return 28;
         when others =>
7           return 31;
      end case;
f  end Days_in;
```

Figure 4-1. Implementation of Days_in

until some error bound is met. These routines certainly need to be tested, but in many cases an abstract view of these functions is sufficient and desirable. Finally, in the case of a routine that is called from many places, or for example a routine that is part of the definition of an abstract data type, it may be desirable to demonstrate the correctness of the routine separately one time (using Partition Analysis if it has a formal specification, or some other method), and then use some other way of referring to its function when it is called.

There are at least two ways to represent a procedure or function without presenting all the details of its implementation. First, one can use a formal specification that has been analyzed using Partition Analysis. Specifications are written at a higher level of abstraction and are usually

```
function Leap_Year (Yr : YEAR_NUMBER)
                          return BOOLEAN is
s  begin
1     if Yr mod 400 = 0 or (Yr mod 4 = 0
              and Yr mod 100 /= 0) then
2        return TRUE;
      else
3        return FALSE;
      end if;
f  end Leap_Year;
```

Figure 4-2. Implementation of Leap_Year

```
P1    : (s, 1, 2, f)
D[P1]: (yr mod 400 = 0) or ((yr mod 4 = 0)
         and (yr mod 100 /= 0))
C[P1]: Leap_Year = true

P2    : (1, 2, 3, f)
D[P2]: (yr mod 400 /= 0) and ((yr mod 4 /= 0)
         or (yr mod 100 = 0))
C[P2]: Leap_Year = false
```

Figure 4-3. Implementation Partition of Leap_Year

much simpler than the corresponding implementation. A specification is ideally also available before a module is written, permitting the analysis of incomplete programs. In practice the use of a specification expression is no different than using the implementation itself.

An alternative approach is even more abstract. If a routine is not yet written or even formally specified, or if it defines a well-known function (such as SIN or SQRT), it may be sufficient to represent it symbolically and give no indication at all during analysis of how it works. This approach is also appropriate for example in the case of an

```
P1    : (s, 1, 2, f)
D[P1]: (month in {4, 6, 9, 11})
C[P1]: Days_in = 30

P2    : (s, 3, 4, 5, f)
D[P2]: (month = 2) and ((year mod 400 = 0) or
       ((year mod 4 = 0) and (year mod 100 /= 0)))
C[P2]: Days_in = 29

P3    : (s, 3, 4, 6, f)
D[P3]: (month = 2) and (year mod 400 /= 0) and
       ((year mod 4 /= 0) or (year mod 100 = 0))
C[P3]: Days_in = 28

P4    : (s, 7, f)
D[P4]: (month in {1, 3, 5, 7, 8, 10, 12})
C[P4]: Days_in = 31
```

Figure 4-4. Implementation Partition of Days_in

abstract data type or when the correctness of a routine has been separately established.

This approach is easily implemented for function calls. When a function F is called, its parameters are replaced by the arguments given but the value returned by F is represented symbolically. For example, if variable X has the value a + b , then after the statement

```
Y := F (X);
```

the variable Y would have the value F (a + b) .

This approach is also illustrated in Figures 4-5 and 4-6. Figure 4-5 is the implementation of an integer function DAYS_BETWEEN that computes the number of days between two dates of the same year. It uses the function DAYS_IN for part of the computation. Figure 4-6 shows the implementation partition of DAYS_BETWEEN with DAYS_IN treated sym-

```
function Days_Between (Date1, Date2 : DATE_TYPE;
                       Year : YEAR_NUMBER)
                       return INTEGER is
      Difference : INTEGER;
      From, To : DATE_TYPE;
s   begin
1     if Date1.Month = Date2.Month then
2        Difference := abs (Date1.Day
                              - Date2.Day);
      else
3        if Date1.Month < Date2.Month then
4           From := Date1;
5           To := Date2;
         else
6           From := Date2;
7           To := Date1;
         end if;
8        Difference := 0;
9        for Mon in From.Month .. To.Month - 1
         loop
10          Difference := Difference
                             + Days_in (Mon, Year);
         end loop;
11       Difference := Difference + To.Day
                             - From.Day;
      end if;
12    return Difference;
f   end Days_Between;
```

Figure 4-5. Implementation of Days_Between

bolically. Note that if the implementation of DAYS_IN had been used instead, then the loop in DAYS_BETWEEN would have had three paths that remained in the loop, with no way of telling which path would be followed during each iteration, so that the loop would not have been analyzable by Partition Analysis. Thus the use of this abstraction technique has allowed a program to be analyzed that otherwise would have been too complicated, assuming that the correctness of DAYS_IN can also be established separately.

```
P1    : (s, 1, 2, 12, f)
D[P1]: (date1.month = date2.month)
C[P1]: Days_Between = abs (date1.day - date2.day)

P2    : (s, 1, 3, 4, 5, 8, 9, (10, 9)*, 11, 12, f)
D[P2]: (date1.month < date2.month)
C[P2]: Days_Between = sum (i := date1.month  ..
                        date2.month - 1 |
                        days_in (i, year))
                        + date2.day - date1.day

P3    : (s, 1, 3, 6, 7, 8, 9, (10, 9)*, 11, 12, f)
D[P3]: (date1.month > date2.month)
C[P3]: Days_Between = sum (i := date2.month  ..
                        date1.month - 1 |
                        days_in (i, year))
                        + date1.day - date2.day
```

Figure 4-6. Implementation Partition of Days_Between

Procedure calls can also be treated symbolically, but the notation is necessarily different. The proposed notation will allow a procedure to be represented functionally so that analysis can proceed.

A procedure can be viewed as a function that has an input vector (X_1, ... , X_N) and produces an output vector (Y_1, ... , Y_M). Parameters of mode "in" are part of the input vector, and parameters of mode "out" are part of the output vector. Parameter mode "in out" is in effect shorthand for an element that holds an input value and will also receive an output value. Such parameters are "split" and listed in both vectors. The output vector can be viewed as an unnamed record type that the procedure "returns".

To represent a procedure call symbolically, the input arguments are bound to the procedure's input parameters, and the values of the output parameters are represented by the procedure name and an output parameter name, using the usual syntax for representing components of a record. For example, the predefined Ada package CALENDAR include a procedure with the specification

```
procedure SPLIT (Date : in TIME;
                 Year : out YEAR_NUMBER;
                 Month : out MONTH_NUMBER;
                 Day : out DAY_NUMBER;
                 Second : out DAY_DURATION);
```

A call to this procedure might look like

```
Split (Todays_Date, Current_Year, Current_Month,
       Current_Day, Current_Time);
```

The effect of this call in the proposed notation would be

```
Current_Year = Split.Year (v (Todays_Date))
Current_Month = Split.Month (v (Todays_Date))
Current_Day = Split.Day (v (Todays_Date))
Current_Time = Split.Second (v (Todays_Date))
```

where v (Todays_Date) stands for the symbolic value of Todays_Date at the time of the call. Analysis can now proceed using these values.

Recursive Procedures and Functions

If a procedure or function is recursive, it cannot be analyzed directly by ordinary Partition Analysis. A simple technique, however, permits analysis in many cases.

The approach to take is analogous to the loop analysis technique presented in Chapter 3. All paths through the routine are identified and symbolically executed. Recursive calls are represented symbolically as described above. The result is a set of recurrence relations consisting of a number of base cases (paths without recursive calls) and a number of recursive cases (paths with such calls). As in the case of loop analysis, these relations are then converted to a closed-form expression to be used if desired wherever the routine is called, or in the further application of Partition Analysis (i.e., the formal verification and testing procedure) to the routine itself.

A short example of this kind of analysis follows. Figure 4-7 is a recursive function for computing factorials. Figure 4-8 gives the recurrence relations derived from the symbolic execution of this routine, including the notation for representing recursive calls. For this example, it is then trivial to show that these relations correspond to the standard definition of factorial:

Factorial (N) = product (i := 1 .. N | i)

Restrictions analogous to those on loops apply to the analysis of recursive procedures and functions. If a recursive routine has several paths that contain recursive calls such that the sequence of paths followed is data dependent, then the recurrence relations derived for the routine will not be solvable (they may not be anyway). For example, a routine that searches a binary search tree by calling itself recursively on either the left or right subtree until either the value being sought or a leaf node is found cannot be

```
function Factorial (N : INTEGER) return INTEGER is
s  begin
1     if (N = 0) or (N = 1) then
2        return 1;
      else
3        return N * Factorial (N - 1);
      end if;
f  end Factorial
```

Figure 4-7. Implementation of Factorial

```
P1: (s, 1, 2, f)
PC = PC and ((n = 0) or (n = 1))
Factorial = 1

P2: (s, 1, 3, f)
PC = PC and (n /= 0) and (n /= 1)
Factorial = n * Factorial (n - 1)
```

Figure 4-8. Recurrence Relations for Factorial

fully analyzed because the sequence of "left" and "right" moves cannot be determined at analysis time.

Of course, this only applies if a closed-form solution is truly necessary: the need for such a solution is not always present. Frequently a routine implemented recursively has a specification that is also recursive. Thus the recurrence relations alone may be sufficient to prove compliance with the specification during the formal verification phase. In the example above, the recurrence relations derived from the implementation clearly correspond to the usual recursive definition of the factorial function.

Whether a closed-form expression is used or not, the selection of data during the testing phase to test each subdomain, that is, each path through the routine, guarantees that all base cases and all recursive cases will be exercised. Hence the traditional guidelines for testing recursive routines are subsumed by the Partition Analysis method.

Application of Partition Analysis to Whole Programs

The purpose of extending Partition Analysis to include procedure and function calls was to be able to analyze entire programs. While the unavailability of I/O, limitations on loops, and complexity of the method when carried out manually preclude any meaningful example from being given, this section outlines one procedure that could be applied once these other problems are solved.

Partition Analysis seems to lend itself best to a bottom-up testing approach. During unit testing individual modules can be analyzed and compared with their specifications, and also tested using suitable driver routines. This is not very different than current practice; the point is that the application of Partition Analysis will make it more systematic and thorough. If a "unit" in fact contains procedures and functions of its own, then they will need to be analyzed first. Further, routines that are called by many units can be symbolically executed once and the resulting expressions placed in a library for use in later analyses.

As units are combined into larger entities, it will become desirable, perhaps crucial, to switch to one of the more abstract (and compact) representations for the various low-level routines, either through the use of specifications or of symbolic representations. The choice of representation is not arbitrary, however. During the formal verification phase in particular, if a symbolic representation is used, there must be enough semantic information available to manipulate formulas containing that representation. For example, during symbolic execution in general it is always assumed that enough is known about operators like "+" and "mod" to be able to simplify and compare formulas containing these symbols; the same must be true for user-defined functions and procedures.

How much information is enough will depend on the application. For example, a function INTEGRATE that computes definite integrals given an arbitrary real-valued function and an arbitrary real interval does not need to know anything about the function beyond the basics that it is computable, defined on the interval, and returns a real value. In such a case, use of a symbolic representation such as SIN(X) is appropriate. In other contexts it may be necessary to know more about the function, such as that SIN(-X) = - SIN(X), or that SIN(2*X) = 2 * SIN(X) * COS(X).

One source of information about a routine at a reasonably abstract level is the specification partition, as long as the implementation is at some time shown consistent with it. If even more information is needed, the full procedure partition itself can be used. Semantic information can be "held in reserve" until needed, by using a symbolic representation and introducing semantic information only where needed in a proof, or it can be included directly by substituting the appropriate expressions into the formulas of the high-level routine being analyzed.

During the test data selection step it is also advised to use some of the information gathered during the analysis of the low-level routines. Specifically, in order to test fully all interfaces and all paths through the program (up to loop iterations and recursive calls), domain information from the procedure partitions of the low-level routines should be used. An example is the best way to illustrate this. A program that made use of factorials might have several domains whose computation includes a call to the FACTORIAL function above. If these calls are represented symbolically, then during test data selection each of these domains should be further subdivided into one where the argument to FACTORIAL is one and another where it is two or more. This inclusion of low-level domain information at higher levels of the program helps to maintain confidence that the sum of the parts is correct, and not just the parts themselves. It also results in a test suite that exercises the entire program, yet was developed in the process of testing the program incrementally.

V. Conclusions and Recommendations

This chapter offers conclusions concerning Partition Analysis in terms of its effectiveness and scope of applicability. It also presents recommendations for future work.

Conclusions

The basic problem of handling procedure and function calls was solved. A problem many verification procedures have, that of getting extremely unwieldy even for programs of modest size, was also addressed. This problem was not solved -- it seems highly unlikely that any effective method of verification will be quick and easy to apply -- but techniques were suggested for controlling some of this explosive increase in complexity. The straightforward approach of direct inclusion of subroutines (in effect, in-line expansion) quickly gets very large, as expected, but abstract representations can greatly simplify in particular deriving the procedure partition. To the extent that information from the specification and/or implementation is reintroduced, the formal verification step will approach the complexity of having used the direct representation in the first place, but it will never be worse than that, and it may remain considerably simpler. In the testing phase only domain information is reintroduced, so a simplification has occurred here as well.

Looking at the method as a whole, Partition Analysis is largely language-independent. Although all of the examples presented here were in Ada, in the original work Richardson presented examples worked out in Ada, Pascal, and FORTRAN. In the course of the work presented here on analyzing recursive routines, examples written in Pascal and Ada were successfully analyzed. Some examples in LISP were also tried, with mixed success. The main difficulty with LISP is the application of semantic information regarding built-in operators such as CAR, CDR, CONS, and so forth, in order to manipulate formulas containing them, and also the recursive nature of S-expressions, which seems to demand an induction proof in most cases. Such difficulties make analysis harder but not impossible. The above languages and similar ones by far represent the bulk of the software being written today.

Partition Analysis seems best suited for general scientific, engineering, and mathematical applications. Some specialized applications such as compilers, operating systems, database and graphics have special techniques of their own that are used for software development. Partition Analysis as a general tool is not well suited to such areas. Partition Analysis also seems to be inappropriate to verifying complex human-computer interfaces, this being a poorly-understood process that is still very much an art. For embedded computer systems, Partition Analysis can usefully analyze many of the algorithms used, and the testing phase will demonstrate the full range of runtime behavior. However, much embedded software uses control structures such as interrupts, coroutines or parallel execution that are beyond the scope of the method.

Partition Analysis works best in a traditional software development life cycle of requirements definition, specification and high level design, detailed design and coding, and unit and integration testing. It is also consistent with the use of an object-oriented design methodology. Implementations of abstract objects can be proven consistent with their specifications and then treated as primitive objects during subsequent analysis as explained in Chapter 4. Partition Analysis is less well suited to a rapid prototyping environment where the requirements are ill-defined and rapidly changing, since formal specifications play such a central role in the method.

As indicated briefly in Chapter 3, early empirical studies indicate that Partition Analysis is extremely effective at detecting program faults. What makes Partition Analysis testing superior to other techniques? Black box testing looks at the input domain and finds tests that thoroughly exercise a correct program, but possibly not an incorrect one. It also ignores some distinctions within input subdomains that are unique to the implementation, and thus may fail to test some relevant cases. Black box testing can do a reasonable job of finding path selection errors, but in general cannot find missing path errors or computation errors effectively.

Glass box testing, on the other hand, looks primarily at the implementation, devising tests that thoroughly exercise whatever the program does, but not necessarily what it is supposed to do, using the specification only as an "oracle" to distinguish right from wrong answers. This characterization applies to Functional Testing as well. Glass box testing is generally good at finding computation errors and path selection errors, but again cannot detect missing path errors in general because the specification is largely ignored.

Partition analysis is more successful at finding errors because it gives equal weight and equal effort to the analyses of the specification and the implementation. This analysis is followed by an attempt at a formal proof, where difficulties or counterexamples will point out faults, and also by extensive testing of all relevant domains in the program, using both black and glass box techniques.

Future Directions

The original Partition Analysis method was very restricted in the language constructs it could handle; it needs to be able to handle the full range of constructs encountered in real programs. The work in Chapter 4 on procedures and functions is a step in this direction. The method desperately needs to be tried on larger examples, to judge better the effectiveness of the method as a whole and to validate the procedure described in Chapter 4 for analyzing whole programs. If done entirely by hand, however, this is impractical. Some automated tools exist, for example to do symbolic execution, automatic theorem proving, and test case execution and monitoring, but these were not available for this thesis. The unavailability of I/O and especially the limited power of existing loop analysis techniques further complicate matters. Any future work directed in any of these areas (use of existing automated tools, inclusion of I/O or new loop analysis techniques) would have considerable value, especially if it included an analysis of larger programs than has heretofore been done. Automation of the Partition Analysis procedure itself would be premature, given the heuristic nature of several parts, in particular the verification and test data selection steps.

Bibliography

Clarke, L. A. "A System to Generate Test Data and Symbolically Execute programs," IEEE Transactions on Software Engineering SE-2 (3): 215-222 (September 1976).

Clarke, L. A., J. Hassell and D. J. Richardson. "A Close Look at Domain Testing," IEEE Transactions on Software Engineering SE-8 (4): 380-390 (July, 1982).

Darringer, J. A. and J. C. King. "Applications of Symbolic Execution to Program Testing," IEEE Computer 11 (4): 51-60 (April 1978).

Dijkstra, E. W. "Notes on structured programming," in Structured Programming, O-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Ed. New York: Academic Press, 1972, pp. 1-81.

Dunn, R. H. Software Defect Removal. New York: McGraw-Hill, Inc., 1984.

Hoare, C. A. R. "An Axiomatic Basic for Computer Programming," Communications of the ACM 12 (10): 576-580+ (October 1969).

Howden, W. "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering SE-3 (4): 266-278 (July 1977).

-----. "Functional Program Testing," IEEE Transactions on Software Engineering SE-6 (2): 162-169 (March 1980).

-----. "A Functional Approach to Program Testing and Analysis," IEEE Transactions on Software Engineering SE-12 (10): 997-1005 (October 1986).

King, J. C. "Symbolic Execution and Program Testing," Communications of the ACM 19 (7): 385-394 (July 1976).

Mostow, J. "What is AI? And What Does It Have to Do with Software Engineering?" IEEE Transactions on Software Engineering SE-11 (11): 1253-1256 (November 1985).

Myers, G. J. The Art of Software Testing. John Wiley & Sons, 1979.

Pressman, R. S. Software Engineering: a Practitioner's Approach (Second Edition). New York: McGraw-Hill, Inc., 1987.

Richardson, D. J. "A Partition Analysis Method to Demonstrate Program Reliability." PhD. Dissertation, University of Massachusetts, Amherst, September 1981.

Richardson, D. J. and L. A. Clarke. "Partition Analysis: A Method Combining Testing and Verification," IEEE Transactions on Software Engineering SE-11 (12): 1477-1490 (December 1985).

Weyuker, E. and T. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Transactions on Software Engineering SE-6 (3): 236-246 (May 1980).

White, L. J. and E. I. Cohen. "A Domain Strategy for Computer Program Testing," IEEE Transactions on Software Engineering SE-6 (3): 247-257 (May 1980).

VITA

Robert P. Graham, Jr. was born on 5 March 1964 in Newark, Delaware. He graduated from high school in North East, Maryland, in 1981. He then attended Virginia Polytechnic Institute and State University under an Air Force ROTC 4-year scholarship. He graduated Summa Cum Laude in 1986 and received the Bachelor of Science degree in Computer Science, in Honors. During undergraduate school he also worked, through a cooperative education program, for the Air Force Data Services Center, Pentagon, for a total of 18 months. In June, 1986 he received his commission in the USAF and reported to the School of Engineering, Air Force Institute of Technology, as his first assignment.

Permanent Address: R.D. 1, Box 161AA
Reedsville, PA 17084

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
AFIT/GCS/MA/88-02

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
School of Engineering

6b. OFFICE SYMBOL *(If applicable)*
AFIT/ENC

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS *(City, State, and ZIP Code)*
Air Force Institute of Technology (AU)
Wright-Patterson AFB OH 45433-6583

7b. ADDRESS *(City, State, and ZIP Code)*

8a. NAME OF FUNDING / SPONSORING ORGANIZATION

8b. OFFICE SYMBOL *(If applicable)*

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS *(City, State, and ZIP Code)*

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Software Verification Using Partition Analysis

12. PERSONAL AUTHOR(S)
Robert P. Graham, B.S., Second Lieutenant, USAF

13a. TYPE OF REPORT
MS Thesis

13b. TIME COVERED
FROM ________ TO ______

14. DATE OF REPORT *(Year, Month, Day)*
1988 March

15. PAGE COUNT
69

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
computer program testing, computer program verification, partition analysis

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Thesis Chairman: Panna Nagarsenker, Ph.D.
Associate Professor of Mathematics and Computer Science

Abstract: see reverse

Approved for public release: IAW AFR 190-1.
[illegible] 14 Mar 88
[illegible] Development
[illegible] (ATC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT.] ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
Panna Nagarsenker

22b. TELEPHONE *(Include Area Code)*
513-255-[illegible]

22c. OFFICE SYMBOL
AFIT/ENC

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. Abstract

Software verification is the process of determining whether a piece of software is reliable--whether it performs as it is supposed to. As traditionally performed, program verification can account for 40 percent or more of the development time and cost of a software product. In spite of this fact, released software is notorious for its unreliability. These two facts, the expense of our attempts at program verification and our limited success, have sustained a great deal of research interest directed at finding more effective methods.

This thesis develops extensions to a promising new verification technique called Partition Analysis, developed by Debra J. Richardson (1981). Partition Analysis appears to be a powerful approach for identifying program faults, but in its current state can only be applied to single program modules that produce no side effects, including input or output. This thesis extends the applicability of Partition Analysis by permitting the use of procedure and function calls, thereby allowing complete programs to be analyzed. The result is a set of techniques for handling regular, non-recursive procedure and function calls, separate methods for the analysis of recursive procedures and functions, and an approach to the larger problem of analyzing entire programs.

END
DATE
FILMED

4-88
DTIC