Generating Test Data From Requirements/Specifications: Phase II Final Report

`Prepared For:` Rockwell Collins, Inc,
`Contract Monitor:` Dave Statezni

Principal Investigator: A. Jefferson Offutt
George Mason University

August 29, 1998

EXECUTIVE SUMMARY

This report presents results for the Rockwell Collins Inc. sponsored project on generating test data from requirements/specifications, which started January 1, 1998. The purpose of this project is to improve our ability to test software that needs to be highly reliable by developing formal techniques for generating test cases from formal specificational descriptions of the software. Formal specifications represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can be easily manipulated by automated means.

This Phase II, 1998 report presents results and strategies for practically applying test cases generated according to the criteria presented in the Phase I, 1997 report [Off98]. This report presents a small empirical evaluation of the test criteria, and algorithms for solving various problems that arise when applying test cases developed from requirements/specifications. One significant problem in specification-based test data generation is that of reaching the proper program state necessary to execute a particular test case. Given a test case that must start in a particular state S, the test case *prefix* is a sequence of inputs that will put the software into state S. We have addressed this problem in two ways. First is to combine various test cases to be run in *test sequences* that are *ordered* in such a way that each test case leaves the software in the state necessary to run the subsequent test case. An algorithm is presented that attempts to find test case sequences that are *optimal* in the sense that the fewest possible number of test cases are used. To handle situations where it is desired to run each test case independently, an algorithm for directly deriving test sequences is presented. This report also presents procedures for removing redundant test case values, and develops the idea of "sequence-pair" testing, which was presented in the 1997 Phase I report, into a more general idea of "interaction-pair" testing.

1 INTRODUCTION

Software system level tests have traditionally been created based on informal, ad-hoc analyses of the software requirements. This leads to inconsistent results, problems in understanding the goals and results of testing, and an overall lack of effectiveness in testing. This research project is attempting to establish formal criteria and processes for generating system-level tests from functional requirements/specifications.

The purpose of this project is to improve our ability to test software that needs to be highly reliable. Formal specifications represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can be automatically manipulated.

Thus far, this work has resulted in a general model for developing test inputs from state-based specifications. This model includes several related criteria for generating test data from formal specifications. These criteria provide a formal process, a method for measuring tests, and a basis for full automation of test data generation.

The principal results in this report are algorithms to solve problems in practically applying specification-based tests. Also presented are empirical results from a small case study evaluation of the test criteria. The report summarizes the results from Phase I (during 1997) [Off98], presents the current year goals, presents and describes various algorithms, and demonstrates their use through examples.

This report uses the following definitions. *Test requirements* are specific things that must be satisfied or covered during testing; e.g., reaching statements are the requirements for statement coverage. *Test specifications* are specific descriptions of test cases, often associated with test requirements or criteria. For statement coverage, test specifications are the conditions necessary to reach a statement. A *testing criterion* is a rule or collection of rules that impose test requirements on a set of test cases. A *testing technique* guides the tester through the testing process by including a testing criterion and a process for creating test case values.

A *test* or *test case* is a general software artifact that includes test case input values, expected outputs for the test case, and any inputs that are necessary to put the software system into the state that is appropriate for the test input values. A test specification language (TSL) is a language that can be used to describe all components of a test case. The components that we consider are *test case values*, *prefix values*, *verify values*, *exit commands*, and *expected outputs*. Test case values directly satisfy the test requirements, and the other components supply supporting values. A *test case value* is the essential part of a test case, the values that come from the test requirements. It may be a command, user inputs, or software function and values for its parameters. In state-based software, test case values are usually derived directly from triggering events and preconditions for transitions. A test case *prefix value* includes all inputs necessary to reach the pre-state and to give the triggering event variables their before-values. Any inputs that are necessary to show the results are *verify values*, and *exit commands* depend on the system being tested. *Expected outputs* are created from the after-values of the triggering events and any postconditions that are associated with the transition.

2 SUMMARY OF PHASE I

Phase I of this project was carried out during summer 1997, and established the long term goal of improving our ability to test software that needs to be highly reliable by developing formal techniques for generating test cases from formal specificational descriptions of the software [Off98]. This research addressed the problem of developing formalizable, measurable criteria for generating test cases from specifications.

During Phase I a general **model** for developing test inputs from state-based specifications was developed. This model includes several criteria criteria for generating tests, a **derivation process** for obtaining the test cases, an **example** for a small system, and **test cases** from specifications of an industrial system. The test data generation model includes **techniques** for generating tests at several levels of abstraction for specifications, including the complete transition sequence level, the transition-pair level, and the detailed transition level. These techniques are novel in that they provide **coverage criteria** that are based on the specifications. It is thought that these are the first formal coverage criteria for functional specifications. The tests are made up of several parts, including test **prefixes** that contain inputs necessary to put the software into the appropriate state for the test values. A test generation process was also developed, which includes several steps for transforming specifications to tests.

Results from applying the model and process to a small example were presented in the final report. This case study was evaluated using Atac [HL92] to measure decision coverage, and the technique was found to achieve a high level of coverage. This result indicates that this technique can benefit software developers who construct formal specifications during development.

As an additional validation, tests were generated for specifications of an industrial software system supplied by Rockwell Collins, the Flight Guidance System. Construction of these tests resulted in several modifications to this technique, and found at least one problem with the specification.

2.1 Summary of Phase II Goals

The current year research is building on this basis in several ways. The first results presented are from a small case study that applied the test criteria of transition coverage and full predicate coverage to the well known cruise control example.

The first algorithm is for test case *ordering*. Test cases are created for particular states or transitions in the system. Each test has some *pre-state* associated with it; the state the system must be in for the test to be executed. Tests can be ordered so as reduce the amount of execution, by ordering test cases such that the post-state of some test case t_i yields the correct pre-state for test case t_{i+1}. This report presents an algorithm that attempts to find optimal ordering of test cases.

Another focus is on test case *prefixes*. A specification-based test for state-based software is a collection of inputs to the software. The test may or may not start at the initial state; if it does not, the complete test case must include additional inputs that will start at the initial state and put the software into the state where the test should begin. Given a test case that must start in a particular state S, the prefix is a sequence of inputs that will put the software into state S. This report presents an algorithm that determines test case prefixes for an arbitrary state.

When a number of tests are created automatically, it is natural for some to be redundant. Different test requirements and specifications will lead to similar or the same tests. As part of this work, analysis techniques have been developed to remove redundant tests so as to minimize the ordering and prefix generation that is required.

A *sequence-pair* is a pair of states that can be entered in sequence. In Phase I, the concept of testing sequence-pairs was introduced, based on the observation that some pairs of states have interactions that should be carefully tested. In Phase II, the notion of sequence-pair testing is expanded to **interaction-pair** testing, where an interaction-pair is a pair of states that have some

data or control interaction.

One observation from the case study in Phase I and the industrial example was that applying the technique takes a lot of very detailed hand analysis. Both to save costs and improve accuracy, a long term goal is to develop automated tool support to transform formal functional specifications into effective test cases. An eventual goal is to build an automatic test data generation tool for this technique.

3 CASE STUDY

An empirical study has been undertaken to demonstrate the feasibility of the criteria from Phase I. The goal was to demonstrate that the specification-based criteria can be effectively used; it is hoped to evaluate them more fully in the future. The methodology and empirical subjects are described first, then the processes used to generate tests for each criterion are described in detail. Then the implementation used, and the faults that were generated are described, and finally results and analysis are given.

3.1 Methodology

Two measurements of the criteria have been carried out. Tests were created and then measured on the basis of the structural coverage criterion of decision testing, and then the tests were measured in terms of their fault-detection abilities. One moderate size program was used, representative faults were seeded, and test cases were generated by hand.

Cruise control is a common example in the literature [Atl94, Jin96], and specifications are readily available. The specifications for a version of the system (note that it does not model the throttle) has four states: OFF (the initial state), INACTIVE, CRUISE, and OVERRIDE. The system's environmental conditions indicate whether the automobile's ignition is on (*Ignited*), the engine is running (*Running*), the automobile is going too fast to be controlled (*Toofast*), the brake pedal is being pressed (*Brake*), and whether the cruise control level is set to *Activate*, *Deactivate*, or *Resume*.

Previous Mode	Ignited	Running	Toofast	Brake	Activate	Deactivate	Resume	New Mode
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	-	-	Off
	t	t	-	f	@T	-	-	Cruise
Cruise	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	-	@T	-	-	-	-	
	t	t	f	@T	-	-	-	Override
	t	t	f	-	-	@T	-	
Override	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	t	-	f	@T	-	-	Cruise
	t	t	-	f	-	-	@T	

Table 1: SCR Specifications for the Cruise Control System

Each row in the table specifies a conditioned event that activates a transition from the mode on the left to the mode on the right. A table entry of @T or @F under a column header C represents a triggering event @T(C) or @F(C). This means that the value of C must change for the transition to be taken, that is, "@T(C)" means C must change from false to true, and "@F(C)" means C must change from true to false. A table entry of `t` or `f` represents a WHEN condition. WHEN[C] means the transition can only be taken if C is true, and WHEN[$\neg$C] means it can only be taken if C is false. If the value of a condition C does not affect a conditioned event, the table entry is marked with a hyphen "-" (don't care condition).

Table 2 shows the transitions of the specification with the trigger events expanded in predicate form (as described in the phase I report), numbered P_1 through P_{12}. Figure 1 shows the specification graph, with the edges labeled with the predicate numbers.

P_1	OFF	$\neg Ignited \wedge Ignited'$	INACTIVE
P_2	INACTIVE	$Ignited \wedge \neg Ignited'$	OFF
P_3	INACTIVE	$\neg Activate \wedge Ignited \wedge Running \wedge \neg Brake \wedge Activate'$	CRUISE
P_4	CRUISE	$Ignited \wedge \neg Ignited'$	OFF
P_5	CRUISE	$Running \wedge Ignited \wedge \neg Running'$	INACTIVE
P_6	CRUISE	$\neg Toofast \wedge Ignited \wedge Toofast'$	INACTIVE
P_7	CRUISE	$\neg Brake \wedge Ignited \wedge Running \wedge \neg Toofast \wedge Brake'$	OVERRIDE
P_8	CRUISE	$\neg Deactivate \wedge Ignited \wedge Running \wedge \neg Toofast \wedge Deactivate'$	OVERRIDE
P_9	OVERRIDE	$Ignited \wedge \neg Ignited'$	OFF
P_{10}	OVERRIDE	$Running \wedge Ignited \wedge \neg Running'$	INACTIVE
P_{11}	OVERRIDE	$\neg Activate \wedge Ignited \wedge Running \wedge \neg Brake \wedge Activate'$	CRUISE
P_{12}	OVERRIDE	$\neg Resume \wedge Ignited \wedge Running \wedge \neg Brake \wedge Resume'$	CRUISE

Table 2: Expanded Cruise Control Specification Predicates

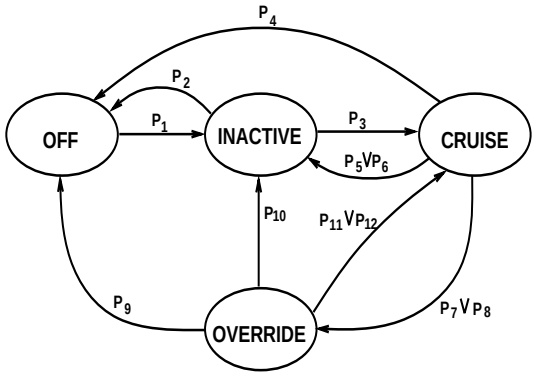

Figure 1: Specification Graph for Cruise Control

3.2 Test Generation

To avoid bias, tests were created independently from the faults, by different people. The tests were created manually before any execution. Each test case was executed against each buggy version of **Cruise**. After each execution, failures (if any) were identified. The number of faults detected was recorded and used in the analysis. The rest of this subsection discusses in detail how tests were generated. This serves both to elaborate on the empirical methodology, as well as to illustrate the criteria defined previously. Although much of this discussion repeats what was provided in the Phase I, 1997 report [Off98], our experience has led to some minor changes, thus the repetition is justified. Because the transition coverage criterion is subsumed by full predicate coverage, it is not described separately. Transition coverage test cases can be taken from the "valid" specifications in the full predicate tests, which are listed first for each transition.

3.2.1 Full predicate coverage criterion

There are nine transitions in the cruise control specifications, and twelve disjunctive predicates. For convenience, the technique is applied by considering each predicate specification separately. Both the before-values and after-values of the triggering event should be separately tested. For SCR, this is handled by treating @ as an operator and expanding it algebraically. If X represents a before-value and X' an after-value, the relevant expansions are:

- $@T(X) \equiv \neg X \wedge X'$
- $@T(X \wedge Y) \equiv \neg(X \wedge Y) \wedge (X' \wedge Y') \equiv (\neg X \vee \neg Y) \wedge X' \wedge Y'$
- $@T(X \vee Y) \equiv \neg(X \vee Y) \wedge (X' \vee Y') \equiv \neg X \wedge \neg Y \wedge X' \wedge Y'$

There are 54 separate test case requirements for the full predicate coverage level, which were given in the Phase I report [Off98]. The third transition, P_3, is used to illustrate the test case requirement derivation. The variable values are taken from the predicates, and are shown as T, F, t, f, and -. A T or F means the clause is triggering, and the table contains a before-value and after-value. The values for the test case are the new value for the triggering clause (T or F), and the t and f values from the WHEN conditions. The expected output for the test specification is derived from the triggering event, the post-state, and any terms or variables that are defined as a result of the transition. P_3 has four clauses:

$@T Activate \wedge Ignited \wedge Running \wedge \neg Brake$

and its expanded version is:

$\neg Activate \wedge Ignited \wedge Running \wedge \neg Brake \wedge Activate'$

Its six test case requirements are:

	Pre State	*Activate*	*Ignited*	*Running*	*Brake*	*Activate'*	Post State
1.	INACTIVE	F	t	t	f	T	CRUISE
2.	INACTIVE	F	f	t	f	T	INACTIVE
3.	INACTIVE	F	t	f	f	T	INACTIVE
4.	INACTIVE	F	t	t	t	T	INACTIVE
5.	INACTIVE	T	t	t	f	T	INACTIVE
6.	INACTIVE	F	t	t	f	F	INACTIVE

The first row is the predicate as it appears in the specification; every clause is `True`. This corresponds to a valid test input (and is also the transition coverage test case for this transition). The subsequent rows make each clause `False` in turn, corresponding to invalid inputs. Because there are no OR operators, the full predicate coverage criterion is satisfied by holding all other clauses `True`. The post-states are the expected values. Five of them represent invalid transitions, and it is assumed that the software will remain in the same state.

Test specifications

The actual test specifications and test scripts are mechanically derived from the test requirements. The predicate P3 is chosen as an illustrative example. P3 has six full predicate level tests. For the first test case for P3, the test case must reach the INACTIVE state; this forms the `Prefix`. The `Test case values` set the before-value for the triggering event, and the WHEN condition variables of *Inactive*, *Running*, and *Brake*, and then sets *Activate* to be `True` as the triggering event. The `Verify` and `Exit` parts of the specifications are not shown, as they depend on the software. The software can safely be assumed to automatically print the current state, and to not require an exit.

1. Test specification P3-1:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
Test case value:	*Activate*	= `False`	– Trigger before-value
	Running	= `True`	– Condition variable
	Brake	= `False`	– Condition variable
	Activate	= `True`	– Triggering event
Expected outputs:	CRUISE		

2. Test specification P3-2:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
Test case value:	*Activate*	= `True`	– Trigger before-value
	Running	= `True`	– Condition variable
	Brake	= `False`	– Condition variable
	Activate	= `True`	– Triggering event
Expected outputs:	INACTIVE		

3. Test specification P3-3:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
Test case value:	*Activate*	= `False`	– Trigger before-value
	Ignited	= `False`	– Condition variable
	Running	= `True`	– Condition variable
	Brake	= `False`	– Condition variable
	Activate	= `True`	– Triggering event
Expected outputs:	INACTIVE		

4. Test specification P3-4:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
Test case value:	*Activate*	= `False`	– Trigger before-value

	Running	= `False`	– Condition variable
	Brake	= `False`	– Condition variable
	Activate	= `True`	– Triggering event
Expected outputs:	INACTIVE		

5. Test specification P3-5:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
Test case value:	*Activate*	= `False`	– Trigger before-value
	Running	= `True`	– Condition variable
	Brake	= `True`	– Condition variable
	Activate	= `True`	– Triggering event
Expected outputs:	INACTIVE		

6. Test specification P3-6:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
Test case value:	*Activate*	= `False`	– Trigger before-value
	Running	= `True`	– Condition variable
	Brake	= `False`	– Condition variable
	Activate	= `False`	– Triggering event
Expected outputs:	INACTIVE		

There are several interesting points to note about these test specifications. First, it should be clear that there is some redundancy; some of the condition variables will not need to be explicitly set, as they will already have the appropriate values. While this is true, the analysis necessary to decide what values do and do not need to be set may outweighs the small savings that could result from eliminating a few variable assignments. It is probable, however, that this could be done automatically. Jin [Jin96] provided algorithms for deriving invariants on modes; these could be used to directly eliminate unneeded variable assignments. This method used a static analysis. A dynamic analysis that uses the information in the test specification could be used to potentially eliminate more variable assignments. This issue is explored in Section 6. Another interesting point is the derivation of the prefix part of the test specification. Reaching the pre-state is essentially a reachability problem. Given a control flow graph of a program, it is an undecidable problem to find a test case that reaches a particular statement. Although no theoretical analysis has been done as yet, most state-based systems are finite and deterministic. Thus, it seems likely that this problem is solvable for specification graphs derived from state-based systems.

Test scripts are simple rewrites of test specifications with modifications made for the input requirements of the program being tested. The test script for the first test specification above is:

```
Ignited = True
Activate = False
Running = True
Brake = False
Activate = True
```

3.2.2 Transition-pair coverage criterion

At the transition-pair level, each state is considered separately. Each input transition into the state is matched with each transition out of the state, and the combination is used to create test

requirements, which are ordered pairs of predicates. The ordered pairs are turned into ordered pairs of inputs to form test specifications.

Following are the test requirements for the four states.

OFF	**CRUISE**
1. P2 : P1	1. P3 : P4
2. P4 : P1	2. P3 : (P5 OR P6)
3. P9 : P1	3. P3 : (P7 OR P8)
INACTIVE	4. (P11 OR P12) : P4
1. P1 : P2	5. (P11 OR P12) : (P5 OR P6)
2. P1 : P3	6. (P11 OR P12) : (P7 OR P8)
3. P10 : P2	**OVERRIDE**
4. P10 : P3	1. (P7 OR P8) : P9
5. (P5 OR P6) : P2	2. (P7 OR P8) : P10
6. (P5 OR P6) : P3	3. (P7 OR P8) : (P11 OR P12)

These ordered pairs are transformed into predicates from Table 2. The "**OR**" entries result from the transitions that have two conditions; either condition could be satisfied to take that transition.

Test specifications

The actual test specifications and test scripts are mechanically derived from the above test requirements, and are too numerous to list. The requirements for the OFF state are chosen as an illustrative example. OFF has three transition-pair coverage level tests. For the first test case for OFF, the test case must reach the INACTIVE state; this forms the `Prefix`. Then the test case must pass through transitions $P1$ and $P2$.

1. Test specification OFF-1:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
Test case values:	*Ignited*	= `False`	– P2 Triggering event
	Ignited	= `True`	– P1 Triggering event
Expected outputs:	INACTIVE		

2. Test specification OFF-2:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
	Ignited	= `True`	– P3 Condition variable
	Running	= `True`	– P3 Condition variable
	Brake	= `False`	– P3 Condition variable
	Activate	= `True`	– Reach CRUISE state
Test case values:	*Ignited*	= `False`	– P4 Triggering event
	Ignited	= `True`	– P1 Triggering event
Expected outputs:	INACTIVE		

3. Test specification OFF-3:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
	Ignited	= `True`	– P3 Condition variable
	Running	= `True`	– P3 Condition variable
	Brake	= `False`	– P3 Condition variable
	Activate	= `True`	– Reach CRUISE state
	Ignited	= `True`	– P7 Condition variable
	Running	= `True`	– P7 Condition variable
	Toofast	= `False`	– P7 Condition variable
	Brake	= `True`	– Reach OVERRIDE state
Test case values:	*Ignited*	= `False`	– P9 Triggering event
	Ignited	= `True`	– P1 Triggering event
Expected outputs:	INACTIVE		

3.2.3 Complete sequence criteria

At the complete sequence level, test engineers must use their experience and judgment to develop sequences of states that should be tested. To do this well requires experience with testing, experience with programming, and knowledge of the domain. These tests are omitted in this case study.

3.3 Implementation and Faults

A model of the cruise control problem was implemented in about 400 lines of C. Cruise has seven functions, 184 blocks, and 174 decisions. The program accepts pairs of variable:values, where a value can be 't', 'f', 'T', or 'F'. Upper case inputs signify a triggering event. For convenience, the program was implemented so that the pre-state could be either set with a test case `Prefix`, or explicitly by entering the name of a state.

Twenty-five faults were created by hand and each was inserted into a separate version of the program. Most of these faults are based on mutation-style modifications, and most were in the logic that implemented the state machine. Four were naturally occurring faults, made during initial implementation.

3.4 Results and Analysis

As a way to measure the quality of these tests, block and decision coverage was computed using the full predicate test cases. The coverage was measured using Atac [HL92]. Of the 174 decisions, 5 are infeasible, leaving 169. The results are shown in Figure 2. The 54 test cases covered 163 of the blocks (89%) and 155 of the decisions (95%). Of the 19 uncovered decisions, five were infeasible, and eleven were related to input parameters that were not used during testing. That is, these eleven decisions were not related to the functional specifications. The remaining three decisions were left uncovered because the variables *Activate*, *Deactivate*, and *Resume* are only used as triggering events in the specifications, not condition variables. Thus, there are statements in the software that handle assignments to these variables as WHEN conditions that are never executed. Although there have been very few published studies on the ability of specification-based tests to satisfy code-based coverage criteria, these results seem very promising.

The other measurement was for the fault-detection ability of the tests. Twelve test cases were generated for the transition coverage criterion, and an additional forty-two for the full predicate criterion (making 54 total). As a control comparison, 54 additional test cases were generated randomly. Although 25 versions of Cruise were created, each one containing one fault, one was such that the program goes into an infinite loop on any input. Since this fault was so trivial, it was discarded. Results from the three sets of test data are shown in Table 3.

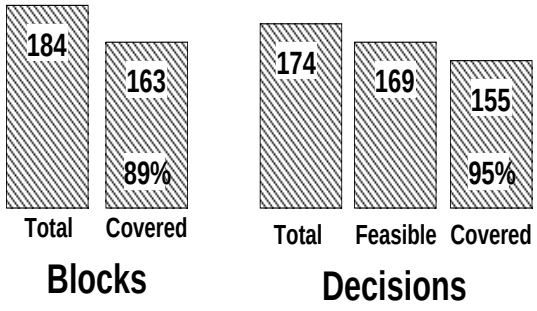

Figure 2: Branch and Decision Coverage Results

	Random	Transition	Full Predicate
number of test cases	54	12	54
faults found	15	15	20
faults missed	9	9	4
percent coverage	62.5%	62.5%	83.3%

Table 3: Faults Detected

Detailed analysis of the faults showed that three of the four faults that the full predicate tests missed could not have been found with the methodology used. The implementation runs in one of two modes. In one mode, the test engineer explicitly sets a pre-state by entering the state name. In the other mode, the software always starts at the initial state, and a test case prefix must be included as part of the test case. The prefix should include inputs to reach the pre-state. All of the tests that were used in this study explicitly set the pre-state, and three of the four faults that were missed could not be found if the pre-state is explicitly set. These faults were in statements that were not executed if the prefix was explicitly set. None of the three sets of tests found these three faults. The other fault that the full predicate tests missed was not found by either of the other two sets of inputs. Of the other five faults missed by the random and the transition tests, two were the same, and the other three were different. All of the naturally occurring faults were found by all three sets of tests.

The goals of this empirical pilot study were twofold. The first goal was to see if the specification-based testing criteria could be practically applied. The second was to make a preliminary evaluation of their merits by evaluating the branch coverage and fault coverage. Both goals were satisfied; the criteria were applied and worked well. They performed better than random generation of test cases. However, there are several limitations to the interpretation of the results. First, Cruise is of moderate size; longer and more complicated programs are needed. Second, the 25 faults inserted into Cruise were generated intuitively. More study should be carried out to reveal the types of faults that can be detected by system testing. In Phase III, we hope to carry out a more complete empirical evaluation.

4 ORDERING OF TEST CASES

The test case generation model creates test case values that are associated with specific portions of the test specification graph. For example, a test case value may be designed to test one transition in the graph. Thus, a test case value has some *pre-state* associated with it; the state the system must be in for the test to be executed, and each test case value needs to have values associated with it to reach that pre-state. This section presents an algorithm that attempts to find an ordering among the test cases such that the post-state of a test case t_i yields the correct pre-state for test case t_{i+1}. This kind of ordering can reduce the amount of execution by allowing tests to be used for two purposes, to satisfy test requirements and to supply prefixes for other tests.

The algorithm in this section attempts to find an optimal ordering of test cases for execution by constructing *test sequences*, which are sequences of tests that will be executed sequentially in one test execution. The algorithm attempts to achieve the following goals:

1. Include all test cases
2. Include as many unique test cases in each sequence as possible
3. Include the fewest possible number of *redundant* test cases (test cases that are included more than once)

This is not a problem that can have a simple solution. In fact, this is an NP-complete optimization problem, and there is no polynomial-time solution. The approach taken here is to approximate optimality, using heuristics that will always satisfy goal one, but may not fully satisfy goal three.

The test case ordering algorithm consists of two major steps. Both use a *Test Case Graph*, which represents ordering relationships among the test cases. In a test case graph (TCG), nodes represent test cases, and edges represent ordering dependencies among the test cases. The edges are constructed according to the following rule:

> If the post-state for test case value t_i is the pre-state for test case value t_j, then there is an edge from t_i to t_j.

The Step I algorithm (OrderTestCases Algorithm, Step I, in Figure 6) uses a depth-first search technique on the TCG to attempt to put each test case value into a test sequence with as little redundancy as possible. If there are test case values that are not included in any test sequence (*uncovered*), Step II (OrderTestCases Algorithm, Step II, in Figure 8) works backwards through the TCG to try to find the shortest path from each remaining test case value to an initial test case value. Step II also tries to put as many uncovered test cases as possible into each test sequence. It does this by pre-computing *weights* and *distances* for each node in the TCG (using the Weight and Distance Algorithm, in Figure 7). The distance is the shortest number of nodes from an initial node to the uncovered node, and the weight is the number of uncovered nodes on that path.

Step I and Step II are called from a main algorithm (OrderTestCasesMain Algorithm, in Figure 4). Figure 3 shows the call relationships among these algorithms. In the rest of this section, the algorithms are described in detail, then several illustrative examples are given.

4.1 Order Test Cases Algorithms

The main algorithm is shown in Figure 4. It calls the algorithm OrderTestCases (in Figure 6), and if there are still unused test cases, it calls FinishOrderTestCases (in Figure 7).

Each test sequence starts from an *initial test case*, which is a test case for which the pre-state is an initial state in the specification graph. First, we try to include as many uncovered test cases as possible in one test sequence. OrderTestCases, Step I uses a depth-first search to build test sequences. Each test case value is initially marked as "uncovered", then for each test case value t, test cases whose post-states are the pre-states for t are saved in the set `PrevTCs(t)` and test cases

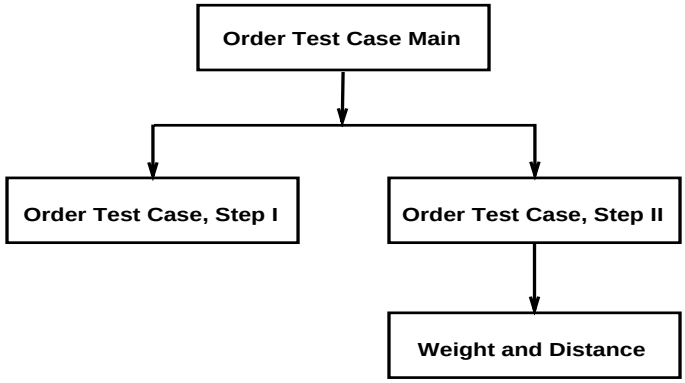

Figure 3: **Call Relationships Among the Test Case Ordering Algorithms**

---

algorithm: **OrderTestCasesMain (TestCaseGraph)**
input: `A graph that indicates ordering relationships among test cases.`
output: `Test Sequence Set.`
output criteria: `Fewest number of sequences with fewest number of test cases.`
declare: `Status -- A variable that represents the coverage status of TestCaseGraph for all test cases. Has two values: Finished, UnFinished.`

```
OrderTestCasesMain (TestCaseGraph)
BEGIN -- Algorithm OrderTestCasesMain
    Status = OrderTestCases (TestCaseGraph)
    IF (Status = NotFinished)
        FinishOrderTestCases (TestCasesGraph)
    END IF
END Algorithm OrderTestCasesMain
```

Figure 4: **The OrderTestCases Main Algorithm**

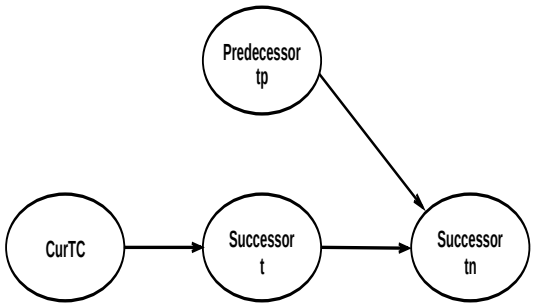

Figure 5: **2-Step Lookahead Decision**

whose pre-states are the post-states for t are saved in the set `NextTCs(t)`. These are predecessors and successors of t.

Then a set of test sequences are built, starting with initial test cases. A test sequence is initialized to be a test case with no previous test case (an initial test case), and that has not been covered, or has been covered fewer times than any other initial test case. At each step in the test sequence construction loop, the NextTCs set for the current test case (`CurTC`) is examined for uncovered test cases. If one exists, it becomes the next current test case. If not, the algorithm uses a "2-step lookahead" heuristic, as illustrated in Figure 5. If there is no uncovered successor for `CurTC`, the algorithm looks for a successor to `CurTC` (`t` in Figure 5) that has an uncovered successor (`tn`) with no uncovered predecessor (`tp`). If one is found, the successor to `CurTC` is concatenated to the test sequence and its successor with no uncovered predecessor becomes the `CurTC`.

If no such successor to `CurTC` exists, the test sequence is finished. If the loop was completed and no tests were added (`UncoveredTCs` is unchanged), this means that Step I has finished without including all the test cases, and the algorithm terminates with the flag `NotFinished`. Otherwise, the test sequence is saved, and the algorithm proceeds to the next test sequence. If all test cases have been covered, then Step I terminates with the flag `Finished`.

There may be some test cases that cannot be reached with the 2-step lookahead approach. For this case, the algorithm FinishOrderTestCases shown in Figure 7 is given. This algorithm finds paths from initial test cases to uncovered test cases, according to two criteria:

1. Each test sequence should include the maximum number of uncovered test cases.

2. When an uncovered test case must be included in a new test sequence that contains no other uncovered test cases, the test sequence is built by finding a shortest path through the test case graph to the uncovered test case.

First, weights and distances are calculated for the graph. The *distance* is the shortest number of nodes from an initial node to the uncovered node, and the *weight* is the number of uncovered nodes on that path. This is computed using CalculateWeightDistance in Figure 8. Next, the algorithm loops until there are no uncovered test cases. New test sequences are constructed by working backwards through the TCG. An uncovered test case with maximum weight is selected, and it is added to the current test sequence. If the current test case (`CurTC`) is an initial test case (its distance is 0), then the current test sequence is completed. Otherwise, the predecessor to `CurTC` that has the maximum number of uncovered test cases on its shortest path is chosen. This is the predecessor with the largest weight. If all predecessors have no weight (no uncovered test cases on their shortest paths), then the test case on the shortest path is chosen (the smallest distance value).

```
algorithm:        OrderTestCases (TestCaseGraph)
input:            A graph that indicates ordering relationships among test cases (TCG).
output:           Test Sequence Set.
output criteria:  Fewest number of sequences with fewest number of test cases.
declare:          UncoveredTCs -- Set of test cases that have not yet been covered.
                  PrevTCs(t) -- The set of predecessor test cases for t.
                  NextTCs(t) -- The set of successor test cases for t.
                  testSequence -- A sequence of test cases that will be executed in order.
                  testSequenceSet -- A set of test sequences.
                  curTC -- A test case that is being examined.
                  t.CoveredTimes -- The number of times t has been covered.

OrderTestCases (TestCaseGraph)
BEGIN -- Algorithm OrderTestCases
    FOR EACH test case t in TestCaseGraph
        UncoveredTCs = UncoveredTCs ∪ {t}
        get PrevTCs(t) and NextTCs(t)
    END FOR
    -- Build one test sequence
    WHILE (UncoveredTCs IS NOT EMPTY) LOOP
        testSequence = EMPTY
        curTC = t such that t is an initial node in TestCaseGraph ⋀
            (t.CoveredTimes ≤ t_n.CoveredTimes ∀ t_n such that PrevTCs(t_n) = ∅)
        LOOP -- Finds one test case sequence
            testSequence = testSequence ‖ curTC -- concatenate to end
            UncoveredTCs = UncoveredTCs - {curTC}
            IF (∃ t ∈ NextTCs(curTC) such that t ∈ UncoveredTCs) THEN
                curTC = t
            ELSE IF (∃ t ∈ NextTCs(curTC) such that ∃ t_n ∈ NextTCs(t) ⋀ t_n ∈ UncoveredTCs) ⋀
                (¬ ∃ t_p ∈ PrevTCs(t_n) such that t_p ∈ UncoveredTCs) THEN
                testSequence = testSequence ‖ t
                UncoveredTCs = UncoveredTCs - {t}
                curTC = t_n
            ELSE
                IF (UncoveredTCs IS NOT CHANGED) THEN
                    RETURN (NotFinished)
                END IF
                testSequenceSet = testSequenceSet ∪ {testSequence}
                EXIT
            END IF
        END LOOP -- Build one test sequence

    END LOOP -- All test cases covered
    RETURN (Finished)
END Algorithm OrderTestCases
```

Figure 6: **The OrderTestCases Algorithm, Step I**

This algorithm terminates only when all test cases are covered; in the worst case, each uncovered test case will appear in a unique test sequence.

The final algorithm is CalculateWeightDistance. It uses a breadth-first search technique to find the distance of the shortest path from an initial node to each node in the graph. It also calculates a weight for each node, which is the number of uncovered test cases along the shortest path. It is a simple modification of Dijkstra's well known shortest path algorithm.

4.2 Test Cases Order Examples

These algorithms are best understood through examples. The first example illustrates a simple case where Step I is sufficient to cover all test cases. The second example is slightly more complicated. The third example illustrates a simple case of Step II. The fourth example uses the common cruise control specification, illustrating a somewhat more complicated case that uses Step II.

Example 1. The first example uses specification graph 1, in Figure 9. The nodes represent states in the system, and the edges are annotated with test cases that will cause the state transition associated with the edge. For example, test case 1 will cause the system to transition from state B to state A. The associated test case graph is shown in Figure 10. There are two initial test cases, TC1 and TC3. The algorithm starts with test case TC1, which has one successor test case, TC2. TC1 is put into the test sequence, then TC2 is added. From TC2, the algorithm finds TC4, and from TC4 it finds TC6. Since TC6 has no next test case, the first test sequence is completed, and is: (TC1, TC2, TC4, TC6). For the next sequence, the algorithm starts with the other initial test case, TC3. From TC3, the algorithm first visits TC4. TC4 is already covered, but the algorithm still checks its next test cases in case TC4 is the only node to reach its successors, and there are uncovered test cases among them. In this example, TC4's only next test case TC6 is already covered, so the algorithm will not include TC4. Instead, TC2's other next test case, TC5, is visited. TC5 is not covered, so it is included in the sequence. Next, the algorithm visits TC5's next test case, which is TC6. TC6 is already covered and has no next test case, so it is not visited. Since TC5 has no other next test case, the second sequence is completed: (TC3, TC5).

The complete set of test sequences for this example is:

1. (TC1, TC2, TC4, TC6)
2. (TC3, TC5)

Example 2. As another example, consider the five-state specification graph in Figure 11. Its test case graph is given in Figure 12. This TCG has five initial test cases. The algorithm will start with test case TC1, then goes to its first next test case TC9. From TC9, the algorithm goes to TC7, then to TC6. From TC6, the algorithm first looks at TC9, but TC9 is already covered. So the algorithm checks its next test cases TC7 and TC8. Since TC7 is already covered, and TC8 has uncovered predecessor test cases, the algorithm bypasses TC9 and visits TC10. TC10 is uncovered, so it is included in the current test sequence, and the algorithm goes to TC8. From there, it checks TC6 again. TC6 still has uncovered next test cases, but they have uncovered previous test cases. So the first test sequence is completed, and is (TC1, TC9, TC7, TC6, TC10, TC8).

For the next sequence, the algorithm starts with TC2, and first visits TC11. Since TC11 has no uncovered next test cases, the second test sequence is simply (TC2, TC11). Likewise, the third test sequence is simply (TC3, TC12). The fourth test sequence starts at node TC4. Its only next test case is TC6, which is already covered. But, since TC6 has an uncovered next test case (TC13), TC6 is included in this test sequence, followed by TC13, giving the test sequence (TC4, TC6, TC13). The final test sequence is simply TC5 by itself.

The complete set of test sequences for this example is below. TC6 is used two times, so there is one redundant test case.

1. (TC1, TC9, TC7, TC6, TC10, TC8)

```
algorithm:        FinishOrderTestCases (TestCaseGraph)
input:            A graph that indicates ordering relationships among test cases.
output:           Test Sequence Set.
output criteria:  Fewest number of sequences with fewest number of test cases.
declare:          Q -- A queue for test cases starting from initial test cases.
                  t.Distance -- Number of test cases on the shortest path from an initial
                                test case to test case t.
                  t.Weight -- Number of uncovered test cases on the shortest path from an
                              initial test case to test case t.

FinishOrderTestCases (TestCaseGraph)
BEGIN -- Algorithm FinishOrderTestCases
    CalculateWeightDistance (TestCaseGraph)
    WHILE (UncoveredTCs IS NOT EMPTY) LOOP
        testSequence = EMPTY
        curTC = t such that t ∈ UncoveredTCs ⋀ t.Weight ≥ t_p.Weight ∀ t_p ∈ UncoveredTCs
        LOOP
            testSequence = curTC ‖ testSequence
            UncoveredTCs = UncoveredTCs - {curTC}
            IF (curTC.Distance = 0) THEN -- Initial test case
                testSequenceSet = testSequenceSet ∪ {testSequence}
                EXIT -- next iteration of loop
            END IF
            -- Find the predecessor with the maximum weight.
            IF (∃ t_p ∈ PrevTCs(curTC) such that t_p.Weight ≠ 0 ⋀
                    t_p.Weight ≥ t.Weight ∀ t ∈ PrevTCs(curTC)) THEN
                t_p.Weight = t_p.Weight - 1
                -- Weight changed - refresh the TestCaseGraph
                CalculateWeightDistance (TestCaseGraph)
                curTC = t_p
            ELSE
                curTC = t_p such that t_p ∈ PrevTCs(curTC) ⋀
                    (t_p.Distance ≤ t.Distance ∀ t ∈ PrevTCs(curTC))
            END IF
        END LOOP
    END LOOP WHILE
END Algorithm FinishOrderTestCases
```

Figure 7: **The OrderTestCases Algorithm, Step II**

```
algorithm:      CalculateWeightDistance (TestCaseGraph)
input:          A graph that indicates ordering relationships among test cases.
output:         Test case graph with weights and distances calculated for all test cases.
declare:        Q -- A queue for test cases starting from initial test cases.
                t.Distance -- Number of test cases on the shortest path from an initial
                              test case to test case t.
                t.Weight -- Number of uncovered test cases on the shortest path from an
                            initial test case to test case t.
                t.Reached -- Whether the test case has been reached or not.

CalculateWeightDistance (TestCaseGraph)
BEGIN -- Algorithm CalculateWeigtDistance
    -- Find distance and weight of each test case
    FOR EACH t such that t in TestCaseGraph ⋀ PrevTCs(t) = ∅
        t.Distance = 0
        t.Weight = 0
        EnQue (Q, t)
    END FOR
    WHILE (NotEmpty (Q)) LOOP
        curTC = DeQue (Q)
        FOR EACH test case t_p such that t_p ∈ NextTCs(curTC) ⋀ t_p.Reached = False
            IF t_p NOT IN Q THEN
                t_p.Distance = curTC.Distance + 1
                IF t_p ∈ UncoveredTCs THEN
                    t_p.Weight = curTC.Weight + 1
                ELSE
                    t_p.Weight = curTC.Weight
                END IF
                t_p.Reached = True
                EnQue (Q,t_p)
            END IF
        END FOR
    END LOOP
END Algorithm CalculateWeightDistance
```

Figure 8: **The Weight and Distance Algorithm**

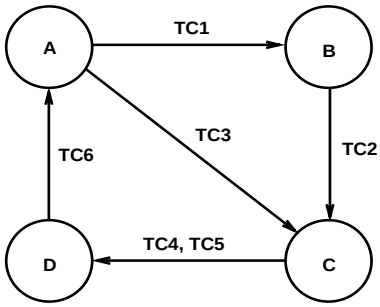

Figure 9: **Example 1: Specification Graph 1**

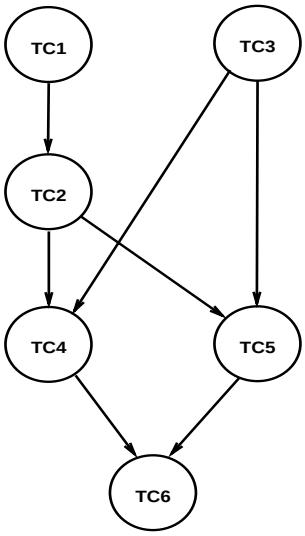

Figure 10: **Example 1: Test Case Graph 1**

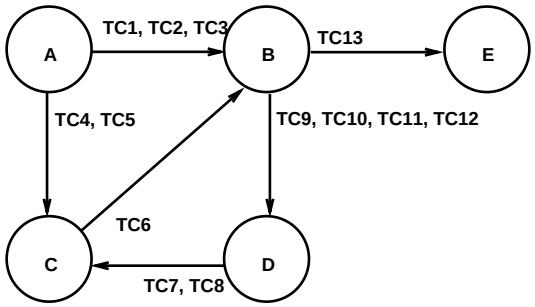

Figure 11: **Example 2: Specification Graph 2**

2. (TC2, TC11)
3. (TC3, TC12)
4. (TC4, TC6, TC13)
5. (TC5)

Example 3. Consider the specification graph in Figure 13 and its test case graph in Figure 14. The algorithm starts with TC1, then goes to its first next test case, TC2. Its next test case is TC4, TC4's next test case is TC6, and TC6's next test case is TC8. TC8 has no next test case, so the complete first sequence is (TC1, TC2, TC4, TC6, TC8). Next, the algorithm starts with TC1 again, then first examines TC2. TC2 is covered, and its next test case TC4 is also covered, so TC2 is not included. TC3 is another next test case for TC1, and it is not covered, so TC3 is included in the test sequence. TC3's only uncovered next test case is TC5, which is uncovered, so the second test sequence is (TC1, TC3, TC5).

TC7 is still uncovered, so the algorithm starts with TC1 again, but it cannot reach TC7. Because this does not change the `UncoveredTCs` set, the Step II algorithm, FinishOrderTestCases, is used. First the weight and distance of each test case is calculated, as shown in Figure 15.

The algorithm FinishOrderTestCases starts from TC7. TC7 only has one predecessor, so the final test sequence is: (TC1, TC2, TC4, TC7).

The complete set of test sequences for this example is below. TC1 is used three times, TC2 twice, and TC4 twice. So there is a total of four redundant test cases.

1. (TC1, TC2, TC4, TC6, TC8)
2. (TC1, TC3, TC5)
3. (TC1, TC2, TC4, TC7)

Example 4. The fourth example is the cruise control specification graph, shown in Figure 16. Its test case graph is shown in Figure 17. The algorithm starts with test case TC1 in the first test sequence, then visits TC1's first next test case TC2. TC2's only next test case is TC1, which is already covered. Although TC1 has an uncovered next test (TC3), TC3 has uncovered predecessors, so it is not visited. Thus, the first sequence is (TC1, TC2).

TC1 is the only initial test case, so the second sequence also starts with TC1. Node TC2 is already covered and has no uncovered successors, so the algorithm proceeds to TC3 and appends it to the sequence. The algorithm next inspects TC5, which is uncovered, so it is appended to the sequence. The only successor to TC5 is TC3, which is already covered, and although TC3 has several uncovered successors (TC6, TC7, TC8, and TC4), they all have uncovered predecessors. So the second sequence is completed as (TC1, TC3, TC5).

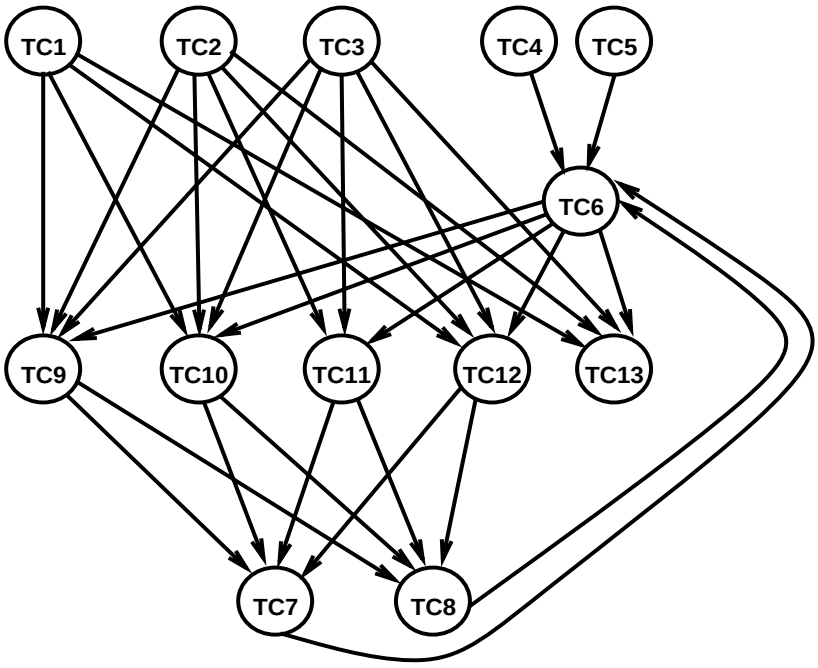

Figure 12: **Example 2: Test Case Graph 2**

On the next iteration through the loop, both successors to the initial node TC1 (TC2 and TC3) are already covered. All of the uncovered successors to TC2 and TC3 (TC6, TC7, TC8, and TC4) have uncovered predecessors, so the OrderTestCases algorithm, Step I algorithm terminates without covering all nodes.

FinishOrderTestCases is called next, and first gets the weight and distance of each test case in the graph, using the CalculateWeightDistance algorithm. The test case graph is repeated in Figure 18, with the weights and distances shown beside each node. The uncovered test cases are shown with dashed lines.

FinishOrderTestCases starts with a node that has the highest weight, and builds a test sequence from the end to the beginning. It chooses arbitrarily from TC9, TC10, TC11, and TC12; assume TC12 is chosen. TC12's predecessors are TC7 and TC8, both of which have weights of 1. Assume TC7 is chosen. Its predecessors are TC3, TC11, and TC12. TC3 and TC12 are covered, so TC11 is taken. Its predecessors are TC7 and TC8, TC7 is covered, so TC8 is chosen. Now all of TC8's predecessors (TC3, TC11, and TC12) are covered, so the node with the shortest distance is chosen (TC3). Its predecessors are TC1, TC5, TC6 and TC10. TC1 and TC5 are covered, and TC6 and TC10 have the same weight of 1, so one is chosen arbitrarily (assume TC6). As with TC8, all of TC6's predecessors (TC3, TC11, and TC12) are covered, so TC3 is chosen. Its only remaining uncovered predecessor is TC10, so it is added to the sequence. Both of TC10's predecessors (TC7 and TC8) are covered, so TC7 is chosen. All of TC7's predecessors (TC3, TC11, and TC12) are covered, so the node with the shortest distance (TC3) is chosen. All of its predecessors are covered, so TC1 is chosen as the initial test case in this sequence. The complete sequence is (TC1, TC3, TC7, TC10, TC3, TC6, TC3, TC8, TC11, TC7, TC12).

The only remaining uncovered test cases are TC4 and TC9. For both of them, the shortest

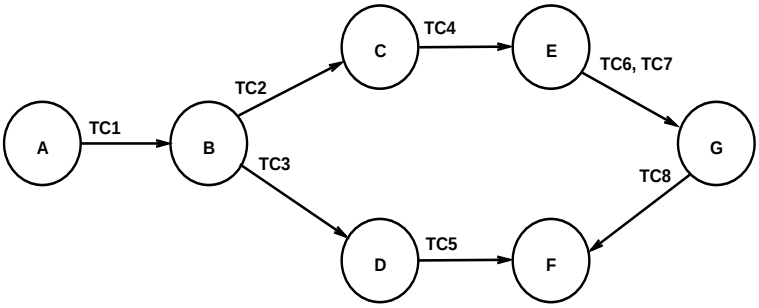

Figure 13: **Example 3: Specification Graph 3**

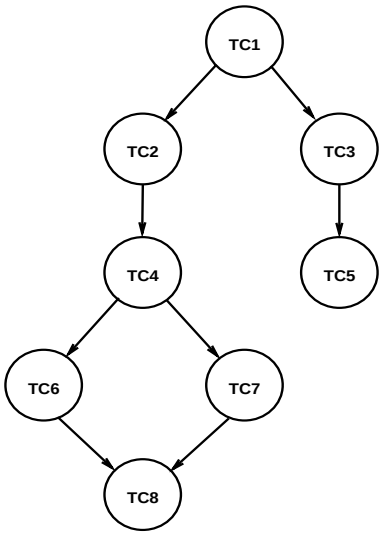

Figure 14: **Example 3: Test Case Graph 3**

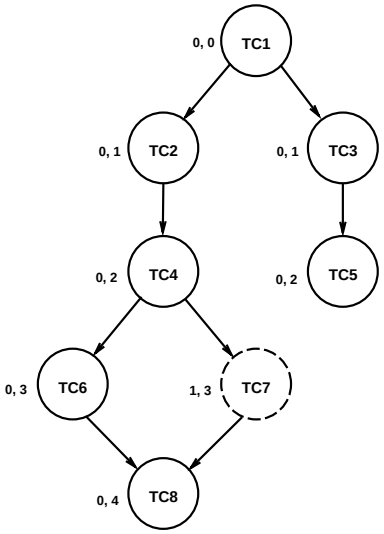

Figure 15: **Example 3: Test Case Graph 3 With Weights**

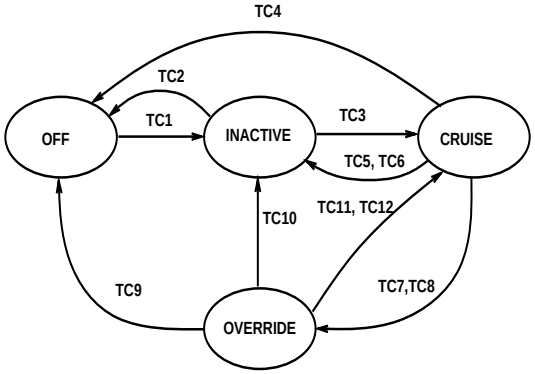

Figure 16: **Example 4: Specification Graph 4**

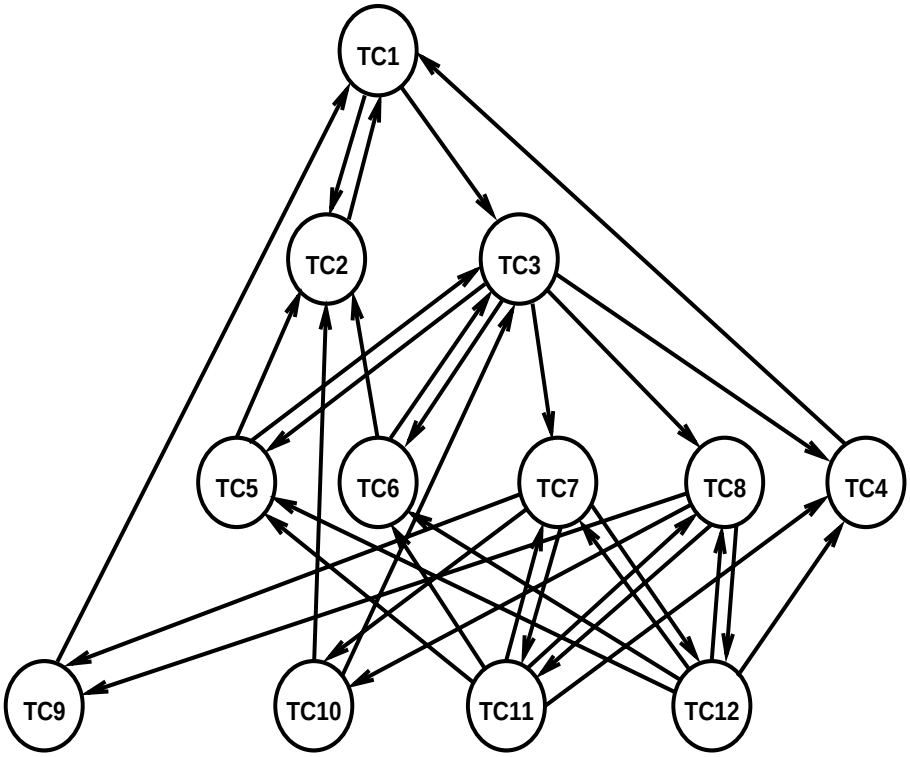

Figure 17: **Example 4: Test Case Graph 4**

path (through covered test cases) is chosen, yielding (TC1, TC3, TC7, TC9) and (TC1, TC4).

The complete set of test sequences for this example is below. TC1 is used five times, TC3 is used five times, and TC7 is used three times. So there is a total of 10 redundant test cases.

1. (TC1, TC2)
2. (TC1, TC3, TC5)
3. (TC1, TC3, TC7, TC10, TC3, TC6, TC3, TC8, TC11, TC7, TC12)
4. (TC1, TC3, TC7, TC9)
5. (TC1, TC4)

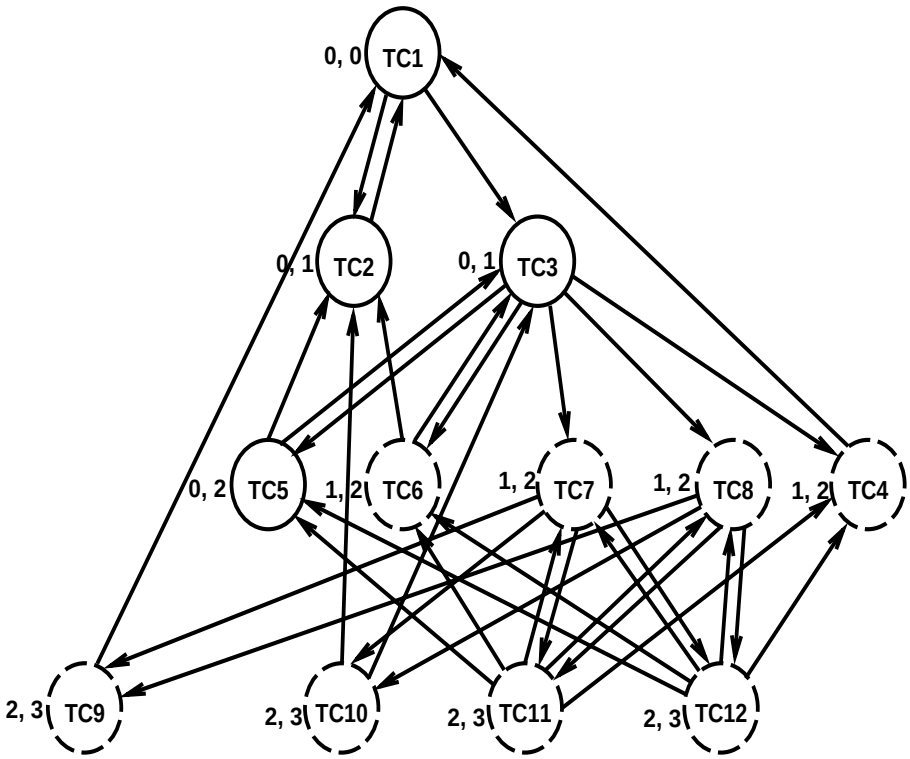

Figure 18: **Example 4: Test Case Graph 4 With Weights**

5 TEST PREFIX GENERATION

The prefix for a test case includes all inputs necessary to reach the pre-state and to give the triggering event variable its before-value.

Prefixes can be generated automatically from the *specification graph*. In the Phase I final report [Off98], the question of whether this problem is generally solvable was left open. The problem of determining prefixes is related to the reachability problem in software analysis, which is generally unsolvable. This Phase II report, however, presents an algorithm that will generate test case prefix specifications, thus showing that the problem is solvable. This problem is solvable for state-based specifications because of the finite deterministic nature of state-based systems.

To generate prefix inputs, the specification graph introduced previously is used to generate prefix inputs to find all paths from the initial state to a *target* state. The target state is the pre-state for the test case. The algorithm uses a depth-first search and is shown in Figure 19.

The specification graph is provided to the algorithm in an adjacency list structure, and returns all possible paths from the initial state to the target state. The paths are returned as lists of predicates that must be satisfied to traverse from the initial state to the target state. These predicates are then resolved into actual input values. There may be more than one path from the initial state to the target state. In this case, all paths are returned, and the enclosing program can choose which one to use. This choice can be made randomly, by choosing the first path, the shortest, or the longest, depending on the needs of the test engineer.

As an example, consider the simple four-state specification graph in Figure 20. Its adjacency list representation is shown in Figure 21. Suppose the target node is S4. The algorithm will start in node S1, then go to its first adjacent node S2. From S2, the algorithm first checks S1, but since S1 has already been visited, the algorithm backtracks and goes to the next node that is adjacent to S2, which is S4, the target node. So the first path discovered is (S1 ; S2 ; S4). Then, the algorithm backtracks to node S1, and visits the next adjacent node, S3. The first adjacent node to S3 is S1, which has already been visited, so the algorithm goes to S4, resulting in the path (S1 ; S3 ; S4). Finally, the algorithm backtracks to node S1, and visits its last adjacent node, S4. This results in the path (S1 ; S4).

The complete set of paths is: {(S1 ; S2 ; S4), (S1 ; S3 ; S4), (S1 ; S4)}. The corresponding predicates are: {($A \bigwedge B$; $B \bigwedge C$), ($A \bigwedge \overline{B}$; $\overline{C}$), ($\overline{A} \bigwedge B$)}.

5.1 Prefix Generation Example

This subsection presents an example of creating test prefixes for a test case developed from the cruise control system specification. The specifications for the system were shown in Table 1, and its specification graph was shown in Figure 1. Figure 22 shows the adjacency list for the cruise control graph.

The test prefixes to reach the pre-state will be found with the algorithm. The test case we choose tests the transition from state OVERRIDE to CRUISE. The pre-state from this test is OVERRIDE and the post-state is CRUISE. The test case without the prefix inputs is:

Prefix:	????		– Reach `OVERRIDE` state
Test case value:	*Resume*	= `False`	– Trigger before-value
	Ignited	= `True`	– Condition variable
	Running	= `True`	– Condition variable
	Brake	= `False`	– Condition variable
	Resume	= `True`	– Triggering event
Expected output:	`CRUISE`		

```
algorithm:   FindSequenceOfTransitions (Size, Target)
input:       A transition digraph represented by an adjacency list structure,
             and a destination node Target.
output:      The sequence set of transitions ST (Target) includes all possible paths
             from the initial state S1 to the target state St.
comments:    There are a total of Size states.  Without loss of generality, let S1
             be the initial state, and St target state.  The algorithm
             uses the following adjacency list structure:

            TYPE VertexType IS 1..Size
            TYPE NodePtr IS ACCESS StateNode
            TYPE StateNode IS RECORD
                 Vertex  :  VertexType
                 Link    :  NodePtr
            END RECORD
            TYPE HeaderArray IS ARRAY (1..Size) OF NodePtr

    StateList :  HeaderArray
    Ptr       :  NodePointer
    V         :  VertexType
    Path      :  ARRAY (VertexType) OF Integer
    ST        :  Sequence Set of Transitions

    -- Recursive function to implement Depth First Search.
    Travel (Targ, Current)
    BEGIN -- Travel
        -- Loop found, already visited this node.
        IF (Current IN Path) RETURN
        -- Find the target.
        -- Path represents one of the existing paths from S1 to St.
        IF (Current = Targ)
            ST (Targ) := ST (Targ) ∪ Path
            RETURN
        END IF
        Add Current to Path

        FOR EACH node S linked to Current
            Add S to Path
            Travel (Targ, S) -- Recursion
            Delete S from Path
        END FOR
    END Travel
```

Figure 19: **The FindSequenceOfTransitions Algorithm**

```
BEGIN -- FindSequenceOfTransitions
    ST (Target) := EMPTY
    Path := EMPTY

    -- Check if no path starts from the initial node
    Ptr := StateList (1)
    IF (Ptr.Link = NULL) RETURN
    -- Check there is a path to the target node
    FOR EACH node N in StateList
        FOR EACH node M adjacent to N
            IF (Target = M)
                Found := Target
            END IF
        END FOR
    END FOR
    IF (NOT Found) RETURN

    Ptr := StateList (1)
    Add 1 to Path -- The initial state
    WHILE (Ptr ≠ NULL) LOOP
        Travel (Target, Ptr.Vertex)
        Ptr := Ptr.Link
    END LOOP
END FindSequenceOfTransitions
```

Figure 19: **The FindSequenceOfTransitions Algorithm - continued**

The derivation process of the prefixes to reach the OVERRIDE state is as follows. There are four states in this system. OFF is the initial state, and OVERRIDE is the target state for the prefix generation algorithm. In this depth-first search, the four vertices are visited in order of increasing distance from the initial state.

First, vertex 2 (INACTIVE) is visited, since it is adjacent to 1 (OFF). The first vertex adjacent to 2 is 1, but it has already been visited. So the search next visits vertex 3 (CRUISE). The algorithm proceeds until the target 4 (OVERRIDE) is visited, which means a path to OVERRIDE has been found. At this point, the algorithm looks for other paths to reach the target by returning to previously visited vertices.

In this example, there is only one non-looping path to reach the OVERRIDE state, (1 ; 2 ; 3 ; 4). So the prefix specifications to OVERRIDE are: $\{P1\ ;\ P3\ ;\ (P7\ \vee\ P8)\}$. A completed test case is therefore:

Prefix:	*Ignited*	= `True`	– Reach INACTIVE state
	Activate	= `True`	
	Running	= `True`	
	Brake	= `False`	– Reach CRUISE state
	Deactivate	= `True`	
	Toofast	= `False`	– Reach OVERRIDE state
Test case value:	*Resume*	= `False`	– Trigger before-value
	Ignited	= `True`	– Condition variable
	Running	= `True`	– Condition variable

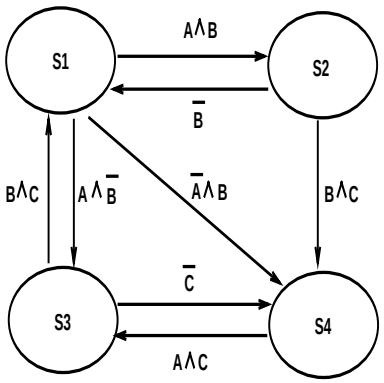

Figure 20: **Example Specification Graph**

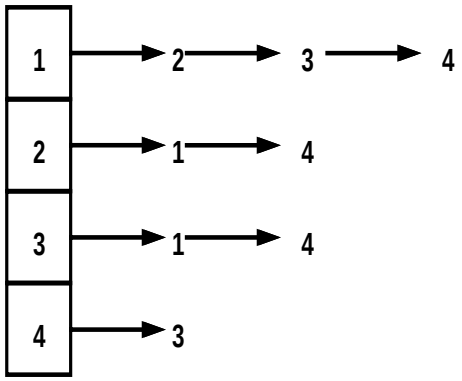

Figure 21: **Example Adjacency List**

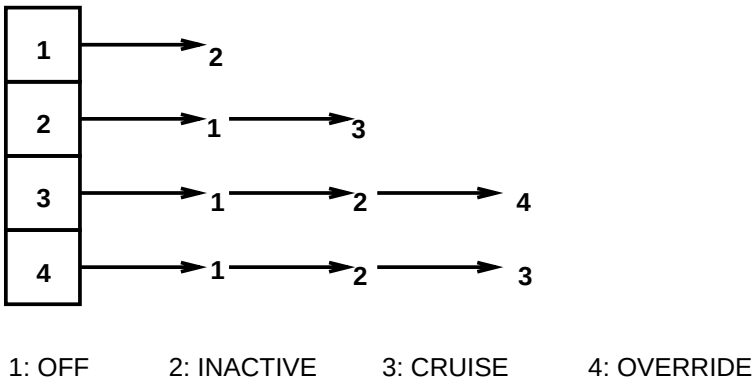

Figure 22: **Adjacency List Structure for Cruise Control Graph**

	Brake = `False`	– Condition variable
	Resume = `True`	– Triggering event
Expected output:	`CRUISE`	

6 REMOVING REDUNDANT TEST CASE VALUES

When a number of tests are created automatically, it is natural for some to be redundant. Different test requirements and specifications will lead to similar or the same tests. This leads to overlap and extra expense when running the tests, evaluating the results from the tests, storing the tests, and re-running the tests. As part of this work, analysis techniques have been developed to remove redundancy so as to reduce the amount of effort that is required.

This also brings up a related issue that is of importance to practical execution of tests. When a number of tests are available, they can be executed in one of two ways. Either each test can be executed separately, or a number of tests can be combined to have fewer test executions. Both ways have advantages and disadvantages.

When combining tests, there will be fewer tests to run, but each one will be larger. Because of the overhead associated with setting up to run each test, the overall execution time will be less when combining tests. On the other hand, it becomes harder to change the tests, and when the software fails, big tests makes it harder for the debugger to track down the fault.

When separating tests, there will be more tests to run, but each one will be smaller. This will result in more overall execution time. The advantages are that it is easier to change small tests, and debuggers will need less effort to track down faults.

A compromise approach is to store and manipulate tests individually, then use automatic techniques to combine them during execution. This gives the test maintenance advantages of small tests, and the execution time advantage of combined tests.

The algorithm in this section attempts to reduce the amount of redundancy in the test sets. This will also reduce the expense of running tests, as well as save effort when storing and modifying tests. There are two kinds of redundancies that can be addressed, external and internal. Each test case generated by our criteria is composed of a number of variable assignments, which can be modeled as simple `(variable:value)` pairs. External redundancy refers to redundancy between test cases; a test case is *externally redundant* if it is exactly the same as another test case. Internal redundancy refers to redundancy inside test cases among the `(variable:value)` pairs; a `(variable:value)` assignment is *internally redundant* if the variable already has the value when the assignment is reached.

Because of the way test requirements and test cases are created under these criteria, there are no externally redundant test cases. On the other hand, there can be many internally redundant test case value assignments. The algorithm in this section searches through each `(variable:value)` pair in each test case, and removes all redundant assignments.

The algorithm is shown in Figure 23. This is a simple algorithm. For each test case value (`CurValue`) in a test case (`CurTC`), the subsequent test case values are searched. If a subsequent test case value assigns the same value to the variable in `CurValue` (that is, it is redundant), then the second assignment is removed. If a subsequent test case value assigns a different value to the variable, then the first assignment can be considered "dead", so the algorithm quits looking for subsequent redundant test case values.

As an example, consider the simple test case below, which has boolean variables `A, B, C, D,` and `E`:

```
A = True
B = True
A = False
B = True
C = False
D = False
A = True
E = True
```

```
algorithm:      RemoveRedundantTestCaseValues (TestCaseList)
input:          A list of complete test case scripts with prefixes and test case values.
output:         Test Cases List without redundant test case values.
declare:        TestCaseList -- Array of test case scripts.  Each item represents a complete test case.
                CurTC -- The test case script being processed.
                CurValue -- The test case value being processed.
                NextValue -- The test case value to compare with CurValue.
                NumTCs -- The number of test cases in TestCaseList.

                TYPE TestCaseValue IS RECORD
                     Name  :  String
                     Value :  Boolean
                     Next  :  TestCaseValue
                END RECORD
                TYPE TestCaseScript IS LIST OF TestCaseValue
                TYPE TestCaseListArray IS ARRAY (1..NumTCs) OF TestCaseScripts

                TestCaseList         :  TestCaseListArray
                CurValue, NextValue  :  TestCaseValue
                CurTC                :  TestCaseScript

RemoveRedundantTestCaseValues (TestCaseList)
BEGIN -- Algorithm RemoveRedundantTestCaseValues
    FOR (i = 1 TO NumTCs) LOOP
        CurTC = TestCaseList (i)
        FOR EACH CurValue in CurTC
            NextValue = CurValue.next
            WHILE (NextValue Exists) LOOP
                IF (CurValue.Name = NextValue.Name) THEN
                    IF (CurValue.Value = NextValue.Value) THEN
                        REMOVE node NextValue from CurTC
                    ELSE
                        -- The name is the same, but a different value,
                        -- so the value is changing.  No need to look further.
                        EXIT LOOP
                    END IF
                END IF
                NextValue = NextValue.Next
            END WHILE LOOP
        END FOR EACH
    END FOR LOOP
END Algorithm RemoveRedundantTestCaseValues
```

Figure 23: **The RemoveRedundantTestCaseValues Algorithm**

A	B	A	B	C	D	A	E
t	t	f	t	f	f	t	t

Figure 24: **Example : Original Test Case Script**

A	B	A	C	D	A	E
t	t	f	f	f	t	t

Figure 25: **Example : Processed Test Case Script**

These test case values constitutes the test case value list shown in Figure 24. The algorithm takes the first test case value, which contains the pair `(A:t)`, and compares it with the rest of the values in the list. The third test case value has the same name, but a different value assignment, so it is not redundant. Instead, the different assignment indicates that the algorithm should not look further for a test case value that is redundant. Then the algorithm considers the next test case value, which contains the pair `(B:t)`. The fourth test case value has the same variable name and value, so the algorithm removes it from the list. The algorithm continues to check other test case values until it reaches the end of the list. After the algorithm terminates, the resulting list of test case values is as shown in Figure 25.

7 INTERACTION-PAIR TESTING

In the Phase I, 1997 report [Off98], transition-pair coverage was introduced to check the interfaces among states. The attempt was to require that certain sequences of states be entered. The formal definition was:

Transition-pair coverage level:	**For each state S, form test requirements such that for each incoming transition and each outgoing transition, both transitions must be taken sequentially.**

In this research, the notion is generalized to that of testing sequence-pairs. A sequence-pair is a pair of states that can be entered in sequence. The concept of testing sequence-pairs is based on the observation that some pairs of states have interactions that should be carefully tested. When states have some interaction, there is a high potential for failures that will not be found unless both states are reached under the same execution. In particular, a new test criterion is proposed that is based on testing **interaction-pair**s, where an interaction-pair is a pair of states that have some data or control interaction.

The interaction addressed in sequence-pair testing is that of control-flow; a pair of states interact if there are direct transitions between them. Three interactions, primarily through data, are identified and defined here. The three interaction types are shared data items on transitions into the states, shared data items on transitions out of the states, and state invariants. Pairs of states that interact will be tested together to ensure that the specifications are defined correctly and that the software is implemented correctly.

Interaction pair testing is defined as follows:

Interaction-pair coverage:	**For each state S_1 and S_2 such that the incoming transition predicates for S_1 and S_2 are the same, the outgoing transition predicates for S_1 and S_2 are the same, or S_1 and S_2 have the same state invariant, form test requirements such that there is an execution path from S_1 to S_2 or from S_2 to S_1.**

Consider the example specification graphs in Figure 26. Figure 26.I shows the case where two states have the same precondition. States `S2` and `S4` both have the precondition $\mathbf{B} \wedge \mathbf{D}$. To test this interaction, an execution path is needed that includes both states. Thus, the test requirement for interaction-pair (S2:S4) is S4:S3:S2 $\Longrightarrow \mathbf{B} \wedge \mathbf{C} : \mathbf{B} \wedge \mathbf{D}$.

Figure 26.II shows the case where two states have the same postcondition. States `S2` and `S4` both have the postcondition $\mathbf{B} \wedge \mathbf{C}$. To test this interaction, an execution path is needed that includes both states. Thus, the test requirement for interaction-pair (S2:S4) is S4:S3:S2 $\Longrightarrow \mathbf{B} \wedge \mathbf{C} : \mathbf{A} \wedge \mathbf{C}$.

Figure 26.III shows the case where two states have the same state invariant. A state invariant encodes predicates that are always true in a state, no matter from where it was reached. State `S2` has the preconditions $\mathbf{A} \wedge \mathbf{B}$ and $\mathbf{B} \wedge \mathbf{D}$, and `S4` has the preconditions $\mathbf{B} \wedge \mathbf{D}$ and $\mathbf{B} \wedge \mathbf{C}$. For both states, $\mathbf{B}$ is always true when the state is entered. To test this interaction, an execution path is needed that includes both states. Thus, the test requirement for interaction-pair (S2:S4) is S2:S3:S4 $\Longrightarrow \mathbf{A} \wedge \mathbf{C} : \mathbf{B} \wedge \mathbf{C}$.

Finally, Figure 27 shows a case where there is an interaction between two states, but no execution path between them. States `S2` and `S4` both have the same postcondition, $\mathbf{B} \wedge \mathbf{D}$, but there is no

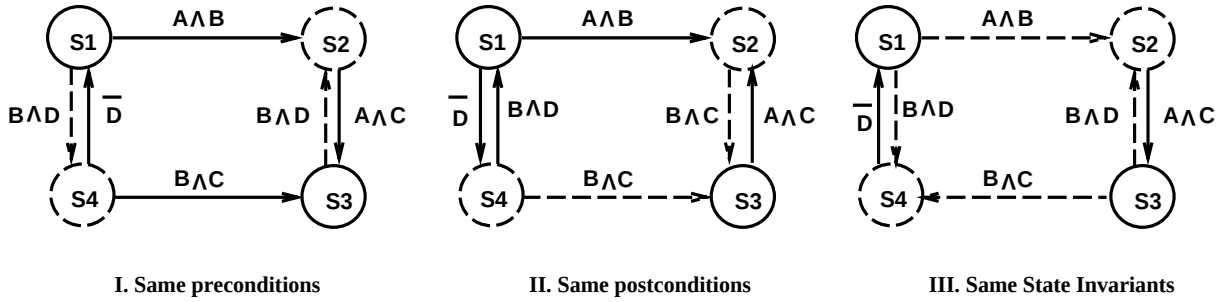

Figure 26: **Example : Specification Graphs for Interaction-Pairs**

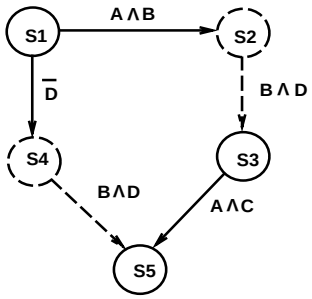

Figure 27: **Example : Specification Graph for Interaction-Pair with no Execution Path**

path from `S2` to `S4` and no path from `S4` to `S2`, so no test requirement is generated.

8 CONCLUSIONS

This report presents results and strategies for practically applying test cases generated according to the criteria presented previously. This research project addresses the problem of developing formalizable, measurable criteria for generating test cases from specifications, and applying those criteria in practical, industrial situations. Algorithms for addressing the prefix problem in specification-based testing were presented. This result provides a solution to a major problem within specification-based testing. The fact that the solution is algorithmic and complete points out a significant advantage of generating tests from specifications instead of from code – the complementary problem in code-based test data generation is unsolvable. Also presented were algorithms to remove redundant test case values and to apply the idea of "interaction-pair" testing.

The immediate goal of this research was to develop mechanisms for practically applying the test generation criteria from the previous year. Short term goals are to develop *mechanical* procedures to derive test cases from formal specifications, and apply the method to industrial software specifications. Future goals are to carry out an empirical evaluation of the testing method, and to adapt the testing model and criteria to the Unified Modeling Language (UML). An eventual goal is to build an automatic test data generation tool for this technique.

References

[Atl94] J. M. Atlee. Native model-checking of SCR requirements. In *Fourth International SCR Workshop*, November 1994.

[HL92] J. R. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10, New Orleans LA, May 1992.

[Jin96] Zhenyi Jin. Deriving mode invariants from SCR specifications. In *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems*, pages 514–521, Montreal, Canada, October 1996. IEEE Computer Society.

[Off98] A. J. Offutt. Generating test data from requirements/specifications: Phase i final report. Technical report ISSE-TR-98-01, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, April 1998.

Contents