Applications of Feasible Path Analysis to Program Testing

Allen Goldberg, T.C. Wang, David Zimmerman*
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304

{goldberg, wang}@kestrel.edu
zim@stc.lockheed.com

Abstract

For certain structural testing criteria a significant proportion of tests instances are infeasible in the sense the semantics of the program implies that test data cannot be constructed that meet the test requirement. This paper describes the design and prototype implementation of a structural testing system that uses a theorem prover to determine feasibility of testing requirements and to optimize the number of test cases required to achieve test coverage. Using this approach, we were able to accurately and efficiently determine path feasibility for moderately-sized program units of production code written in a subset of Ada. On these problems, the computer solutions were obtained much faster and with greater accuracy then manual analysis. The paper describes how we formalize test criteria as control flow graph path expressions; how the criteria are mapped to logic formulas; and how we control the complexity of the inference task. It describes the limitations of the system and proposals for its improvement as well as other applications of the analysis.

* Work performed by the author while affiliated with Kestrel Institute. Author's current affiliation is the Software Technology Center, Research and Development Division, Lockheed Missiles and Space Company, Palo Alto, CA.

1. Introduction

Given a description of a set of control flow paths through a procedure, *feasible path analysis (FPA)* determines if there is input data which causes execution to flow down some path in the collection. FPA is central to most applications of program analysis. But, because this problem is formally unsolvable, syntactic-based approximations are used in its place. For example, the dead-code analysis problem is to determine if there are any input values which cause execution to reach a specified program point. The approximation determines whether there is a control flow path from the start of the program to the point. This syntactic approximation is efficiently computable and conservative: if there is no such path the program point is clearly unreachable, but if there is such a path, the analysis is inconclusive, and the code is assumed to be live.

Such conservative analysis too often yields unsatisfactory results because the approximation is too weak. As another example, consider data flow analysis. A *du-pair* is a pair of program points such that the first point is a definition of a variable and the second point a use and for which there exists a definition-free path from the definition to the use. The sharper, semantic definition of a du-pair requires that there be a *feasible* definition-free path from the definition to the use. A compiler using du-pairs for detecting dead variables may miss optimizations by not considering feasibility. Similarly, a program analyzer computing program slices to merge parallel versions may report conflicts where none exist.

In the context of software testing, feasibility analysis plays an important role in identifying testing requirements which are infeasible. This is especially true for data flow testing and modified condition/decision coverage. This will be discussed in section 3.

Syntactic approximations are generally used because there is a jump from the near linear complexity of syntactic methods to intractable complexity of semantic criteria. In this paper we describe our approach to implementing feasible path analysis, and provide experimental evidence that it can be both efficient and effective for many applications of program analysis. The analysis is not (nor can it be) exact, but is conservative in that only those paths which can be proved infeasible are identified as such. Feasible path analysis may thus enhance program analysis systems, trading additional computation for sharper analytical results.

Feasible path analysis requires both *symbolic analysis* and *theorem proving*. Symbolic analysis relates expressions occurring at different program points and theorem proving determines the validity of logical relations between expressions. The relevance of symbolic analysis and theorem proving to testing and analysis is well known**.** Traditionally, the inefficiency of such methods precluded their applicability to practical problems. At Kestrel Institute we have been developing a high-performing theorem prover [Wang87, Wang92] which is used for verification and program transformation. The prover has been effectively adopted to feasible path analysis because most inference problems encountered in practice are *broad and shallow*. The proofs typically require just a few inference steps, but the formulas passed to the prover are generally very large, with much of the information being irrelevant. It is precisely this kind of problem for which automated inference is cost effective compared with manual analysis.

Theorem provers are notoriously difficult for people to use interactively. In our work all aspects of the prover, other than the setting of high-level strategies and resource bounds, are invisible to the user. Our prover runs completely automatically, generally taking on the order of seconds for each inference task.

Consider the Ada procedure in Appendix A. This problem is generally illustrative of the sample problems we have treated except for the fact that all the variables are Boolean-valued. (The example was chosen to illustrate why a standard theorem proving technique of placing formulas is conjunctive normal form is a poor strategy in this context.) A typical problem that may be posed is: if underlined statements are executed is the value of `dc-14` (and hence the outcome of the `if statement)` determined at point E? A reader attempting a manual analysis of this problem will quickly get a sense of the complexity of the task. Appendix B contains a logical formula derived by symbolic analysis which captures this problem as a theorem proving task.

The main body of this paper describes our design and implementation of feasible path analysis for a very restricted subset of Ada. While limitations on the Ada subset limit the applicability of the tool, this work points the way to developing capabilities which can have pervasive impacts on software testing, re-engineering and other tasks.

The next section describes our approach to FPA. In section 3 we describe results of its application to structural (white-box) testing and mention other applications. A companion paper [] describes some specific applications to testing. In section 4 we describe related work and in section 5 our future plans.

2. Feasible Path Analysis

A control flow path through a procedure is *feasible* if there is an assignment to input values, (i.e. global variables and parameters) which drives execution down the path. A set of control flow paths is *feasible* if one of its members is feasible. We use a regular expression to describe a set of paths. Such an expression is called a *path regular expression* (PRE).

Feasible path analysis, given

- A procedure and associated type, constant, and variable declarations
- A path regular expression *P* from a start point *s* to an end point *e*
- A formula φ,

attempts to determine if there is a computation state, that is, values for program variables, at *s*, which drives execution down one of the paths denoted by *P*, such that the formula φ evaluated at point *e* is true. Typically *s* is the entry to the procedure and we seek values of global variables and parameters with the required properties.

FPA constructs a formula in first order logic which is satisfiable if and only if there are inputs meeting the specification above. Construction of the formula and related axioms is called *symbolic evaluation*. A theorem prover is invoked to test the satisfiability of the constructed formula. The output may be yes, no, or inconclusive if the prover does not yield a definitive result within the resource bounds imposed.

2.1. System Architecture

Figure 1 illustrates the basic architecture of the FPA system. The system incorporates components of *Software Refinery* and *Refine/Ada* [Reasoning92], products available from Reasoning Systems, Incorporated. *Software Refinery* is an environment for the transformation and manipulation of the abstract syntax trees stored in an object base, and for graphical and mouse sensitive display of these entities. *Refine/Ada* provides a parser, static analyzer, control flow graph, and analysis capabilities, such as data flow analysis, for Ada. Our system is implemented in Lisp and Refine [Reasoning90]. Some of the features which make Refine a good environment for building a testing system are described in [Kotik89].

In this architecture we assume the existence of some client, such as the Test Specification and Determination Tool [Jasper94], which generates requests for FPA using the interface illustrated above. Prior to processing such requests, the Ada procedure and associated definitions are parsed to an abstract syntax tree, any transformations such as loop unrolling or

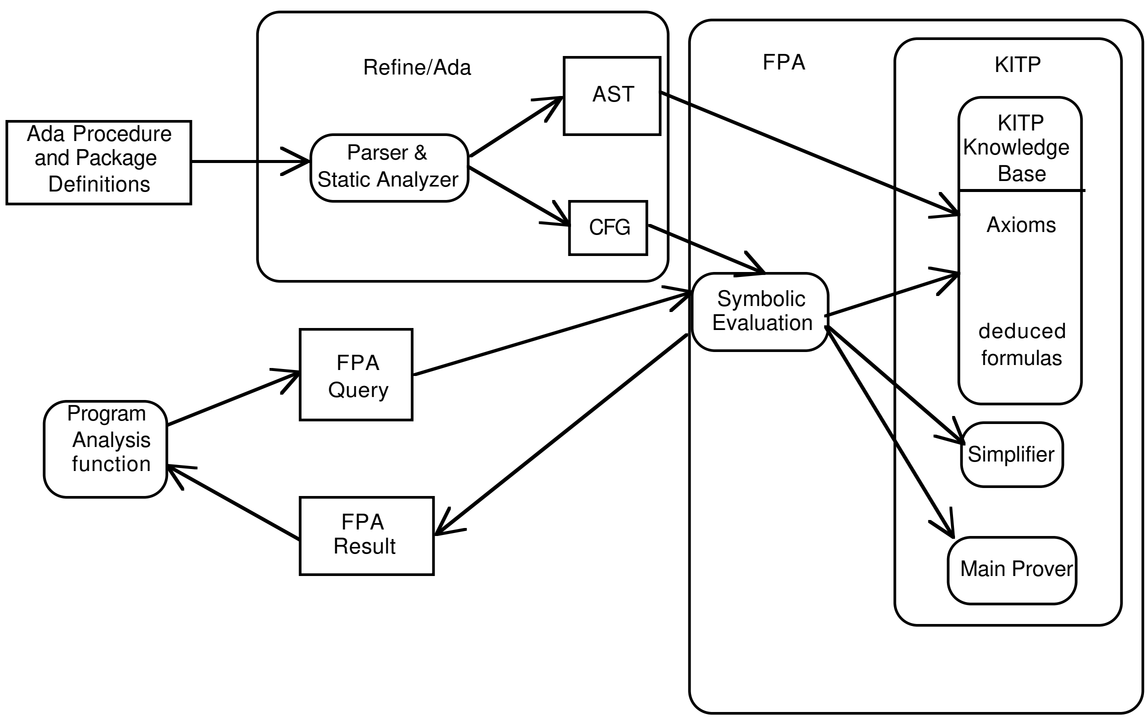


Figure 1. Basic FPA System Architecture

function inlining are performed, and a control flow graph (CFG) is constructed. The symbolic analysis component then analyzes relevant type, variable and constant declarations to generate a set of axioms about the types and operations used in the procedure. Next the symbolic analyzer examines each basic block (represented as a node in the CFG) and captures its semantics as a set of equations relating the values of variables on entry to the block to their value on exit. FPA requests are processed by generating a formula in first-order logic and testing its satisfiability with the theorem prover.

2.1.1. Subset of Ada Control Constructs

Unlike syntactic analysis, feasible path analysis requires formalization of program semantics. Often such formalizations are unavailable, or are incompatible with first-order logic. In this work we have employed straightforward formalizations of rudimentary Ada features, including functions and procedure calls, assignment, `case` and `if` statements. Ada tasks and exceptions are not treated.

Procedure calls are treated in one of two ways. The procedure may be unfolded (inlined), which increases the size and complexity of the formulas, but enhances the analysis. A procedure may also be treated as a black box by determining which global variables and parameters it modifies, and assuming that it modifies each of them in some arbitrary way.

Limited facilities are currently provided for treating loops. If the loop does not set variables which affect control flow or the value of φ, then it raises no difficulties. If the loop consists of a statically-determinable finite enumeration it is unrolled. In section 5.2 we describe a future approach to more comprehensive treatment of loops. Otherwise loops are assumed to modify assigned variables in some arbitrary way.

Our motivation—to demonstrate the effectiveness of techniques on production, safety-critical code— has been fortuitous, since safety-critical software is often written in such restricted subsets for understandability and predictability of run-time performance. While extending the analysis to a richer set of programming language constructs is non-trivial, once done, we see no reason why the effectiveness of the analysis will degrade. A possible exception is high-order functions; their treatment requires enhancements to our first-order prover.

2.2. Symbolic Evaluation

The task of symbolic evaluation is to construct first-order logic formulas to pass to the theorem prover. Some of these formulas are data type axioms derived from type declarations. Others are derived from Ada expressions, usually in control conditions within the procedural code.

Variables in an inference system are referentially transparent. Unlike program variables, their denotation does not depend on a store of values. To deal with this difference, each program variable is represented as a family of logic variables within the theorem proving environment. The family is indexed by the program points where their values may differ. "Backward" symbolic evaluation is used to generate assertions which relate the values of variables at different program points.

2.2.1. Data Representation

Ada data types and operations must be represented and reasoned about by the theorem prover. For each Ada data operation the system constructs a corresponding function in a first-order logical theory to represent the operation. The semantics of the operations are characterized by axioms about the associated functions. Thus the task of representation is two-fold: first to translate Ada expressions into the corresponding expressions in the logic and second to generate axioms, statically for built in types, and dynamically for user-defined types. Axioms are generated for relevant constant, variable and type definitions directly from the Ada source code.

2.2.1.1. Integers

Our axiomatization of the integers formalizes the standard properties of addition, multiplication, equality, inequality, and other Ada arithmetic operations. The axioms are expressed in an optimized form for effective theorem proving performance, because arithmetic reasoning is in general a difficult task. The axiomatization assumes that integer values are unbounded. This is a pragmatic choice in accordance with our overall conservative approach. Specifically, by omitting integer overflow restrictions, we may miss identifying an infeasible path, but we will never make an incorrect assertion of infeasibility.

Explicitly declared subranges are axiomitized as inequalities.

2.2.1.2. Enumerations and Subranges

Enumeration types are mapped into integer subranges, with operations such as equality and successor mapped into their integer equivalents. For subranges, inequalities or a disjunction of equalities are asserted.

2.2.1.3. Real Numbers

Axiomatization of the properties of floating point numbers is difficult. Our approach is to only formalize those properties of the reals which are also true about their representation as floating point numbers. For example, associativity of addition is not part of the axiomatization but the transitivity of *less than* is.

2.2.1.4. Arrays

The primary difficulty with arrays is that since the indices used in array assignments can be arbitrary expressions, one does not in general know whether consecutive assignments to an array involve the same element. We model such assignments through a symbolic history list which represents sequential assignments to individual elements or index subranges. The history list is built from nested functions over which the prover can reason and apply rewriting rules. Some specific examples of this basic approach are presented in [Jasper94]. The approach can be extended to model multi-dimensional arrays, as well as operations such as array slicing and array aggregate definition.

2.2.1.5. Records

Ada records are represented by a construction function which builds an aggregate from component fields, along with a set of projection functions which select individual fields from an aggregate.

2.2.2. Back Substitution

A control flow graph (CFG) is a directed graph in which nodes represent *basic blocks* and edges represent control flow. A basic block is a segment of code with a single entry point and a single exit point. Control flow path sets may be expressed as regular expressions over the alphabet of control flow nodes (CFNs). The graph itself is *language neutral*.

In imperative languages such as Ada, each CFN is associated with a sequence of assignment statements, possibly terminated by a test predicate which controls a multi-way branch, such as from an `if` or `case` statement. In the discussion below we restrict our attention to `if` statements. `Case` statements are treated similarly.

The point of symbolic evaluation is to logically relate the values of expressions at different program points. To work within a logical framework we introduce for each global *program* variable a *logical* variable which denotes its value at either the start or end of a CFN; strictly local variables are replaced through substitutions. Thus in node n referencing global variable v, v_n denotes the variable's value at exit to the associated code block n and v_n^* denotes its value upon entering the block.

A naive implementation of this model will introduce a plethora of logical variables. We optimize logical variable introduction by only introducing those variables that are a logical necessity. A program variable's value at program points p and q can be represented by the same logical variable if for every execution of the program that reaches p and q the value of the program variable is the same at both points. In particular there need only be one more logical variables than assignments to the variable.

By applying the well-known Hoare axiom for assignment [Hoare69], a set of equations, called the *defining equations* for CFN n, are constructed which give the value of a variable at the end of the block associated with n in terms of an expression over variables at the start of a block. We denote the set of these equations for a CFN n by σ_n. If a node contains a call to procedure F (which has not been inlined), then let $r_1 \cdots r_n$ be the variables that F may potentially modify. Let $f_1 \cdots f_n$ be new variables whose type is the same as $r_1 \cdots r_n$ respectively. Then the equations $r_i = f_i$ are asserted. Since $f_1 \cdots f_n$ are unconstrained by any axioms other then type axioms, this asserts that the value of the outputs of F have been modified in an arbitrary way.

Let r be a regular expression denoting a collection of paths from the entry to CFN s to the exit of CFN e. Then $P(s, e, r, \phi)$ denotes a logical formula over free variables v_s^* which is satisfiable if and only if there is a state *S* (a map of program variables to values), such that if execution is started at the start of *s* in state *S* then control will reach the exit of *e* along a path denoted by *r* and in the resulting state, ϕ evaluates to true. The formula $P(s,e,r,\phi)$ can be computed by expressing *P* recursively in its argument *r*. If ϕ is a formula and σ a set of equations of the form $x_i = e_i$ where x_i is a variable and e_i is an expression, then $\varphi[\sigma]$ is the formula obtained by the simultaneous substitution of e_i for x_i in ϕ.

As a base case consider $P(s,s,s,\varphi)$, i.e. the execution of a single CFN *s*.

$$P(s,s,s,\varphi) = \varphi[\sigma_s].$$

If *r* is the alternation of two expression r_1 and r_2 then

$$P(s,e,r_1 | r_2, \varphi) = P(s,e,r_1,\varphi)''\vee''P(s,e,r_2,\varphi)$$

We have written the disjunction in quotes to emphasize that it denotes the *expression* formed as the disjunction of two formulas, not the boolean evaluation of two truth values.

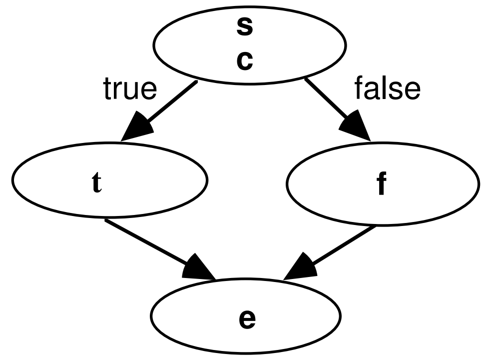
Similarly, suppose $r = r_1 \circ r_2$, such that terminus of r_1 is *m*, the start of r_2 is *n*, and in order to exit *m* and flow to *n* the condition *c* must be true. Then

$$P(s,e,r_1 \circ r_2, \varphi) =$$
$$P(s,m,r_1,c)''\&''P(n,e,r_2,\varphi)[\bar{v}_m = \bar{v}_n^*])^{\cdot}$$

This formula is a conjunction of the back-substituted value of ϕ and the back-substituted value of condition *c* which must be satisfied to flow along the specified path.

2.2.2.1. Simplification

It is instructive to analyze the form of the equations derived from this recursion. Alternation of paths introduces disjunctions in the formula and concatenation introduces conjunctions. Thus the general form of the resulting formulas is a nesting of conjunctions and disjunctions. Consider the control flow subgraph:



Where c is the conditional expression controlling the two-way branch. For this graph

$$p(s,e,(s \circ t \circ e)|(s \circ f \circ e),\varphi) =$$
$$(c[\sigma_s]\&\varphi[\sigma_e][\sigma_t][\sigma_s]) \vee$$
$$(\neg c[\sigma_s]\&\varphi[\sigma_e][\sigma_f][\sigma_s])$$

Note that "back substituting" the formula ϕ through an `if-then-else`, as represented by the diamond shaped CFG, generates a formula which is a least twice the size of ϕ (since there are two substitution instances of ϕ). In a program which is the sequential composition of diamond-shaped subgraphs, a formula back substituted to the beginning of the program will be exponential in the number of such diamonds, i.e. the number of control flow merge points. Basically, formula size grows in proportion to the number of potential paths through the program.

This behavior, if not addressed, prevents scalability of our analysis tools to large programs. Our defense against this growth, which is unavoidable in the worst case, is aggressive simplification of the formulas as they are constructed. The theorem prover provides this simplification, primarily through conditional and unconditional term rewriting. An intuition as to why such simplification prevents formulas from growing too large is that *P* may be thought of as converting an imperative-style program into a functional one. Experience has shown that functional programs *need not* be much larger then equivalent imperative ones.

A fundamental simplification rule is $(c\&\varphi) \vee (\neg c\&\varphi) \equiv \varphi$. This rule can be applied in the formula above if $\varphi[\sigma_t] = \varphi[\sigma_f]$. This will happen most simply if the assignments in nodes t and f do not modify variables of ϕ. In our experience, as program size grows, the number of assignments to any given variable and the number of variables a given expression depends on remains bounded. These observations contribute to frequent application of the fundamental simplification rule.

Simplification of generated formulas relies on other rules and techniques as well. To pursue this discussion further, we describe the organization and functionality of our prover. We then continue with the discussion of general simplification and testing for satisfiability.

2.3. Theorem Prover

In this work we are leveraging off the capabilities of our theorem prover, KITP. This prover is used in Kestrel Institute's work in program transformation [Smith90], and program verification [Wang92]. KITP maintains a knowledge base (KB) which holds axioms, inference rules, and deduced results, which are utilized by the main prover.

The main prover of KITP consists of a natural deduction controller (NDC) [Bledsoe83], a simplifier, a forward inference procedure (FIP), and a backward inference procedure (BIP). FIP is based on unit resolution, paramodulation, and set of support strategy [Wos68].

The BIP is the most powerful part of the prover, and is based on a goal-oriented proving procedure, hierarchical deduction [Wang87]. The BIP proves a theorem by traversing a tree of nodes. Each node contains a different set of rule clauses. All candidate goal clauses are contained in a goal-list. Each literal of a goal is indexed by a node name, through which a set of nodes can be located to obtain rule clauses for the resolution and paramodulation upon that literal. The "legal" resolvents of BIP are produced under a set of constraints or narrowing strategies, such as local subsumption, constraints on common tails, proper reduction, global subsumption, subgoal reordering, partial set of support, semantic guidance, etc.

The simplifier is designed to replace a term or subterm by a simpler, but semantically equivalent term or subterm. The simplification is carried out by term-rewriting and partial evaluation. Term-rewriting is the main component of the simplifier; its behavior is determined by the set of term-rewriting rules, which can be augmented by the user. Partial evaluation is used to deduce canonical forms for those terms or subterms that are computable. Simplification is applied exhaustively to each result of FIP and BIP as part of the symbolic evaluation procedure as it constructs formulas.

The natural deduction controller is the top-level control structure of the prover. It decomposes a conjecture into sub-formulas, and dispatches them to subcomponents.

To adopt the prover to feasible path determination we made several enhancements and modifications to KITP.

- The term rewriting system was extended to support conditional term rewriting rules. This was required for effective treatment of arrays, a traditional difficulty for symbolic evaluation.
- We scaled up the knowledge base to hold very large formulas.
- We improved KITP's capability for propositional reasoning.
- We adapted the backward inference procedure so that it does not require a formula in conjunctive normal form. Normalization of formulas can cause an exponential increase in their size.
- In order to promote effective use of term-rewriting and automatic operation, we transformed as many rules as possible into term-rewriting rules.
- We modified the natural deduction controller so that the prover is focused on disproving a non-theorem, instead of, as is usual, proving a theorem. This strategy is useful because in most cases, the conjecture given to KITP is a non-theorem, and a standard theorem prover would waste time before reporting failure.
- We tuned the control parameters (number of generated clauses, depth of search, function nesting limit, etc.) of the prover to run effectively and automatically with constrained resources.

The general inference process of the prover may be outlined by the following. Given a conjecture to be proved, it is first simplified. If the resulting formula is *true* then a proof has been found. Otherwise the formula is negated and transformed into a negation normal form (i.e. an and-or tree), and then given to the NDC. The NDC will try to find a connective path [Andrews81, Bibel81, Murray87] (a set of literals contained by an and-branch of the and-or tree) that is consistent or can not be proved to be inconsistent. Once such a path is found, the inference process is terminated and the prover reports that the input conjecture is a non-theorem. If all connective paths have been traversed and proved to be inconsistent, then the prover reports that the input conjecture is a theorem.

3. Applications: Structural Testing of Software

Structural (white box) testing is a widely-used testing methodology in which test cases are constructed from code without reference to requirements, specification, or other description of intended functionality. A collection of test cases, called a *test suite*, is constructed which satisfies one or more coverage criteria. A coverage criterion is a parameterized family of coverage instances defined in terms of some structural property of the code. For example, the coverage criterion "statement coverage" has one parameter instance for each statement of the program which requires that the test set contain a test case in which flow passes to the statement. More formally, a coverage instance is a path regular expression denoting a set of flows through a program. A test case satisfies a coverage instance if the control flow path traversed in execution of the test case is in the class of paths denoted by the regular expression. A test suite satisfies a coverage criterion if for each coverage instance of the coverage criterion there is a test case that satisfies it.

Many coverage criteria have been defined. We note several with respect to consideration of a path regular expression P and formula φ, as outlined above:

- **Statement Coverage** Each statement of the program should be exercised. For each statement instance S , P_S will denote the set of all paths passing through S, and $\varphi \equiv true$. Statement coverage is a very minimal criteria.
- **Decision Coverage** The outcome of every decision (e.g. `if` alternative, or `case` alternative) should be exercised to be both true and false. Each decision D will have an associated P_D denoting the set of all paths passing through D, and a pair φ_t and φ_f, each of which must be made true.
- **Condition Coverage** Besides testing the outcome of every decision, the various ways in which each outcome can be reached through different values of the subconditions within the decision must be exercised. This is a refinement of decision coverage which will introduce a set of φ_i over the individual conditions, according to the particular structure of the decision.
- **All Paths** All control flow paths through a program are tested. Thus each P_i is a family of precisely one path. This is a very stringent coverage criteria which becomes infeasible for large programs since the size of the test suite must grow exponentially in the size of the target procedure.
- **DU-Pair Coverage** This example of data-flow coverage was discussed in the introduction. For each variable v and pair of its definition d and use u, $P_{d/u}$ will denote all definition-free paths between d and u. As with the all paths criteria, the associated $\varphi_{d/u}$ will simply be *true*.

The relevance of feasible path analysis to testing is based on the following observations:

- Though a single test case can exercise only one path, it can in general satisfy many coverage instances. For example a single test case will exercise more than one statement and (normally) more than one du-pair. Moreover, two distinct test cases may exercise a non-empty intersection of coverage instances. Feasibility analysis can guide the judicious selection of test cases to achieve a “minimal” size test suite satisfying a given coverage criterion.
- It is not obvious when a test set achieves coverage since some of the coverage instances defined by a coverage criterion may not be feasible. For example there may be statements in a program which are not reachable in the sense that there are no input values that cause execution

to reach the statement. For data flow criteria this is a significant issue.

In an initial use of FPA we used it to generate an enumeration, by depth-first search, of all feasible paths. The algorithm tests the consistency of partial paths starting at entry to the program. If a partial path is inconsistent, so are all of its extensions. With a complete enumeration of all feasible paths, detection of infeasible testing requirement and test suite size optimization become tractable problems for moderately sized programs.

However, this solution does not scale since all feasible paths must be enumerated, and the number of feasible paths grows exponentially in program size. Nonetheless, because of the overall efficiency of the enumeration procedure, programs with over 200K feasible paths were analyzed. This corresponds to procedures of moderate size— roughly 20 sequential `if` statements (with approximately 50% of the structural paths feasible). It could be argued that writing procedures with more decisions than that is bad engineering practice, so this represents an adequate solution.

Here is a typical result: To determine all of the feasible paths in the program shown in Appendix A, the prover was passed 337 conjectures by the symbolic analyzer. Among these, 95 were proved to be theorems. The entire analysis took about 9 minutes, 8.5 of which were used by the prover. On average, the prover used about 2 seconds checking an individual conjecture.

This example demonstrates the importance of our use of a natural deduction controller for KITP and our strategy of focusing on disproving that the input conjecture is a theorem. Since NDC tries to find a model for the negation of the input conjecture, a non-theorem conjecture may be detected before the entire formula has been processed and checked.

Appendix B shows one of the 337 conjectures passed to the prover. This conjecture is interpreted as asserting that if the path defined by the underlined statements is executed, then the `if` statement at point `E` must take the `then` branch. The formula is an implication composed of a hypothesis which is a conjunction of the conditions that must be satisfied if the underlined path is taken. The conjunctions on the first three lines correspond to the condition at point A. The conditions corresponding to the `if` statements at points `B`, `C` and `D` follow and are delineated by blank lines. The condition at point `E` is the conclusion of the implication. All of these conditions are expressed by back substitution in terms of the initial values of the variables on entry to the procedure.

By means of NDC, KITP determined that it is a non-theorem in 0.05 second. However, for the same conjecture, if the prover attempts to prove it to be a theorem then the negated conjecture must first be normalized into a set of 739 clauses. The attempted proof of this conjecture takes over five minutes. The result of this example analysis is that out of a possible 1018 paths through the program, 242 were shown to be feasible.

In fact the inference system has no difficulty testing individual path feasibility for programs with over 100 sequential `if` statements. (Sequential `if` statements is an appropriate complexity measure since each condition is incorporated into the formula tested for satisfiability.) Thus the prover could scale to much larger problems. The intractability of this approach on large problems is not due to the theorem prover itself, but the enumeration of individual paths.

In collaboration with the group at Boeing [Jasper94], we are now working to apply FPA to structural testing using the following approach: Each test requirement corresponds to a path regular expression. Satisfiability of the formula associated with the PRE demonstrates feasibility of the corresponding test. To obtain a minimal size test suite we use a greedy algorithm which partitions the formulas corresponding to feasible tests into maximal satisfiable subsets.

The fundamental correspondence at work is that the conjunction of two formulas, each representing a set of test requirements, produces a formula which, if satisfiable, will meet both sets of requirements. Thus the partitioning is done through incrementally conjoining the formulas and testing for satisfiability.

4. Related Work

The significance of feasible path analysis to structural testing is described in [Clarke88, Frankl88, Woodward80, Korel90, Weyuker90]. For example Weyuker [Weyuker90] writes:

> Even though the size of the required test sets were not nearly as large as predicted by the theoretical upper bounds, we did encounter one practical difficulty when using the data

> flow criteria which has negative implications to the use of these criteria for large programs. The problem was determining which of the definition/use associations or du-paths were executable [feasible]. This problem is encountered when using many program-based criteria, including statement and branch coverage, but is particularly acute for all-du-paths criterion since there are frequently a large number of unexecutable du-paths. In fact, we found that the unexecutable path problem, not the large number of required test cases, was the primary practical difficulty in using the all du-paths criterion.

Clarke et. al. in [Clarke88] state "For example, data flow analysis tends to produce too many uninteresting anomalies unless it is integrated with a tool to evaluate path feasibility and subsequently remove unexecutable anomalies."

Despite this, to our knowledge, there are few recent attempts to solve the infeasibility problem. On the other hand there were a number of attempts in the 1970's. Typical of this work is [Bicevskis79, Boyer75, Howden77, Clarke88, Ramamoorthy76, Clarke76]. The sophistication of symbolic evaluators and the lack of significant computational resource at that time precludes any practical results from such systems.

Field [Field93] has described a formal system based on an equational axiomatization with a confluent set of rewrite tools to capture reasoning about imperative programs. In particular, his system has a nice treatment of pointers, somewhat in the style of our formalization of array indexing.

Werner and Howden describe limited methods that detect infeasible paths in COBOL programs [Werner91].

Laski's STAD system [Laski 90] performs the symbolic evaluation but provides no tools for testing feasibility. Indeed, without a simplifier our experience is that these expressions become unmanageable. Having determined that a testing requirement is infeasible, he states rules which propagate infeasibility assertions to other test requirements. In [Yates89] a statistical path generation strategy which minimizes the number of infeasible paths is proposed.

Problems of infeasibility are not restricted to white-box testing. Tripathy and Sarikaya describe a test generation from LOTOS specifications which may also generate infeasible tests [Tripathy91].

An interesting alternative to symbolic evaluation is presented in [Korel90].

5. Future Work

5.1. Extensions to the Inference System

While the overall effectiveness of the prover in this application has been a key enabling technology, further improvements in its performance will enable faster, more accurate results, inter-procedural feasibility analysis, specification-based testing and enhanced defect analysis. While tuning and implementation of minor improvements based on experimental results is a constant activity, we believe that incorporation of a special purpose decision procedure for integer linear programming (ILP) will yield significant performance improvements. This belief is based on two observations.

1. New methods for solving ILP problems have obtained tremendous performance. These methods employ linear programming techniques, but search the feasible space for lattice point (i.e. integer valued) solutions. While ILP is NP-Complete these algorithms have been quite practical over the problem sizes we anticipate.
2. Because most operations on integer, enumeration, record, and array types are generally represented by linear arithmetic formulas in the prover, there is a significant opportunity to apply these methods.

A notable application of ILP methods in the context of program analysis is array dependency analysis [Pugh92, Goff91] as used in parallelizing compilers.

5.2. Extensions to the Symbolic Evaluator

We also wish to implement a more comprehensive and accurate analysis of loops. The traditional means for establishing such relationships is via a loop invariant, but our goal is a completely automated system which does not require the user to provide loop invariants. Instead we plan to "lift" loops so that each loop is replaced by assignment statements assigning values to the variables modified in the loop with values expressed using high-level functional forms such as accumulations, reductions, filters, and maps [Letovsky88, Wills87]. Lifting loops into a sequence of assignments written over high-order functionals allows us to back formulas through loops in the same way as we would back a formula through straight-line code.

Pointers are not treated by the system, but their implementation is generally straightforward. For user defined (abstract) data types we could construct axioms for their behavior or directly unfold the implementation bodies.

5.3. Other Applications of Feasible Path Analysis

Program dependence graphs [Horwitz88] have been used to support diverse applications such as program slicing, program optimization, and program understanding. Integrating feasible path analysis into the construction of program dependence graphs will leverage the sharper analysis our techniques provide into a broad range of applications.

There are a number of potential applications associated with finding program defects, such as zero divides, nil pointer references, array bounds violations, and so forth. The basic capability of querying whether an arbitrary formula at a given program point reached by some collection of control flow paths can (or must be) true is a quite general and powerful analysis paradigm.

Acknowledgments: Part of this work has been supported by NASA and Boeing Corporation.

6. References

Andrews81 Andrews, P. B. Theorem Proving via general mating. *J. ACM* 28 (2), 1981, 193-214.

Bibel81 Bibel, W. On matrices with connections. *J. ACM* 28(4) (1981) 633-645.

Bicevskis79 Bicevskis, J., Borzovs, J., Straujums, U., Zarins, A., and Miller, E. SMOTL -- a system to construct samples for data processing program debugging. *IEEE Transactions on Software Engineering SE-5, 8* (August 1990), 60--66.

Bledsoe83 Bledsoe, W. W. The UT interactive prover. Tech. Report ATP-17B, Department of Mathematics, The University of Texas at Austin (1983).

Boyer75 Boyer, R., Elspas, B., and Levitt, K. SELECT -- A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices 10, 6,* June 1975, 234--245.

Boyer86 Boyer, R. S. and Moore, J. S. Integrating decision procedures into heuristic theorem provers: a case study with linear arithmetic. In, *Machine Intelligence 11*, Oxford University Press, 1986.

Clarke76 Clarke, L.A. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering SE-2, 3* (September 1976), 215--222.

Clarke88 Clarke, L. A., Richardson, D.J., and Zeil, S.J. TEAM: A support environment for testing, evaluation, and analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* Boston, MA, November 28--30, 1988, ACM, pp. 153--162.

Field93 Field, J., A simple rewriting semantics for realistic imperative programs and its application to program analysis, Revision of paper appearing in the 1992 Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Francisco.

Frankl88 Frankl, P.G., and Weyuker, E.J. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering SE-14, 10*, October 1988, 1483--1498.

Goff91 Goff, G., Kennedy, K. and Tseng, C., Practical dependence testing. In, *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.* ACM. Toronto, Canada, June 26--28, 1991, 15-29.

Hoare69 Hoare, C. An axiomatic basis for computer programming, *CACM,* 12,10 (1969).

Horwitz88 Horwitz, S., Prins, J., and Reps. T. On the adequacy of program dependence graphs for representing programs. In, *Fifteenth ACM Symposium on Principles of Programming Languages.* ACM San Diego, CA, January 13--15, 1988, 146--157.

Howden77 Howden, W.E. Symbolic testing and the DISSECT symbolic evaluation techniques. *IEEE Transactions on Software Engineering SE-4, 4* (1977), 266--278.

Jasper94 Jasper, R., Brennan, M., Williamson, K., Currier, C., and Zimmerman, D. Test data generation and feasible path analysis. *International Symposium on Software Testing and Analysis* (Seattle, WA, August 17--19, 1994).

Korel90 Korel, B. Automated software test data generation. *IEEE Transactions on Software Engineering 16, SE-8* August 1990, 870--879.

Kotik89 Kotik, G.B., and Markosian, L.Z. Automating software analysis and testing using a program transformation system. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification* (Key West, Fl, December 13--15, 1989), ACM, 48--54.

Laski90 Laski, Janusz, Data flow testing in STAD, *J. Systems and Software*, vol. 12 (1990), 3-14.

Letovsky88 Letovsky, S. I. Plan Analysis of Programs. Ph.D. Thesis, Yale University, Department of Computer Science December 1988.

Murray87 Murray, N.V., and Rosenthal, E. Inference with path resolution and semantic graphs, *Journal of the ACM 34,2,* (April, 1987), 225--254

Pugh92 Pugh, W. A practical algorithm for exact array dependence analysis. *Communications of the ACM 35, 8*, August 1992, 102--115.

Ramamoorthy76 Ramamoorthy, C., Ho, S., and Chen, W. On the automated generation of program test data. *IEEE Transactions on Software Engineering SE-2, 4* (December 1976), 293--300.

Reasoning90 *Refine^{TM} 3.0 User's Guide*, Reasoning Systems Incorporated, Palo Alto, CA. May 25 1990.

Reasoning92 *Refine/Ada User's Guide*, Reasoning Systems Incorporated, Palo Alto, CA. July 26, 1992.

Shostak79 Shostak, R. E. A practical decision procedure for arithmetic with functional symbols. *Journal of the ACM 26, 2*, April 1979, 351--360.

Shostak81 Shostak, R. E. Deciding linear inequalities by computing loop residues. *Communications of the ACM 28, 4*, October 1981, 769--779.

Shostak84 Shostak, R. E. Deciding combinations of theories. *Journal of the ACM 31, 1*, January 1984, 1--12.

Smith90 Smith, D.R. KIDS -- a semi-automatic program development system.*IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16, 9* (September 1990), 1024--1043.

Tripathy91 Tripathy, P., and Sarikaya, B. Test generation from LOTOS specifications, *IEEE Trans. on Computers*, 40, 4 (April, 1991), 543-561.

Wang85 Wang, T.C. Designing examples for semantically guided hierarchical deduction. In *9th International Joint Conference on Artificial Intelligence* (Los Angeles, California, August 18-23, 1985).

Wang87 Wang, T.C., and Bledsoe, W.W. Hierarchical deduction. *Journal of*

Automated Reasoning 3, 1 (March 1987), 35--77.

Wang92 Wang, T.C., and Goldberg, A. RVF: an automated formal verification system.In Eleventh Conference on Automated Deduction, D.Kapur, Ed. Springer-Verlag, Berlin, 1992, pp.131--138.Lecture Notes in Computer Science, Vol. 607.

Werner91 Werner, L. and W.E. Howden, An investigation of the applicability of data usage analysis. *J. Systems and Software,* Vol 15 (1991), 205-215.

Weyuker88 Weyuker, E.J. An empirical study of the complexity of data flow testing. In *Proceedings of the Second Workshop on Software Testing, Analysis, and Verification*, Banff, Canada, July 19--21, 1988, IEEE Computer Society, pp. 188--195.

Weyuker90 The cost of data flow testing: an empirical study, *IEEE Trans. on Software Engineering,* 16,2 (1990), 121-128.

Wills87 Wills, L. M. Automated program recognition. Tech. Rep. MIT-AI-904, MIT AI Laboratory, February 1987.

Woodward80 Woodward, M.R., Hedley, D., and Hennell, M.A. Experience with path analysis and testing. In Tutorial: Software Testing & Validation Techniques, 2Ed, E.Miller and W. Howden, Eds. IEEE Computer Society Press, Los Alamitos, CA, 1981, pp.194--206.

Wos68 Wos, L., and Robinson, A, Paramodulation and set of support. Proceedings of the IRIA Symposium on Automatic Demonstration, Versailles, France, France, Spring-Verlag (1968) 276-310.

Yates89 Yates, D.F., and Malevris, N. Reducing the effects of infeasible paths in branch testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium onSoftware Testing, Analysis, and Verification* (Key West, Fl, December 13--15, 1989), ACM, 48--54.

Appendix A

```
      -- All variables are externally declared and of type Boolean


      procedure test is
      begin
       dc_1 := cc_1 or ic_1;
       dc_2 := cc_2 and dc_1 and ic_2;
       dc_3 := (ic_3 or cc_3 or cc_1) and (ic_4 or not (ic_5 and cc_2
                 and not cc_3));
       dc_4 := (dc_3 or not cc_4);
       dc_5 := (dc_1 and cc_2) or cc_3 or not cc_4;
       dc_6 := ic_6 and cc_5;
       dc_7 := ic_7 and cc_5;
       dc_8 := ic_8 xor dc_5;
       dc_9 := dc_3 or ic_9;
       dc_10 := (not (dc_4 and dc_2) or ic_10) and dc_9;
       dc_11 := true;
A       if not (not dc_7 or not dc_2 or ic_11 or cc_6) then
B          if ic_12 then
             dc_11 := ic_13;
          end if;
       end if;
       dc_12 := dc_11 or not ic_14;
C       if not (dc_2 and cc_3) then
          null;
D       elsif not dc_12 then
          null;
       end if;
       dc_13 := dc_6 and (cc_6 or not (dc_2 and dc_12));
       dc_14 := dc_13 and dc_8 and dc_10;
E      if dc_14 then
          null;
       end if;
       if not dc_14 then
          if ic_15 then
             null;
          end if;
       end if;
     end test;
```

Appendix B

```
(CC-5 & IC-7)
& (IC-2 & ((IC-1 or CC-1) & CC-2))
& ~IC-11 & ~CC-6

& IC-12

& (IC-2 & ((IC-1 or CC-1) & CC-2))
& CC-3

& ~(~IC-14 or IC-13)

=>

   ((IC-9
       or (~(~CC-3 & (CC-2 & IC-5)) or IC-4)
            & (CC-1 or (CC-3 or IC-3)))
      & (IC-10
           or ~((IC-2 & ((IC-1 or CC-1) & CC-2))
                  & (~CC-4
                       or (~(~CC-3 & (CC-2 & IC-5))
                             or IC-4)
                            & (CC-1 or (CC-3 or IC-3))))))
     & (((~CC-4
            or (CC-3 or CC-2 & (IC-1 or CC-1)))
           & ~IC-8
           or ~(~CC-4
                  or (CC-3 or CC-2 & (IC-1 or CC-1)))
                & IC-8)
          & ((~((~IC-14 or IC-13)
                  & (IC-2 & ((IC-1 or CC-1) & CC-2)))
                or CC-6)
               & (CC-5 & IC-6)))
```