Test Data Generation and Feasible Path Analysis

Robert Jasper Mike Brennan Keith Williamson Bill Currier
The Boeing Company

David Zimmerman
Z-Access Consulting

ABSTRACT

This paper describes techniques used by Test Specification and Determination Tool (TSDT), an experimental prototype for analysis and testing of critical applications written in Ada. Two problems dominate structural testing of programs: exponential explosion in the number of execution paths and feasible path determination. A path is feasible if there exists some input that will cause the path to be traversed during execution.

We present techniques based on new representations combined with automated theorem proving to deal with these problems. The paper describes how these techniques can be used to determine the feasibility of expressions containing references to Ada arrays. Finally, we present algorithms specific to generating test data under the modified condition decision coverage (MCDC) criterion.

While we realize that many of the problems we are attempting to solve are, in general, undecidable, we are encouraged by preliminary results obtained from running TSDT against vendor supplied code. Based on our results, we feel these techniques can be applied to a broad enough section of Ada code to make them cost effective.

1 INTRODUCTION

Testing consumes at least half the labor of developing software intensive systems [3]. These costs are especially evident in embedded avionics systems where testing is both difficult and costly to perform.

ISSTA 94 - 8/94 Seattle Washington USA

Part of the difficulty in testing these systems lies in their inherent complexity and dynamic real-time nature. In addition, regulatory authorities have imposed specific structural coverage requirements.

The avionics software development guideline, DO-178B [19], allows the satisfaction of structural test coverage requirements during functional integration or system testing. Typically, some percentage of the structural coverage requirements are not easily satisfied this way and it becomes cost effective to resort to module-level testing.

The most critical avionics software must be tested using a structural coverage criterion called modified condition decision coverage (MCDC) [5, 19]. MCDC is difficult to apply manually, even by experienced testers.

TSDT attempts to address the difficulty of applying MCDC to a limited subset of Ada. TSDT analyzes a program and execution trace data and determines if MCDC has been achieved. TSDT can generate test cases for those uncovered areas in the code. In fact, TSDT has generated complete MCDC test suites from previously untested vendor supplied code that are smaller than those generated manually.

2 TERMINOLOGY & BACKGROUND

2.1 TERMINOLOGY

The *input domain* of a program is the set of all possible inputs restricted by finite word size and memory of a particular implementation. A *test input specification* is a formula in first order logic representing restrictions on the input domain intended to force specific boolean expressions in the code to take on particular values.

A *test input* is an element of the input domain that satisfies a test input specification. A *test suite specification* is a set of test input specifications that satisfies some coverage criterion. A *test suite* is a set of test inputs that collectively satisfy a test suite specification.

A *decision* is a boolean expression that determines the flow of control in a program such as an IF or a CASE statement. A decision contains zero or more boolean connectives (i.e., and, or, not). A *condition* is a leaf-level boolean expression of a boolean expression (i.e., it contains no boolean connectives). A decision can be made up of a single condition.

Modified condition decision coverage (MCDC) is defined in [19] as:

> Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decisions outcome by varying just that condition while holding fixed all other possible conditions.

2.2 BACKGROUND

Research has highlighted the significance of feasible path analysis to the area of structural testing [8, 9, 30, 31]. Early research systems attempted to determine feasibility using a combination of symbolic evaluation, linear solvers, and ruled-based checks [2, 7, 8, 14, 18, 20]. These techniques, while providing a basis for testing research, were limited with respect to the complexity and types of the programs they were able to handle. TSDT's goal is to reduce language and complexity limitations through the use of new representations and automated theorem proving.

A previous prototype, Valid Conditions Table Generator (VCTG) [6,15], focused on the feasibility of individual paths. VCTG produced a table of all feasible paths which was intersected with a table created for a particular coverage criterion. This produced a smaller table, which provided guidance in selection of the smallest test set satisfying a particular coverage criterion.

This approach proved to be limited because the number of potential paths in a program (without loops) is exponential in the number of control flow merge points. Thus, even if the analysis of each path can be done very quickly, and infeasible path continuations are pruned, simply enumerating the different paths and constructing path-wise coverage tables quickly becomes intractable for moderate size programs [11]. Prior research in feasible path analysis also tended to focus on individual paths.

TSDT's strategy is to formally characterize and reason about collections of paths (path families) rather than individual paths. A path family can be specified by a logical formula over a program's input domain. The elements of such formulae are assertions constructed from the program's decisions and conditions, with the appropriate substitutions and expansions required to express them in terms of program inputs. By not being concerned about just which path is taken, we expect to gain tractability in reasoning about program behavior and generating test inputs.

Any test input which satisfies such a formula will cause some path in the collection of paths to be executed. By using a theorem prover, TSDT is able to reason about formulae containing constructs that are not easily handled using other methods (e.g., arrays); [13] presents more detail. Early research systems often avoided troublesome constructs. While TSDT is limited by resource constraints imposed by the prover, application of automated theorem proving in this area appears to be a promising area of research.

2.3 LIMITATIONS

TSDT has several limitations. Test Generation supports stand-alone unit test with stubs and drivers. Path analysis is intra-subprogram and assumes sequential execution without interruption. Currently, TSDT only analyzes statically bound loops, which are unrolled. TSDT generates inputs for integer, integer subrange, enumerated, and boolean types; TSDT does not yet handle paths controlled by floating point, fixed point, or access types.

3 FUNCTIONAL OVERVIEW

Figure 1 presents a functional and architectural overview of TSDT. TSDT performs the following functions on Ada subprograms:

- Test Generation—generate a test suite specification and inputs that achieve MCDC.
- Test Coverage Determination—determine whether MCDC has been achieved based on trace data.

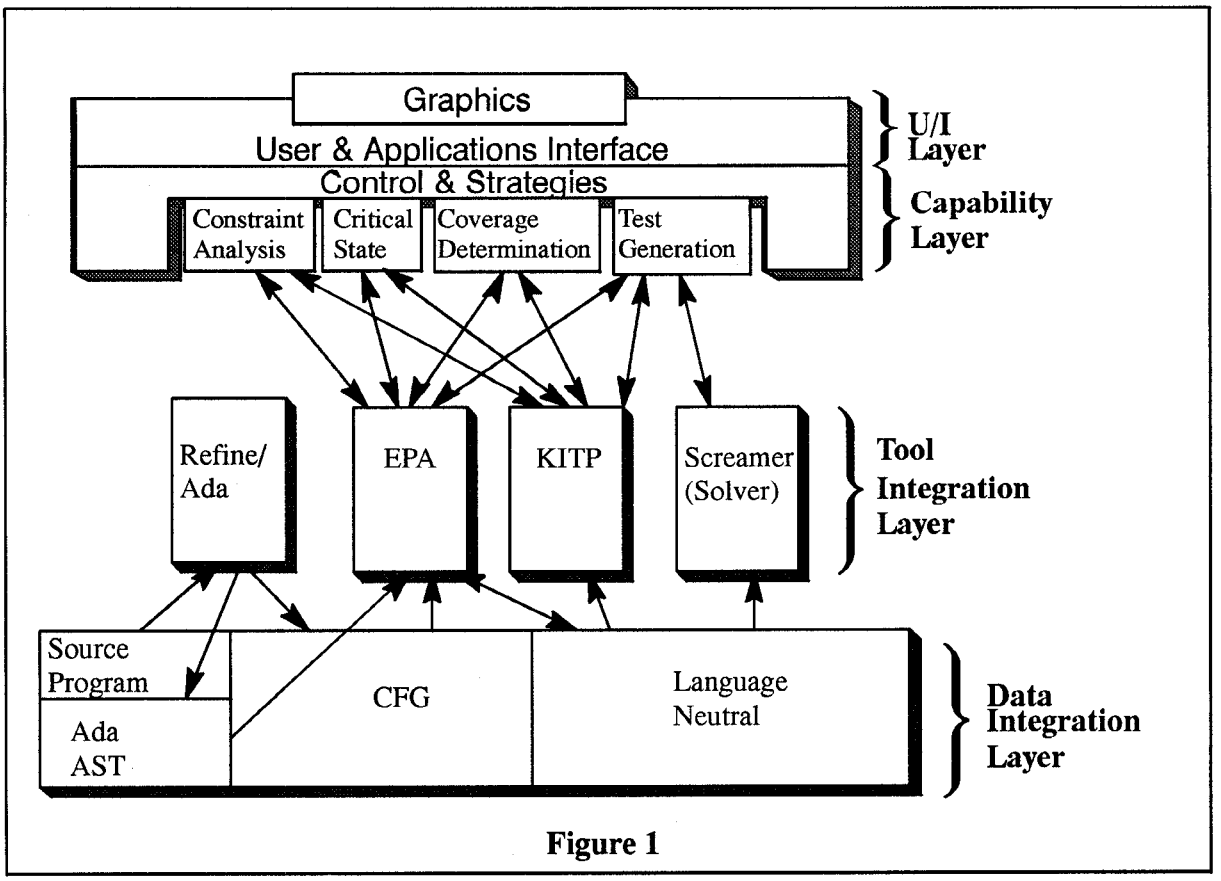


Figure 1

- Path Constraint Analysis—determine the weakest precondition that will force execution through a specific path in the code or through some path between two specific nodes.
- Critical State Analysis—determine whether an expression can ever be true at a specific program point.

4 ARCHITECTURAL OVERVIEW

TSDT was built using Software Refinery and Refine/Ada[6, 21, 22]. Refinery is an environment that has been used to build software analysis, reverse engineering and software synthesis tools. Refinery includes a high-level programming language that supports set theory, a subset of first order logic, and a LALR parsing system.

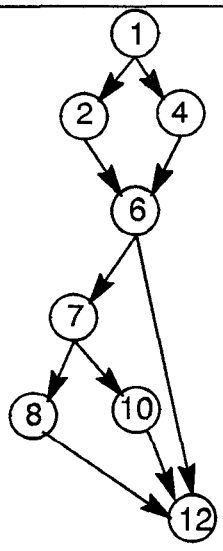
As shown in Figure 1, TSDT incorporates the following tools:

- Refine/Ada—a commercial product that provides an Ada parser, linker, and various tools including a control flow graph (CFG) generator.
- Edge Predicate Analyzer (EPA)—a general purpose engine for representing program expressions in terms of variables at some prior point (usually entry). This is accomplished through a combination of symbolic analysis, back substitution, and term-rewriting; details can be found in [10, 12, 16] and companion paper [13].
- Kestrel Institute Theorem Prover (KITP)—an automated resolution-based prover for first order logic[17, 28, 29].
- Screamer (Solver)—an extension of Common Lisp that includes a constraint satisfaction subsystem[25, 26, 27].

The four major domains of the data integration layer are:

- Source Program—a library or file containing Ada source code.
- Ada Abstract Syntax Tree (AST)—an annotated tree representation of the source program produced by the Refine/Ada parser and linker.
- Control Flow Graph (CFG)—a directed graph consisting of nodes and edges. Each node represents a basic block (i.e., a sequence of Ada statements that must be executed linearly, without change in control); edges represent flow of control.

```
1: if set_point > 100 then
2:   n_count := 0;
3: else
4:   n_count := n_count - elapsed;
5: end if;
6: if elapsed > 0 then
7:   if (n_count >= 0) and (e_count >= 0) then
8:     do_something();
9:   else
10:    do_something_else();
11:  end if;
12:end if;
```



Reach(7) ≡ elapsed > 0

C(n_count >= 0) ≡ set_point > 100 ∨ n_count - elapsed >= 0

C(e_count >= 0) ≡ e_count >= 0

Figure 2

- Language Neutral—an internal form for expressions in first order logic using Refine ASTs [23]. These expressions are processed by EPA, KITP, and Screamer.

5 REPRESENTATIONS AND FEASIBILITY

This section describes TSDT's test specification representation, how this representation supports our proof strategy, and how it can be used to represent Ada arrays. In addition, this representation attempts to minimize the cost of generating formulae corresponding to many different test specifications related to a specific decision.

5.1 TEST SPEC REPRESENTATION

Let $\mathcal{E}$ represent a boolean expression in CFG node $\mathcal{N}$. With respect to the MCDC criterion, $\mathcal{E}_j$ represents the j^{th} condition within $\mathcal{E}$; n represents the number of conditions within $\mathcal{E}$. *Reach*($\mathcal{N}$) represents the weakest precondition (i.e., constraints on inputs) under which control reaches node $\mathcal{N}$; $C(\mathcal{E}_j)$ represents the weakest preconditions under which $\mathcal{E}_j$ holds, assuming control reaches node $\mathcal{N}$. In other words, $C(\mathcal{E}_j)$ is constructed under the assumption *Reach*($\mathcal{N}$). An example construction of $C(\mathcal{E}_j)$ and *Reach*($\mathcal{N}$) is shown in Figure 2.

Under MCDC, we are required to form input specifications involving $\neg\,\mathcal{E}_j$. The negation of a specific condition $\mathcal{E}_j$ can be formed by negating $C(\mathcal{E}_j)$:

$$C(\neg\,\mathcal{E}_j) \equiv \neg\, C(\mathcal{E}_j)$$

Selection of which $\mathcal{E}_j$ conditions to negate for any boolean expression is described in section 6.2. The choice of which conditions to negate is represented using boolean n-vector $\mathcal{V}$ where v_j represents the j^{th} element of $\mathcal{V}$. Test input specifications are generated using the following formula:

$$Reach(\mathcal{N}) \wedge \bigwedge_{j=1}^{n} B(\mathcal{E}_j, v_j)$$

where,

$$B(\mathcal{E}_j, v_j) = \begin{cases} C(\mathcal{E}_j) & \text{if } v_j = \text{t;} \\ \neg\, C(\mathcal{E}_j) & \text{if } v_j = \text{f.} \end{cases}$$

Using the example in Figure 2, the test input specification for the boolean expression on line 7 where $\mathcal{V}$=`tf` is:

elapsed > 0 ∧
(set_point > 100 ∨ n_count - elapsed >= 0) ∧
e_count < 0

For any boolean expression $\mathcal{E}$, 2^n possible test input specifications can be formed, where n is the number of conditions contained in $\mathcal{E}$. The time taken to generate a single specification can be exponential with respect to the number of merge nodes in the CFG. Our design seeks to minimize the time taken to generate formulae for arbitrary values of $\mathcal{V}$.

This is accomplished by separating *Reach*($\mathcal{N}$) from $C(\mathfrak{E}_j)$. A single call to EPA generates $C(\mathfrak{E}_j)$ for j in {1..n}. Test input specifications for arbitrary $\mathcal{V}$'s are formed using $B(\mathfrak{E}_j,v_j)$ (which may simply negate $C(\mathfrak{E}_j)$). This represents a significant time savings. After computing *Reach*($\mathcal{N}$) and $C(\mathfrak{E}_j)$ for j in {1..n}, the complexity of generating 2^n possible $B(\mathfrak{E}_j)$'s is reduced to an $O(n)$ problem. This separation also provides a means of focusing the prover.

5.2 DETERMINING FEASIBILITY

Test input specifications generated by TSDT are checked for feasibility using the Kestrel Institute Theorem Prover (KITP) [17]. KITP is a resolution-refutation theorem prover [4] that incorporates hierarchical deduction[28, 29] and mating strategies[1, 24]. Our project has developed axioms specifically for proving theorems about test input specifications generated by TSDT.

For TSDT, a test input specification is considered feasible if it can not be proven unsatisfiable (within the resource bounds of the tool) or a model can be found that satisfies the formula. TSDT considers a test input specification infeasible if its negation can be proved a theorem:

$$\vdash \neg (Reach(\mathcal{N}) \wedge \bigwedge_{j=1}^{n} B(\mathfrak{E}_j,v_j))$$

The negation of a test input specification will be referred to as an *infeasibility conjecture*, or simply the conjecture when the context is obvious.

KITP has two primary modes of operation: *Proof* and *Disproof*. In *Proof* mode, KITP attempts to prove the infeasibility conjecture is a theorem (i.e., the conjecture is valid in the current theory).

In *Disproof* mode, KITP focuses on proving the infeasibility conjecture is a non-theorem by looking for a counter example; if KITP cannot find a counter example, it attempts to prove the conjecture a theorem. Finding a counter-example to the conjecture corresponds to finding a set of inputs that satisfies the specification.

5.3 KITP PROOF STRATEGY

KITP employs three main procedures in Proof mode: simplifier procedure, Forward Inference Procedure(FIP), and Backward Inference Procedure(BIP). KITP uses axioms corresponding to each of these specific components during a proof. For our purposes, KITP performs best when given a conjecture of the form H→ ¬ G, where H is assumed to be consistent. KITP first calls the simplifier component on the conjecture. If the conjecture simplifies to true, a proof has been found. If not, the formula is negated and transformed into a set of clauses:

$$S: \{h_1, h_2, \ldots, h_n, g_1, g_2, \ldots, g_m\}$$

where $h_1, h_2, \ldots, h_n$ are called hypothesis clauses and $g_1, g_2, \ldots, g_n$ are called goal clauses. We will refer to the set of hypotheses and goal clauses as H and G respectively.

FIP is called with H to derive new consequents. BIP is then invoked on S and the consequences derived from FIP. BIP uses hierarchical deduction using G as set-of-support (SOS) [4, 32]. Because H is assumed to be consistent, the contradiction must involve G. SOS resolution is a resolution of two clauses that are not both from H.

Our strategy with TSDT is to check test input specifications for feasibility using a conjecture of the form:

$$Reach(\mathcal{N}) \rightarrow \neg \bigwedge_{j=1}^{n} B(\mathfrak{E}_j,v_j))$$

If the conjecture is a theorem, it is infeasible to both reach the node and satisfy the collection of condition outcomes represented by the conjunction of $B(\mathfrak{E}_j,v_j)$'s.

This strategy is based on the assumption that each *Reach*($\mathcal{N}$) is satisfiable. This appears to be a reasonable assumption for newly coded avionics systems; a program containing any *Reach*($\mathcal{N}$) that is unsatisfiable implies that node $\mathcal{N}$ is semantically dead. As a precaution, this assumption can be tested by attempting to prove the conjecture ¬*Reach*($\mathcal{N}$); if this conjecture is a theorem, node $\mathcal{N}$ can not be reached.

5.4 ARRAY REPRESENTATION

Representing test input specifications involving arrays poses special problems. Consider the following basic block involving references to the array A:

```
A[i] := x
A[j] := y
If A[k] = 4 then
```

It is difficult to determine the value of the boolean expression `A[k] = 4` in terms of entry to the basic block. Normally, assignment statements are used to back substitute expressions to determine their value at some prior point. In this example, references to `A` are ambiguous because the values of `i`, `j`, and `k` are unknown.

It's unclear if the assignment to `A[j]` destroyed (shadowed) the previous assignment to `A[i]`. Some mechanism to capture the history of assignments to A is required.

One approach for representing expressions involving arrays is to unfold the history of assignments into a disjunction of potential assignments. Assertions are made to assure each assignment to the array does not shadow a prior assignment or that no assignment has been made. This is reflected in the following formula which represents the value of `A[k] = 4` on entry to the basic block :

$$(j = k \wedge y = 4) \vee$$
$$(\neg i = j \wedge i = k \wedge x = 4) \vee$$
$$(\neg i = k \wedge \neg j = k \wedge A\_k = 4)$$

where the variable reference `A_k` represents the value of array `A` at element `k` prior to entry of the basic block.

This approach was not used due to difficulties generating the test input specifications and potential explosion in the number of terms in the expression.

Our current approach involves generating expressions for Ada array operations using the following functions and the constant `*null-array*`:

```
a-range-seg(i:index,j:index,
            v:value):seg
a-index-seg (i:index,v:value):seg
a-cons(s:seg,a:array):array
a-elem(a:array,i:index):value
```

The `a-range-seg` function represents assignment to a segment of an array, such as in the Ada assignment statement:

```
A[1..10] := 0;
```

Correspondingly, `a-index-seg` represents assignment to an individual element of an array. The history of assignments can be formed via recursive composition using the `a-cons` function. This function takes a segment and an array and returns a new array. Use of `a-cons` can be viewed as pushing new assignments onto a stack of previous assignments. The constant `*null-array*` represents the bottom of the stack.

The function `a-elem` represents access to elements within an `a-cons` expression, such as in the Ada expression:

```
If A[k] = 4
```

Using this representation, the expression for `A[k]=4` in terms of inputs to the basic block is:

```
a-elem(a-cons(a-index-seg(j,y),
       a-cons(a-index-seg(i,x),
       *null-array*)),k) = 4
```

Conditional rewrite rules within KITP are used to access elements of the array based on the array's assignment history. Conditional rewrite rules are universally quantified formulae of the form:

$$H \rightarrow L = R$$

where L can be replaced with R if hypothesis H holds. The following four conditional rewrite rules are used to access components of the array:

```
element-in-head
∀(i,j,v,a)(i=j →
  (a-elem(a-cons(
   a-index-seg(i,v),a)),j)
   = v)

element-in-tail
∀(i,j,v,a)(¬i=j) →
  (a-elem(a-cons(
  a-index-seg(i,v),a)),j)
  = a-elem(a,j))

element-in-head-seg
∀(i,j,k,v,a)(i<=k ∧ k<=j →
  (a-elem(a-cons(
   a-range-seg(i,j,v),a)),k)
   = v)
```

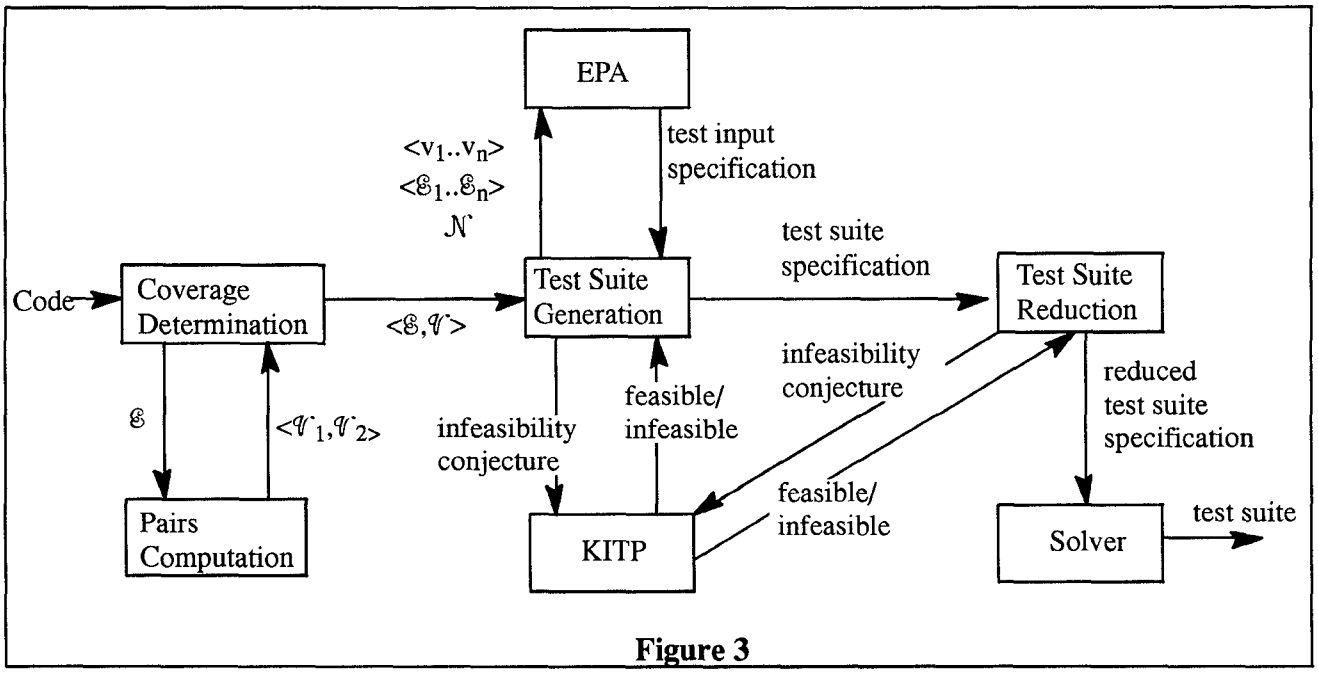


Figure 3

```
element-in-tail-seg
∀(i,j,k,v,a)(k<i ∨ k>j) →
  (a-elem(a-cons(
  a-range-seg(i,j,v),a)),k)
  = a-elem(a,k))
```

These rules provide semantics to the `a-elem` accessor function. If it can be shown that `i=j`, the element is at the head of the `a-cons` expression; if it can be shown that ¬ `i=j`, the element is definitely not in the head, look in the tail of the `a-cons` expression.

Consider the following Ada code fragment:

```
1:  A[x] := x
2:  A[y] := y
3:  if x > 2 and y < 2 then
4:    if A[y] >= 2 then
5:      do-something();
6:    end if;
7:  end if;
```

where the conjecture for

Reach(4) ∧ *B*(`A[y] >= 2,t`) is

```
(x > 2 ∧ y < 2)→
   ¬ (*a-elem(*a-cons(
       *a-index-seg(x,x),
       *a-cons(*a-index-seg(y,y),
       *null-array*)),y) >= 2)
```

Using standard axioms of arithmetic and the four axioms described above, it can be shown that this conjecture is a theorem. Therefore, it is not possible to reach statement 4 and force the condition to take on a true outcome.

6 TEST DATA GENERATION UNDER MCDC

Figure 3 shows how the test input generation components of TSDT interface with KITP and EPA. Tests under MCDC are naturally grouped into pairs. Section 6.1 defines MCDC pairs and the concept of condition coupling. Section 6.2 describes an algorithm for determining MCDC pairs that takes into account this coupling. Section 6.3 presents example output from TSDT and describes the algorithms for coverage determination and test suite reduction. Although we do not present our design for the solver component, TSDT interfaces directly with the constraint language portion of Screamer as described in [25, 26, 27].

6.1 MCDC PAIRS AND COUPLING

Under the MCDC criterion, each condition in a decision requires two tests called an MCDC pair—one with the condition true and one with it false and each resulting in a different decision outcome. If a decision has n uncoupled conditions (i.e., each condition can be toggled without changing the other values), then MCDC coverage can be satisfied with n+1 tests.

For example,

```
A or (B and C)
```

can be tested with

	T	F
A	ttf	ftf
B	ftt	fft
C	ftt	ftf

which requires four tests, `ttf, ftf, ftt and fft`. In the table, each row corresponds to a condition tested by an MCDC pair, the column labels indicate the value of the decision and a table element such as ttf indicates **A**=`t` and `B=t` and `C=f`.

Because of coupling, it is not always possible to toggle a single condition while holding all others constant. When a decision has coupled conditions, the MCDC criteria as defined by [19] can be interpreted in at least two ways. TSDT uses the following interpretation presented in [5],

> An ... interpretation is to view each condition as a distinct entity to be varied. but to find tests in which the effect of varying one instance of a variable affects the overall outcome while simultaneously masking the effects of all other instances of the variable.

Masking is best described by example. In the expression `(A and False)` there is no effect in varying `A`—the operand `False` *masks* `A`. Correspondingly, in the expression `(A or True)` `True` *masks* `A`. A more complicated expression that requires masking is:

`(A or B) and (A or C)` which can be tested with

	T	F
A	tftt	ffft
B	ftft	ffft
A	tttf	ftff
C	ftft	ftff

This can be tested with five tests. Note that in the pair for the second `A` (row 3), holding `B` true masks the effect of changing the first instance of `A`. Masking can be more subtle than shown in these examples. A precise definition of masking and the TSDT algorithms for MCDC pair computation are discussed in the next section.

6.2 MCDC PAIR COMPUTATION

In TSDT, MCDC pairs are determined using two functions defined on tuples $(\mathcal{E},\mathcal{V})$ where $\mathcal{E}$ is a boolean expression and $\mathcal{V}$ is a boolean n-vector and n is the number of conditions in $\mathcal{E}$. One function, $eval(\mathcal{E},\mathcal{V})$, evaluates $\mathcal{E}$ when the conditions of $\mathcal{E}$ are assigned the values of $\mathcal{V}$.
For example, if,

$$\mathcal{E} = \texttt{A and (B or C)}$$

and

$$\mathcal{V} = \texttt{tff},$$

then

$$eval(\mathcal{E},\mathcal{V}) = \texttt{(t and (f or f))} = \texttt{F}.$$

The other function, $I(\mathcal{E},\mathcal{V})$, computes the set of conditions of $\mathcal{E}$ that have *influence* on $eval(\mathcal{E},\mathcal{V})$.

$$I(\mathcal{E},\mathcal{V}) = \begin{cases} \{\mathcal{E}\} & \text{if } \mathcal{E} \text{ is a condition;} \\ I(\mathcal{F},\mathcal{V}) & \text{if } \mathcal{E} = \neg\,\mathcal{F}; \\ H(\mathcal{E},\mathcal{F},\mathcal{V}) \cup & \\ H(\mathcal{E},\mathcal{G},\mathcal{V}) & \text{if } \mathcal{E}=\mathcal{F}\ op\ \mathcal{G}. \quad (*) \end{cases}$$

where *op* is `and/or`, and

$$H(\mathcal{E},\mathcal{F},\mathcal{V}) = \begin{cases} I(\mathcal{F},\mathcal{V}') & \text{if } eval(\mathcal{E},\mathcal{V}) = eval(\mathcal{F},\mathcal{V}'); \\ \emptyset & \text{otherwise.} \end{cases}$$

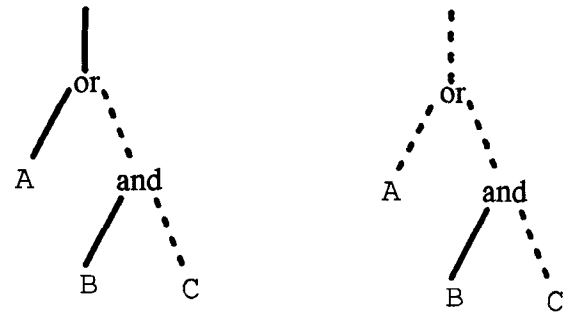
$\mathcal{V}'$ is the projection of $\mathcal{V}$ onto the conditions of $\mathcal{F}$ since $\mathcal{F}$ is a subexpression of $\mathcal{E}$.

One way to understand $I(\mathcal{E},\mathcal{V})$ is to interpret it as a flow on an expression tree for $\mathcal{E}$. Each condition (leaf node) is a source for a flow towards the root. Equation (*) says that subexpressions of an and/or expression that evaluate differently than the expression have their flows killed. $I(\mathcal{E},\mathcal{V})$ is the set of conditions with flow that reaches the root. Conditions whose flow does not reach the root are *masked* in the terminology of [5].

This is best illustrated with an example. Figure 4 shows computation of $eval(\mathcal{E},\mathcal{V})$ and $I(\mathcal{E},\mathcal{V})$ for a single MCDC pair computation for the decision

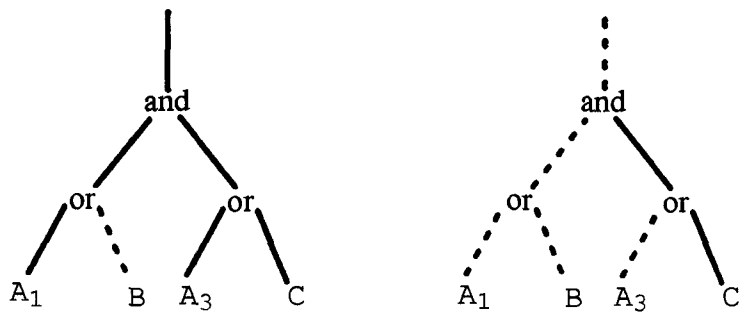
```
A or (B and C)
```

which has no coupling. Solid lines represent an evaluation of true and dashed lines an evaluation of false. Leaf conditions whose line type flows continuously to the



$\mathcal{V}_1$	$eval(\mathcal{E},\mathcal{V}_1)$	$I(\mathcal{E},\mathcal{V}_1)$	$\mathcal{V}_2$	$eval(\mathcal{E},\mathcal{V}_2)$	$I(\mathcal{E},\mathcal{V}_2)$	$I(\mathcal{E},\mathcal{V}_2) \cap I(\mathcal{E},\mathcal{V}_2)$
ttf	T	{A}	ftf	F	{AC}	{A}

Figure 4 -Influence trees for MCDC pair `ttf` and `ftf` for `A` or (`B` and `C`)



$\mathcal{V}_1$	$eval(\mathcal{E},\mathcal{V}_1)$	$I(\mathcal{E},\mathcal{V}_1)$	$\mathcal{V}_2$	$eval(\mathcal{E},\mathcal{V}_2)$	$I(\mathcal{E},\mathcal{V}_2)$	$I(\mathcal{E},\mathcal{V}_2) \cap I(\mathcal{E},\mathcal{V}_2)$
tftt	T	$\{A_1A_3C\}$	ffft	F	$\{A_1B\}$	$\{A_1\}$

Figure 5 -Influence trees for `tftt` and `ffft` of (`A or B`) `and` (`A or C`)

root of the tree have influence. Other conditions are masked. Figure 5 shows the same computation for the decision

```
A or B and (A or C)
```

which contains the coupled condition `A`.

With this machinery in hand, $\mathcal{V}_1$ and $\mathcal{V}_2$ form an MCDC pair for condition A in $\mathcal{E}$ if

(1) $\mathcal{V}_1$ and $\mathcal{V}_2$ are equal on conditions not equivalent to A.
(2) $eval(\mathcal{E},\mathcal{V}_1) \neq eval(\mathcal{E},\mathcal{V}_2)$.
(3) $I(\mathcal{E},\mathcal{V}_1) \cap I(\mathcal{E},\mathcal{V}_2) = \{A\}$.

In words, $\mathcal{V}_1$ is $\mathcal{V}_2$ with A "toggled" which toggles the evaluation of $\mathcal{E}$, but only A has influence on both evaluations—other conditions coupled with A are masked. When A is not coupled, only (1) and (2) need to be checked as they imply (3).

Adding influence (shown as {...}) to the table for `(A or B) and (A or C)` above gives,

	T	F
A_1	tftt{A_1A_3C}	ffft{A_1B}
B	ftft{BC}	ffft{A_1B}
A_3	tttf{A_1BA_3}	ftff{A_3C}
C	ftft{BC}	ftff{A_3C}

6.3 COVERAGE DETERMINATION AND TEST SUITE GENERATION

The operation of the Coverage Determination component and its relation with Test Generation is easiest to explain with an example. In Figure 6, the procedure `example()` is called three times by the harness procedure `test_it` (line 22) which produces the trace outputs shown in Figure 7. The four fields of trace output are source line number, statement type, decision outcome, and condition outcomes.

The Coverage Determination component examines the CFG derived from the source file and identifies all decision expressions. TSDT reads the trace file and constructs a summary of MCDC coverage using the definition of MCDC pair above. A complete MCDC test suite specification is obtained by calling Coverage Determination with an empty trace file.

The output above the dotted line in figure 8 reports the MCDC coverage obtained from trace output. This shows that the decision at line 9 is covered; line 10 needs to be tested true; line 12 requires a false test for the first condition and a true/false pair for the second condition.

The requirements to complete coverage are bundled into a set of tuple of (decision, set of boolean-vector) and passed as input to the EPA component. In the example, the requirements to complete coverage are:

```
{((y<7), {t}), (x>20 or y<10, {ft,ff})}
```

The EPA component returns a set of predicates in terms of input variables that defines a test suite specification. Each predicate is tested for feasibility by KITP and the infeasible tests are removed. If a different MCDC pair that is feasible is available to replace an infeasible test, then this pair can be added to the test suite. The final result is a test suite specification that maximizes MCDC coverage. TSDT attempts to minimize the number of specifications in a test suite specification.

The problem of finding a minimum test suite specification is known to be NP-Complete [13]. In such situations, heuristics often work well. TSDT employs a greedy algorithm (suggested in [10]), along with checking for graph reachability. Let `S` represent a test suite specification and $t_1, \ldots, t_n$ represent test input specifications. The algorithm (without reference to graph reachability) is:

```
1     Let S = {t1,...,tn}
2     Let S' = ∅ be the output
3
4     while (S ≠ ∅)
5     {
6           let t ∈ S, S ← S - {t}
7           foreach (s ∈ S)
8           {
9                 if (feasible(t ∧ s))
10                {
11                      t ← t ∧ s
12                      S ← S - {s}
13                }
14          }
15          S' ← S' ∪ {t}
16    }
17    return S'
```

Experience has shown this algorithm to produce adequate results. The reduced specification is passed to the solver which computes a test input that satisfies each predicate. In Figure 8 , Test Generation returns three predicates, and MCDC coverage is completed by three more executions - `example(5,10)`, `example(5,7)`, and `example(5,6)`.

TSDT computed the output below the dotted line to complete MCDC coverage. Line 12.1 F(ff) [1], indicates that coverage for condition 1 in the decision at source line 12 is obtained from test input [1]; this input forces each condition false and the decision false.

7 CONCLUSIONS

By formally characterizing and reasoning about collections of paths, rather than individual paths, we feel that significant improvements can be made in reasoning about program behavior.

A 50-line Ada procedure with 21 conditions and 14 decisions provides an example. This procedure contains 71,136 condition-level paths; 674 are feasible. The minimum test suite specification contains 10 tests. Our early prototype, VCTG, took 11 hours of computing time to make these determinations. TSDT has computed test inputs in less than 1/2 hour by reasoning about

```
1 -- example.ada
2
3
4 procedure example(x:integer; y:integer) is
5
6
7 begin
8
9 if x > 4 and y > x then
10   if y < 7 then
11     null ;
12   elsif x > 20 or y < 10 then
13     null ;
14   end if ;
15 end if ;
16
17
18 end example ;
19
20 with example ;
21
22 procedure test_it is
23 begin
24
25   example(6,5) ;
26   example(-2,-1);
27   example(21,22) ;
28 end ;
```

Figure 6

```
-- example.trc
9 if F tf
9 if F ft
9 if T tt
10 if F f
12 elsif T tf
```

Figure 7

```
source: ~/work/test/example.ada
trace : ~/work/test/example.trc

line 9 IF  x > 4 and y > x
9.1 covered by   3 T tt   2 F ft
9.2 covered by   3 T tt   1 F tf

line 10 IF  y < 7
10.1 uncovered          4 F f

line 12 ELSIF  x > 20 or y < 10
12.1 uncovered   5 T tf
12.2 uncovered
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Retest Specifications:
SUBPROGRAM-BODY--EXAMPLE
line 10 IF y < 7
10.1 T(t)[3]
line 12 ELSIF x > 20 or y < 10
12.1          F(ff)[1]
12.2 T(ft)[2]  F(ff)[1]
TEST SPECIFICATIONS:
1: ((Y > X and X > 4) and Y >= 7))
  and (Y >= 10 and X <= 20)
2: ((Y > X and X > 4) and Y >= 7)
  and (Y < 10 and X <= 20)
3: (Y > X and X > 4) and Y < 7
TEST INPUTS:
1: X = 5,
  Y = 10
2: X = 5,
  Y = 7
3: X = 5,
  Y = 6
```

Figure 8

path families and by settling for test suites larger than minimum. It is unrealistic to expect manual analysis to solve problems of this complexity in a reasonable amount of time without error.

Despite the potential improvements, we are concerned that the inherent size of these formulae and complexity in constructing them might outweigh the benefits of their generality. Both the size and complexity of these formulae adversely impact the prover and the solver. Additional research must be conducted and empirical data must be gathered before we can determine whether this approach is effective for test data generation.

With respect to feasibility analysis, TSDT was run on a large sample of avionics code involving boolean, integer, integer subrange, and enumerated type data. KITP was very effective at proving infeasibility conjectures

with the exception of those conjectures requiring knowledge of type information. KITP is an untyped prover; additional axioms must be added to distinguish and effectively deal with typed data. We have been unable to demonstrate the effectiveness of KITP when type information is axiomatized; manual analysis indicates that KITP would be 100% effective at determining infeasible paths if this information were included. Kestrel is working on a typed prover.

A majority of the potential test input specifications that were proved infeasible by KITP required very shallow reasoning. It is our opinion that many of these specifications could be proved infeasible by less sophisticated techniques (e.g., simple term rewriting). Nevertheless, a broader sample of programs containing other data types (e.g., arrays) may reveal that the full power of an automated theorem prover is required.

Acknowledgement

The authors wish to acknowledge the contributions of other members of the TSDT team: John Chilenski, Ron Jobmann, Ed Marvin, Phil Newcomb, Stephen Nicoud, Stan Payzer, and Les Richey (Boeing); Allen Goldberg, T.C. Wang (Kestrel); Richard King, and Gordon Kotik (Reasoning Systems).

References

[1] W. Bibel. On matrices with interconnections. In *Journal of the ACM*, 28(4), October, 1981, pages 633–664.

[2] J. Bicevskis, J. Borzovs, U. Staujums, A. Zarins, and E. Miller. SMOTL—a system to construct samples for data processing and program debugging. In *IEEE Transactions on Software Engineering*, SE–5, 8, August, 1990, pages 60–66.

[3] B.W. Boehm, The high cost of software. In *Practical Strategies for Developing Large Software Systems*. E. Horowitz (editor), Addison–Wesley, Reading, MA, 1975.

[4] C.L. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., 1973.

[5] J.J. Chilenski and S.P. Miller. *Applicability of Modified Condition/Decision Coverage to Software Testing*. Copyright® 1993, The Boeing Company and Rockwell International Corporation, Submitted for publication.

[6] J.J. Chilenski and P.H. Newcomb, *Formal Specification Tools for Test Coverage Analysis*, In preparation.

[7] L.A. Clarke. A system to generate test data and symbolically execute programs. In *IEEE Transactions on Software Engineering*, SE–2, 3, September, 1976, pages 215–222.

[8] L.A. Clarke, D.J. Richardson, and S.J. Zeil. TEAM: A support environment for testing, evaluation, and analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium of Practical Software Development*. November, 1988, ACM, pages 153–162.

[9] P.G. Frankl, and E.J. Weyuker. An applicable family of data flow testing criteria, In *IEEE Transactions on Software Engineering*. SE–14, October 10, 1988, pages 1483–1898.

[10] A. Goldberg. *Design Study: Computation of RDTs and Test Specifications*. Kestrel Institute, December 1992.

[11] A. Goldberg. *Final Report: Valid Condition Table Generation Pilot Study*. Kestrel Institute, April 14, 1992.

[12] A. Goldberg and D. Zimmerman. *Design Study for RDTs and Beyond*. Kestrel Institute, December 10, 1992.

[13] A. Goldberg, D. Zimmerman, and T.C. Wang. Applications of feasible path analysis to program testing. To be presented at the International Symposium on Software Testing and Analysis, (Seattle, WA August 17–19, 1994).

[14] W.E. Howden. Symbolic testing and the DISSECT symbolic evaluation techniques. In *IEEE Transactions on Software Engineering*, SE–4, 4, 1977, pages 266–278.

[15] Kestrel Institute. *VCTG 4.0 User's Guide*. December 18, 1992.

[16] Kestrel Institute. *VCTG(EPA) 5.0 Interim Usage Notes*. April 2, 1993.

[17] Kestrel Institute. *Kestrel Interactive Theorem Prover (KITP) User's Guide*. October 20, 1993.

[18] T.E. Lindquist and J.R. Jenkins. Test–case generation with IOGen, In *IEEE Software. January 1988*, pages 72–79.

[19] RTCA, Inc. *Software Considerations in Airborne Systems and Equipment Certification*. Document No. RTCA/DO–178B, Dec. 1992.

[20] C. Ramomoorthy, S. Ho, and W. Chen. On the automated generation of program test data, In *IEEE Transactions on Software Engineering*, SE-2, 4, December 1976, pages 293-300.

[21] Reasoning Systems Incorporated. *Refine/Ada Programmers Guide*. July 26, 1992.

[22] Reasoning Systems Incorporated. *Refine/Ada User's Guide*. July 26, 1992.

[23] Reasoning Systems Incorporated. *Refine ™ 3.0 User's Guide*. May 25, 1990.

[24] P.B. Andrews, Theorem proving via general matings. In *Journal of the ACM*, 28(2), April, 1981, pages 193-214.

[25] Jeffery Mark Siskind. *Screaming Yellow Zonkers*. M.I.T. Artificial Intelligence Laboratory, Draft of September 29, 1991. Available by anonymous FTP from the directory ftp.ai.mit.edu:/pub/screamer.

[26] J.M. Siskind and D.A. McAllester. *Screamer: A Portable Efficient Implementation of Nondeterministic Common Lisp*. Available by anonymous FTP from the directory ftp.ai.mit.edu:/pub/screamer.

[27] J.M. Siskind and D.A. McAllester. Nondeterministic lisp as a substrate for constraint logic programming. In *Proceedings of the 11th Annual Conference on Artificial Intelligence*, July, 1993, pages 133-138.

[28] T.C. Wang. Designing examples for semantically guided hierarchical deduction. In *9th International Joint Conference on Artificial Intelligence*, August, 1985, pages 1201-1207.

[29] T.C. Wang and W.W. Bledsoe. Hierarchical Deduction. In *Journal of Automated Reasoning* 3, 1, March 1987, pages 35-77.

[30] E.J. Weyuker. An empirical study of the complexity of data flow testing. *In Proceedings of the Second Workshop on Software Testing, Analysis, and Verification.* July, 1988, IEEE Computer Society, pages 188-195.

[31] M.R. Woodward, D. Hedley, and M.A. Hennel. Experience with path analysis and testing. In *Tutorial: Software Testing and Validation Techniques*. E. Miller and W. Howden Eds. IEEE Computer Society Press, Los Alamitos, CA, 1981, pages 194-206.

[32] L. Wos and A. Robinson. Paramodulation and set of support. In *Proceedings of the IRIA Symposium on Automated Demonstration.* 1968, Springer-Verlag, pages 276-310.