Generating Test Cases from UML Specifications

by

Aynur Abdurazik and Jeff Offutt

ISE-TR-99-09

May, 1999

Information and Software Engineering
George Mason University
Fairfax, Virginia 22030

TABLE OF CONTENTS

Contents

LIST OF TABLES

List of Tables

LIST OF FIGURES

List of Figures

ABSTRACT

GENERATING TEST CASES FROM UML SPECIFICATIONS

Aynur Abdurazik, M.S.

George Mason University, 1999

Thesis Director: Dr. A. Jefferson Offutt

Unified Modeling Language (UML) is a third generation modeling language in object-oriented software engineering. It provides constructs to specify, construct, visualize, and document artifacts of software- intensive systems.

This paper presents a technique that uses Offutt's state-based specification test data generation model to generate test cases from UML statecharts. A tool TCGen has been developed to demonstrate this technique with specifications written in Software Cost Reduction (SCR) method and Unified Modeling Language (UML). Experimental results from using this tool are presented.

1 INTRODUCTION

Unified Modeling Language (UML) is a third-generation method for specifying, visualizing, and documenting artifacts of object-oriented systems [BRJ98]. UML unifies the Booch [Boo94], Objectory [Jac92], and OMT [RBP$^+$91] methods, and incorporates ideas from a number of other methodologists. One of UML's goals is to provide the basis for a common, stable, and expressive object-oriented development method. However, UML still cannot help to develop fault free software. Software developed with UML has to be tested to assure its quality, and also to prevent faults. This thesis provides a model for generating test cases from state charts in UML specifications.

1.1 Software Testing

Software testing typically ranges from 40% to 70% of the development effort [Bei90]. Beizer defines six levels at which software testing occurs, *unit test*, *module test*, *integration test*, *subsystem test*, *system test* and *acceptance test*. There are two general testing approaches, *black box* and *white box*. *Black box* testing approaches create test data without using any knowledge of the structure of the software under test, whereas *white box* testing approaches explicitly use the program structure to develop test data. Black box testing is usually based on the requirements and specifications, while white box testing is usually based on the source code. White box testing approaches are typically applied during unit test, and black box testing approaches are typically applied during integration test and system test.

Software testing includes executing a program on a set of test cases and comparing the actual results with the expected results. Testing and test design, as parts of quality assurance, should also focus on fault prevention. To the extent that testing and test design do not pevent faults, they should be able to discover symptoms caused by faults. Finally, tests should provide clear diagnoses so that faults can be easily corrected [Bei90].

A central question of software testing is "what is a test criterion?" A *test criterion* is a rule or collection of rules that impose requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of requirements that are satisfied [Off98b].

There are various ways to classify adequacy criteria. One of the most common is by the source of information used to specify testing requirements and in the measurement of test adequacy. Hence, an adequacy criterion can be specification-based or program-based.

A *specification-based* criterion specifies the required testing in terms of identified features of the specifications of the software, so that a test set is adequate if all the identified features have been fully exercised. Here the specifications are used to produce test cases, as well as to produce the program. An abstract view of the specification-based testing process is shown in Figure 1.

A *program-based* criterion specifies testing requirements in terms of the program under test and decides if a test set is adequate according to whether the program has been throughly exercised. An abstract view of the program-based testing process is shown in Figure 2.

It should not be forgotten that for both specification-based and program-based testing, the correctness of program outputs must be checked against the specification or requirements. However, in both cases, the measurement of test adequacy does not depend on the results of this checking. Also, the definition of specification-based criteria given previously does not presume the existence of a formal specification.

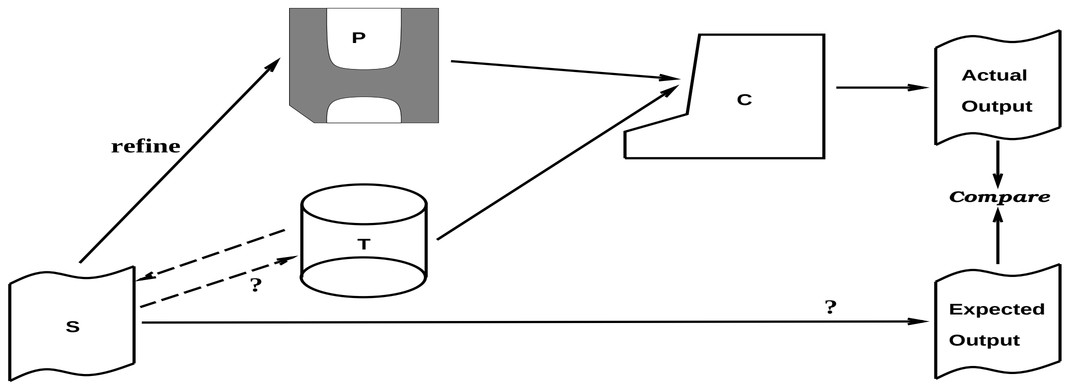


Figure 1: **Specification-based Testing**

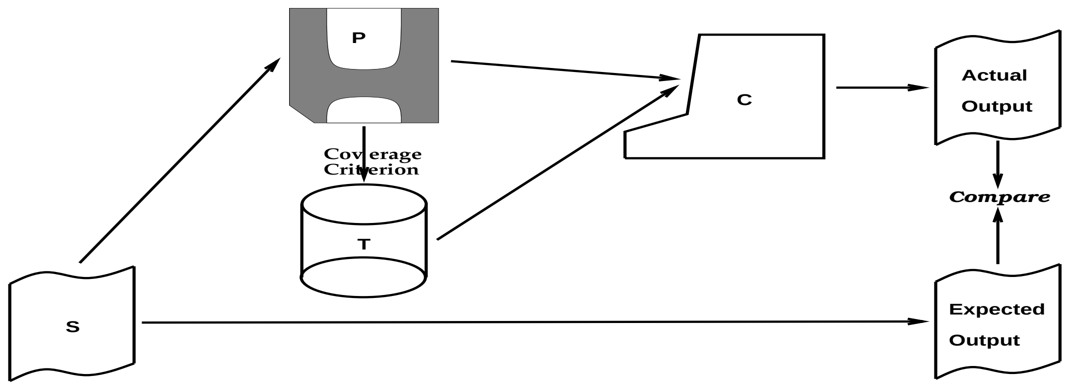


Figure 2: **Program-based Testing**

1.2 Specification-based Software Testing

Specification-based testing derives testing information from a specification of the software under test, rather than from the implementation. Even fully formal developments should be subjected to testing [BH95]. However, when the implementation is developed informally from a formal specification, the specification can play a major role in the testing process by allowing us to derive test inputs and the expected outputs from these inputs. There are two main roles a specification can play in software testing [RAO92]. The first is to provide the necessary information to check whether the output of the program is correct [PC89, PC90]. Checking the correctness of program outputs is known as the *oracle problem*. The second is to provide information to select test cases and to measure test adequacy [ZHM97].

Specification-based testing (SBT) offers many advantages in software testing. The (formal) specification of a software product can be used as a guide for designing functional tests for the product. The specification precisely defines fundamental aspects of the software, while more detailed and structural information is omitted. Thus, the tester has the essential information about the product's functionality without having to extract it from inessential details.

SBT from formal specifications offers a simpler, structured, and more formal approach to the developmet of functional tests than non-specification based testing techniques do. The strong relationship between specification and tests helps find faults and can simplify regression testing. An important application of specifications in testing is to provide test oracles. The specification is an authoritative description of system behavior and can be used to derive expected results for test data. Other benefits of SBT include developing tests concurrently with design and implementation, using the derived tests to validate the original specification, and simplified auditing of the testing process. Specifications can also be analyzed with respect to their testability [SC91].

There are three major approaches to formal software functional specifications: (1) model-based specifications, (2) property-oriented specifications such as axiomatic and algebraic specifications, and (3) state-based specifications [Off98b].

Model-based specification languages, such as Z and VDM, attempt to derive formal specifications of the software based on models of real world objects. Algebraic specification languages describe software by making formal statements, called *axioms*, about relationships among operations and the functions that operate on them. State-based specifications describe software in terms of state transitions. Typical state-based specifications define *preconditions* on transitions, which are values that specific variables must have for the transition to be enabled, and *triggering events*, which are changes in variable values that cause the transition to be taken.

This thesis focuses on generating test cases from state-based specifications. Methods and languages for developing state-based specifications are discussed in the next section.

1.3 Methods for State-based Specifications

There are several methods for specifying state-based systems. Offutt has defined specification-based testing criteria, and developed a test case generation model for SCR and CoRE specifications. The goal of this paper is generalize the model to UML specifications. Therefore, we introduce the SCR, CoRE, and UML methods in the next sub-section.

1.3.1 SCR Method

The Software Cost Reduction (SCR) specification uses structured tables to specify the behavioral requirements for real-time embedded software systems. One benefit of the SCR method is that its well-defined structure allows structural analysis to be used to check the consistency, completeness of the specification. Also, SCR applications provide traceability from the software requirements to the source code [FBWK92]. Various types of formal analysis have been applied to SCR specifications [HKL97, AG93].

In an SCR specification, a *modeclass* is a state machine whose states are called *system modes* or *modes*. Complex systems' behavior may be defined by several modeclasses operating in parallel. Each modeclass describes one aspect of the system's behavior, and the global behavior of the entire system is defined by the composition of the system's modeclasses. The system‘s environment is represented by a set of boolean environmental *conditions*. An *event* occurs when the values of a condition change from TRUE to FALSE or vice versa. Events therefore specify instants of time, while conditions specify intervals of time. Formally, a *conditioned event* is an event that occurs when certain conditions hold.

Modes and mode transitions specify system properties that hold under certain conditions. A *mode invariant* must be TRUE when the system enters the mode, must remain TRUE while the system stays in the mode, and can either be TRUE or become FALSE when the system leaves the mode. Mode invariants are the invariant properties of a system mode. If certain conditions hold then either the system is in a particular mode or the next system transition will be into that mode. Whenever the system is in a particular mode, certain system conditions have invariant values. SCR specifications specify software system functional behavior. System invariants are implicitly or sometimes explicitly specified in the specification. An SCR specification may include a set of invariants as safety assertions, but these system invariants are not additional constraints on the required behavior, they are included only as the goals that the tabular requirements are expected to imply. Also, if these invariants are not explicitly listed, then we should be able to derive them from the specification. The derived invariants can be compared with the explicit invariants as verification criteria.

1.3.2 CoRE

Consortium Requirement Engineering (CoRE) is a method for capturing, specifying, and analyzing real-time software requirements [MH97]. CoRE integrates the formal model of SCR with object-oriented and graphical techniques to define a methodology for generating a software requirements specification. A CoRE specification has two parts: a behavioral (formal) model and a class model. The *behavioral model* provides a standard structure for analyzing and capturing the behavioral requirements of an embedded system. The behavioral model is based on a four-variable model. The four variables are monitored, controlled, input, and output variables. Figure 3 illustrates the four-variable model. A *monitered variable* represents an environmental quantity that system responds to, a *controlled variable* is an environmental quantity that the system controls. An input variable is a variable through which the software senses the monitered variables. An *output variable* is a variable the software uses to change the controlled variables.

The behavioral model defines the required, externally visible behavior in terms of two relations from monitored variables to controlled variables called NAT (for nature) and REQ (for required). The NAT relation describes those constraints placed on the software system by the external environment. The REQ relation describes the additional constraints on the controlled variables that must be enforced by the software. REQ also describes properties that the software system is required to maintain between the monitored and controlled

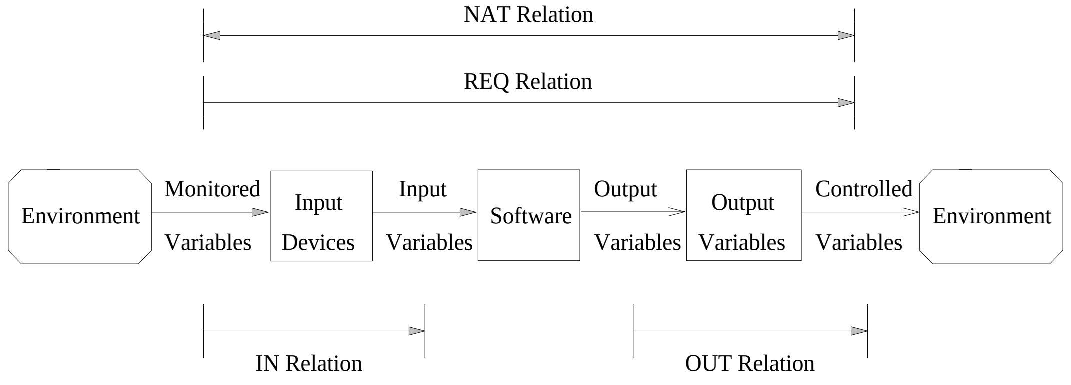


Figure 3: **The Four-Variable Model**

variables. CoRE treats the software system's actual inputs and outputs as resources available to the software to determine the values of monitored and controlled quantities. The relationship between the software software system input and output and the environmental variables is expressed in two additional relations called IN (for input) and OUT (for output).

The *class model* provides a set of facilities for packaging the behavioral model as a set of objects, classes, and superclasses. Classes group together portions of the specification logically related and likely to change. Each class has an *interface section* and an *encapsulated section.* The *interface section* contains all constants, monitored variables, modes, and terms that can be referenced by other classes. The *encapsulated section* contains information that can only be referenced by the enclosing class. A class is allowed to reference names on the interfaces of other classes, creating *dependencies* between classes. Dependencies are the primary relationships between classes and often depicted graphically as labeled arrows between classes. CoRE classes are categorized into *boundary* classes, *mode* classes, and *term* classes. Boundary classes group together the definitions of the *monitored* variables and *controlled* variables, and possibly encapsulate the corresponding *input* variables, *output* variables, REQ, IN, and OUT relations that are expected to change together. Boundary classes are often associated with a single external entity, such as a control panel or user display. Boundary classes serve to define and encapsulate monitored and controlled variables. Mode classes export the system modes used by several classes and encapsulate the rules transitioning among them. Mode classes export the modes associated with that class by placing the mode name on its interface. The rules for transitioning among modes are hidden within the encapsulated section. Terms are usually defined in the class they are most closely related with, but *term* classes can be created for terms not clearly associated with any single boundary or mode class. Term classes are used to collect in one location the definition of related terms that do not belong in a specific boundary or mode class. Terms, like monitored and controlled variables, are continuous functions of time.

1.3.3 UML

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems. The artifacts include requirements, architecture, design, source code, project plans, tests, prototypes, and releases etc. The UML provides a language for expressing, modeling and documenting a software system's architecture, requirements, tests, project planning activities and release management. The UML is not limited to modeling software, it is expressive enough to model nonsoftware systems.

The UML has three major elements that form a conceptual model of the language: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML. Things, relationships, and diagrams are the building blocks of UML. Things are the abstractions that are first-class citizens in a model, relationships tie these things together, and diagrams group interesting collections of things.

There are four kinds of things in the UML: *structural, behavioral, grouping*, and *annotational.*

Structural things are the nouns of the UML model. There are seven kinds of structural things: *class, interface, collaboration, use case, active class, component,* and *node.* A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a rectangle that includes name, attribute, and operation compartments represents a class.

An *interface* describes complete or part of the externally visible behavior of a class or component. An interface defines the specification of operations, but not implementations. Graphically, a circle with a name represents an interface.

A *collaboration* defines an interaction; it has structural and behavioral dimensions. A collaboration represents the implementation of patterns that make up a system. Graphically, an ellipse with dashed lines, including a name, represents a collaboration.

A *use case* is a description of a set of sequence of actions. A use case is used to structure the behavioral things in a model, and realized by collaboration. Graphically, an ellipse with solid lines, including a name, represents a use case.

The remaining three things - active classes, components, and nodes - are all class-like, but they have slight differences.

An *active class* is a class for which its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is presented in the same way with class, except the border lines of rectangle are bold.

A *component* represents the physical packaging of logical elements, such as classes, interfaces, and collaborations. Graphically, a rectangle with tabs, including a name, represents a component.

A *node* exists at run time, and represents a computational resource, generally memory and processing capability. Graphically, a cube with a name represents a node.

Behavioral things are the dynamic parts of UML model, they represent behavior over time and space. There are two kinds of behavioral things: *interaction* and *state machine.*

An *interaction* comprises a set of messages exchanged among a set of objects in a particular context. An interaction involves a number of other elements, including messages, action sequences, and links. Graphically, a directed line with an operation name represents a message.

A *state machine* specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behavior of an individual class or a collaboration of classes can be specified with a state machine. A state machine has states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a round rectangle with a name represents a state.

Interactions and state machines are connected to structural elements, mainly classes, collaborations, and objects.

Grouping things are the organizational parts of UML models. The package is the only kind of grouping thing in the UML. A *package* is a general-purpose mechanism for organizing elements into groups. Graphically, a tabbed folder with a name represents a package.

Annotational things are the comments to elements in a model. There is one main kind of annotational thing, a note. A *note* includes constraints and comments that are attached to an element or a group of elements. Graphically, a dog-eared rectangle with textual or graphical comment represents a note.

There are four kind of relationships in the UML: *dependency, association, generalization*, and *realization*.

A *dependency* is a semantic relationship between two things. The change in the independent thing may affect the semantics of the dependent thing. Graphically, a directed dashed line represents a dependency.

An *association* is a structural relationship that describes a set of links, a link being a connection among objects. *Aggregation* is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, a solid line, possibly directed, occasianlly including a label, and often containing other adornments, such as multiplicity and role names, represents an association.

A *generalization* is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). This relation is also referred as inheritance - the child inherits the structure and behavior of parent. Graphically, a solid line with a hollow arrowhead pointing to the parent represents a generalization.

A *realization* is a semantic relationship between classifiers – one classifier specifies a contract that another classifier carries out. It is also referred as *instantiation.* It is a special type of dependency that exists between a *parameterized* class and the class that is created as a result of instantiation. Realization relationships exist in two places, between interfaces and the classes or components that realize them, and between uses cases and the collaborations that realize them. The realization relationship is a cross between a generalization and a dependency. Graphically, a realization relationship is shown with a dashed line with a hollow arrowhead pointing to the specifying classifier.

UML supports the following kinds of diagrams:

1. Class diagrams
2. Object diagram
3. Use-case diagram
4. Interaction diagram
   - Sequence diagram
   - Collaboration diagram
5. Statechart diagram
6. Activity diagram
7. Component diagram
8. Deployment diagram

Each type of diagram captures a different perspective. A *class diagram* shows a set of classes, interfaces, collaborations, and their relationships. Class diagrams address the static design view of a system. A *object diagram* shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. A *use case diagram* shows a set of use cases and actors (a special kind of class) and their relationships. An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that are sent among them. Interaction diagrams address the dynamic view of a system. A *sequence diagram* emphasizes the time-ordering of messages. A *collaboration diagram* emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams can be transformed into each other. A *statechart diagram* shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic behavior of a system, they are especially important in modeling the behavior of an interface, class, and collaboration. Statechart diagrams emphasize the event-ordered behavior of an object, and this is particularly useful in modeling reactive systems. An *activity diagram* is a variation of the statechart diagram, in which states are activities representing operations, and transitions are triggered by completion of actions within states. Activity diagrams model the function of a system, and emphasize the flow of control among objects. A *component diagram* models development time relationships among components. A *deployment diagram* shows the configuration of run-time processing nodes and the components that live on them.

The interaction, state and deployment diagrams have features specially intended for the modelling of real-time systems.

The UML has semantic rules for names, scope, visibility, integrity, and execution of its models.

The common mechanisms in the UML are specifications, adornments, common divisions, and extensibility mechanisms. A *specification* provides a detailed textual statement of the syntax and semantics of a graphical notation. An *adornment* is a textual or graphical symbol added to the basic graphical notation. A *common division* mehanism divides the class and object, as well as interface and implementation. An *extensibility* mechanism provides means to extend the UML in controlled ways, thus makes the UML an open-ended language. The UML's extension mechanisms include stereotypes, tagged values, and constraints. A *stereotype* extends the vocabulary of the UML, it is used to establish new kinds of elements. A *tagged value* extends the properties of a UML building block, it allows to create new information in an element's specification. A *constraint* extends the semantics of a UML building block, it allows users to add new rules or modify existing ones of a building block.

1.3.4 Comparison of UML with SCR and CoRE

There are several major differences among UML, SCR, and CoRE:

1. UML is a modeling language, but SCR and CoRE are methods. A *modeling language* has *model elements*, *notation*, and *guidelines.* It does not have a standard process. A method consists of guidelines and rules, it has a set of *modeling concepts*, *views* and *notations.* A method usually has a standard process.
2. UML and CoRE support object-orientation, SCR does not.
3. All three of them support modeling object behavior through state machines.
4. UML supports hierarchical state machines, while SCR and CoRE do not.
5. UML is flexible, it is used to describe different artifacts of a software system. It is closer to natural language, yet has mechanisms for formal specification. SCR and CoRE are formal methods, which can be harder to understand.

The UML unified three prominent methodoligies in object-oriented software development field. Besides being used to specify, construct, visualize,and document software artifacts, the various UML notations help with comunication among people who are involved in the software development process. Obviously, UML has the capability to become a standard notation. This project studies the artifacts of UML to see if they can be used to generate test data. If the UML artifacts can be used to help generate tests, the results will help to make UML useful in practice.

The next chapter presents a model for generating test case from SCR specifications. The modification of the test case generation model for SCR specification to UML specification will be described in Chapter 3; a proof of concept tool that uses the models for SCR and UML specifications will be provided in Chapter 4; we will give our analytical results of these tools in Chapter 5; and finally, conclusions and recommendations are in Chapter 6.

2 TEST CASE GENERATION TECHNIQUE FOR SCR SPECIFICATION

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. 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. Offutt has carried out a research project to establish formal criteria and processes for generating system-level tests from functional requirements/specifications [Off98a]. His work presents 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.

This chapter 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.

2.1 Testing Model

Predicate satisfaction uses preconditions, invariants, and postconditions to create predicates, and then generates test cases to satisfy individual clauses within the predicates. This is closely related to previous code-based automatic test generation research [DO91]. The model presented here extends the promising ideas of predicate satisfaction in several ways.

Instead of just covering the pre and postconditions, it is important to use the tests to relate the preconditions to the postconditions. Tests should also be created to find faults, as well as to cover the input domain. This report presents examples using Software Cost Reduction specifications (SCR) [Hen80, AG93] and CoRE [FBWK92].

In this model, tests are generated as multi-part, multi-step, multi-level artifacts. The multi-part aspect means that a test case is composed of several components. Input values are the values for the test case; these are the values needed to directly satisfy the test requirements. The other components supply supporting values, including expected outputs, inputs necessary to get to the appropriate pre-state, and inputs necessary to observe the effect of the test case. The multi-step aspect means that tests are generated in several steps from the functional specifications by a refinement process. The functional specifications are refined into test specifications, which are then refined into test scripts. The multi-level aspect means that tests are generated to test the software at several levels of abstraction.

The multiple parts of the test case are based on research in test case specifications [BHO89, SC93]. The category-partition method includes a test specification language called TSL [BHO89], which was designed for command-line interface software. A *test case* in TSL is a command or software function and values for its parameters and relevant environment variables. A *test specification* in TSL consists of the operations necessary to create the environmental conditions (called the SETUP portion), the test case itself, whatever command is necessary to observe the affect of the operation (VERIFY in TSL), and any exit command (CLEANUP in TSL). Test specifications written in TSL can be used to automatically generate executable *test scripts* that are ready for input to the software.

In this state-based approach, the test case operation of TSL is replaced by `Test case values`, which are derived directly from a triggering event and the preconditions for the transition. The setup operation is called a `Prefix`, and includes all inputs necessary to reach the pre-state and to give the triggering event variable its before-value. Any inputs that are necessary to show the results are `Verify` operations, 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.

The model currently defines test cases at four levels: (1) the transition coverage level, (2) the full predicate coverage level, (3) the transition-pair coverage level, and (4) the complete sequence level. These are defined in the next four subsections. To apply these, a state-based requirement/specification is viewed as a directed graph, called the *specification graph*. Each node represents a state (or mode) in the requirement/specification, and edges represent possible transitions.

It is possible to apply all levels, or to choose a level based on a cost/benefit tradeoff. The first two are related; the transition coverage level requires many fewer test cases than the full predicate coverage level, but if the full predicate coverage level is used, the tests will also satisfy the transition coverage level (full predicate coverage *subsumes* transition coverage). Thus only one of these two should be used. The latter two levels are meant to be independent; transition-pair coverage is intended to check the interfaces among states, and complete sequence testing is intended to check the software by executing the software through complete execution paths. As it happens, transition-pair coverage subsumes transition coverage, but they are designed to test the software in very different ways.

2.1.1 Transition Coverage Level

This level is analogous to the branch coverage criterion in structural testing. The requirement is that each transition in the specification graph is taken at least once. This is another way of requiring that each precondition in the specification is satisfied at least once.

Transition coverage:	**Each transition in the specification graph is taken at least once.**

2.1.2 Full Predicate Coverage Level

One question during testing is whether the predicates in the specifications are formulated correctly. Small inaccuracies in the specification predicates can lead to major problems in the software. The full predicate coverage level takes the philosophy that to test the software, we should at least provide inputs to test each clause in each predicate. This level requires that each clause in each predicate on each transition is tested independently, thus attempting to address the question of whether each clause is necessary and is formulated correctly. Following the definitions in DO178B [SC-92], the Boolean operators are AND, OR, and NOT, and clause and predicate are defined as follows (DO178B uses the terms "condition" and "decision"):

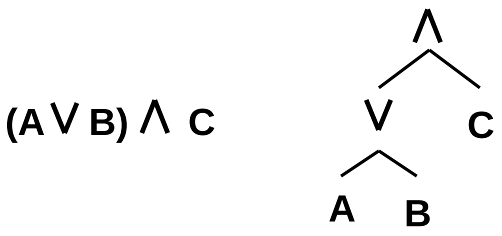
- A *clause* is a Boolean expression that contains no Boolean operators. For example, relational expressions and Boolean variables are clauses.
- A *predicate* is a Boolean expression that is composed of clauses and zero or more Boolean operators. A predicate without a Boolean operator is also a clause. If a clause appears more than once in a predicate, each occurrence is a distinct clause.

The concept of full predicate coverage is based on the structural testing criterion of modified condition/decision coverage (MC/DC) [CM94], which requires that every decision and every condition within the decision has taken every outcome at least once, and every condition has been shown to independently affect its decision. The full predicate coverage level is defined as follows:

Full predicate coverage:	**Each clause in turn takes the values of `True` and `False` while all other clauses in the predicate have values such that the value of the predicate will always be the same as the clause being tested.**

This definition ensures that each clause is tested without being influenced by the other clauses. Note that if full predicate coverage is achieved, transition coverage will also be achieved. To satisfy the requirement that the *test clause* controls the value of the predicate, other clauses must be either `True` or `False`. If the predicate is $(X \wedge Y)$, and the test clause is X, then Y must be `True`. Likewise, if the predicate is $(X \vee Y)$, Y must be `False`.

The simplest way to satisfy full predicate is to use an expression parse tree. An expression parse tree is a binary tree that has binary and unary operators for internal nodes, and variables or constants at leaf nodes. The relevant binary operators are **and** $(\wedge)$, **or** $(\vee)$, and the relational operators $\{>, <, \leq, \geq, =, \neq\}$; the relevant unary operator is **not**. For example, the expression parse tree for $(A \vee B) \wedge C$ is:



Given a parse tree, full predicate coverage is satisfied by walking the tree. First, a test clause is chosen. Then the parse tree is walked from the test clause up to the root, then from the root down to each clause. If its parent is $\vee$, its sibling must have the value of `False`, if its parent is $\wedge$, its sibling must have the value of `True`. If a node is the inverse operator **not**, the parent node is given the inverse value of the child node. This is repeated for each node between the test clause and the root.

Once the root is reached, values can be propagated back down using a simple tree walk. If a $\wedge$ node has the value of `True`, then both children must have the value `True`; if a $\wedge$ node has

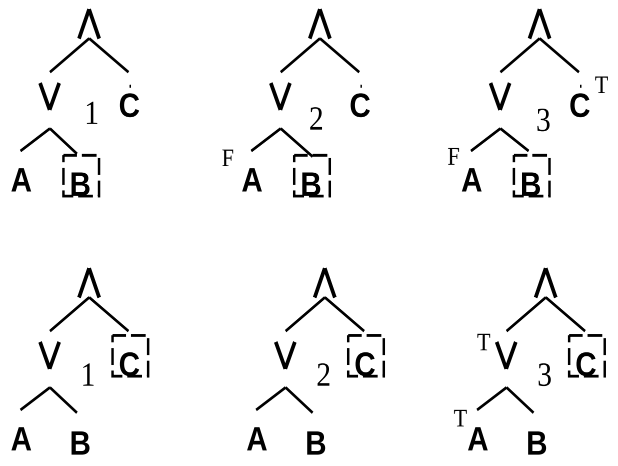


Figure 4: **Constructing Test Case Requirements From an Expression Parse Tree**

the value of `False`, then either child must have the value `False` (which one is arbitrary). If a ∨ node has the value of `False`, then both children must have the value `False`; if a ∨ node has the value of `True`, then either child must have the value `True` (which one is arbitrary). If a node is the inverse operator **not**, the parent node is given the inverse value of the child node.

Figure 4 illustrates the process for the expression above, showing both B and C as test clauses. In the top sequence, B is the test clause (shown with a dashed box). In tree 2, its sibling, A, is assigned the value `False`, and in tree 3, C is assigned the value `True`. In the bottom sequence, C is the test clause. In tree 2, C's sibling is a ∨ node, and is assigned the value `True`. In tree 3, A is assigned the value `True`. Note that in tree 3, either A or B could be given the `True` value; the choice is arbitrary.

Although this may seem complicated, it is easy to automate and relatively straightforward to apply by hand. It has also been our experience that most specification predicates tend to be fairly small and simple in form.

Full predicate test cases sample from both **valid** and **invalid** transitions, with only one transition being valid at a time. In addition, the test engineer may choose semantically meaningful combinations of conditions. Testing with invalid inputs can help find faults in the implementation as well as the formulation of the specifications. Of course, this brings up a philosophical question about responsibility. Many developers believe that if a software component has well-stated preconditions, it is the responsibility of the user to ensure that the preconditions are met. This can be taken to imply that the component does not need to be tested with inputs that violate the preconditions (as in the design-by-contract approach [MM92]). Without taking a side on this issue, the technique described here provides a mechanism for developing invalid inputs; they can be used or discarded as the tester sees fit.

As a concrete example, consider the formula whose parse tree was given above, $(A \vee B) \wedge C$. The following partial truth table provides the values for the test clauses:

	$(A$	$\vee$	$B)$	$\wedge$	C
1	T				
2	F				
3			T		
4			F		
5					T
6					F

To ensure the requirement that the test clause must control the final result, the partial truth table must be filled out as follows (for the last two entries, either A or B could have been `True`, both were assigned the value `True`):

	$(A$	$\vee$	$B)$	$\wedge$	C
1	**T**		F		T
2	**F**		F		F
3	F		**T**		T
4	F		**F**		F
5	T		T		**T**
6	T		T		**F**

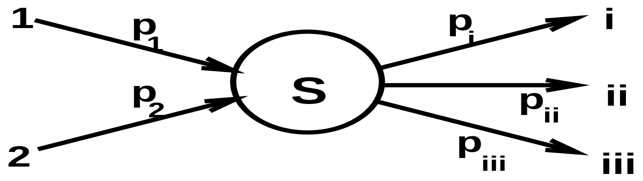
Some specification languages, such as SCR and CoRE, treat triggering event variables differently from other variables in transition predicates. When this is the case, the clause that corresponds to the triggering event should be given a different value, but should **remain** the triggering event. If it is no longer a triggering event, it is equivalent to not executing a test case. Moreover, a triggering event actually specifies two values, a before-value and an after-value. To fully test predicates with triggering events, both before- and after-values should be tested. This is done by assuming two versions of the triggering event, A and A', where A represents the before-value of A and A' represents its after-value.

2.1.3 Transition-Pair Coverage Level

The previous testing levels test transitions independently, but do not test sequences of state transitions. This level requires that pairs of transitions be taken.

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.**

Consider the following state:



To test the state `S` at the transition-pair level, six tests are required: (1) from 1 to i, (2) 2 to i, (3) 1 to ii, (4) 2 to ii, (5) 1 to iii, and (6) 2 to iii. These tests require inputs that satisfy the predicates: P_1:P_i, P_1:P_{ii}, P_1:P_{iii}, P_2:P_i, P_2:P_{ii}, and P_2:P_{iii}.

2.1.4 Complete Sequence Level

It seems very unlikely that any successful test method could be based on purely mechanical methods; at some point the experience and knowledge of the test engineer must be used. Particularly at the system level, effective testing probably requires detailed domain knowledge. A *complete sequence* is a sequence of state transitions that form a complete practical use of the system. This use of the term is similar to that of "use cases". In most realistic applications, the number of possible sequences is too large to choose all complete sequences. In many cases, the number of complete sequences is infinite.

Complete sequence level:	**The test engineer must define meaningful sequences of transitions on the specification graph by choosing sequences of states that should be entered.**

Which sequences to choose is something that can only be determined by the test engineer with the use of domain knowledge and experience. This is the least automatable level of testing.

2.1.5 Summary

To generate tests according to this methodology, tests must be generated at the following four levels:

1. Transition Coverage Level
    - Definition: Each transition in the specification graph is taken at least once.
    - Requirements: Predicates on the edges must evaluate to `True`.
    - Specifications:
        - Prefix: Inputs to get to the pre-state immediately preceding the edge.
        - Test case values: Assignments to variables to satisfy the preconditions and a new value for the triggering event variable.
        - Verify: Input to the software to show the post-state; depends on the software.
        - Exit: Input to the software to stop execution; depends on the software.
        - Expected outputs: Post-state from the requirements.
    - Script: A sequence of inputs to the software; the format depends on the software.
2. Full Predicate Coverage Level
    - Definition: Each clause in turn takes the values of `True` and `False` while all other clauses in the predicate have values such that the value of the predicate will always be the same as clause being tested.
    - Requirements: Certain rows from the truth tables of the predicates must be chosen.
    - Specifications:
        - Prefix: Inputs to get to the pre-state immediately preceding the edge.
        - Test case values: Assignments to variables to satisfy the preconditions and a new value for the triggering event variable.
        - Verify: Inputs to the software to show the post-state; depends on the software.

    - Exit: Input to the software to stop execution; depends on the software.
    - Expected outputs: Post-state from the requirements.
  - Script: A sequence of inputs to the software; the format depends on the software.

3. Transition-Pair Coverage Level
    - Definition: For each state S, form test requirements such that for each incoming transition and each outgoing transition, both transitions must be taken in sequence.
    - Requirements: Predicates on two edges of the specification graph must evaluate to `True`.
    - Specifications:
        - Prefix: Inputs to get to the pre-state immediately preceding the edge.
        - Test case values: Assignments to variables to satisfy the preconditions and a new value for the triggering event variable.
        - Verify: Inputs to the software to show the post-state; depends on the software.
        - Exit: Input to the software to stop execution; depends on the software.
        - Expected outputs: Post-state from the requirements.
    - Script: A sequence of inputs to the software; the format depends on the software.

4. Complete Sequence Level
    - Definition: The test engineer must define **meaningful** sequences of transitions on the specification graph by choosing sequences of states that should be entered.
    - Requirements: Lists of states.
    - Specifications:

  - Setup: Should be empty
  - Test case value: Value assignments necessary to take every transition on the sequence path.
  - Verify: Inputs to the software to show the post-state; depends on the software.
  - Exit: Input to the software to stop execution; depends on the software.
  - Expected outputs: Sequence of states.
- Script: A sequence of inputs to the software; the format depends on the software.

2.2 Derivation Process

This section presents a process that can be used to derive test cases. The process steps for all four levels of testing are presented together, as there is a fair amount of overlap. If not all four levels are used, some of these steps should be skipped. The steps are presented as being purely manual; in the future schemes for automating as many of the steps as possible will be developed.

The general process is shown in Figure 5; this merely reflects the multi-step aspect of our test generation process that was presented in Section 2.1.

1. **Develop transition conditions.** The first step is to develop **transition conditions**, which are predicates that define under what conditions each transition will be taken. With some specification languages (e.g., SCR and CoRE), the transition conditions are encoded directly into the specifications. With other languages, the conditions may have to be derived.

2. **Develop specification graph.** The specification graph was described in Section

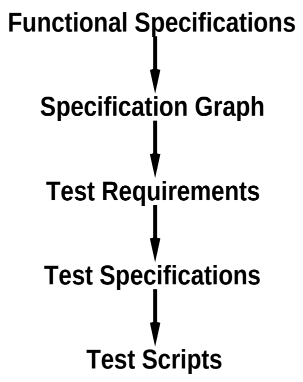


Figure 5: **General Process for Generating Test Cases**

2.1. It can be directly derived from the specification table, and edges annotated with the conditions derived in step 1.

This is the point at which the process separates for the four testing levels.

3. **Develop transition coverage test requirements.**

   (a) **Derive transition predicates.** The conditions from step 1 are listed one at a time to form test requirements.

4. **Develop full predicate test requirements.**

   (a) **Construct truth tables for all predicates in the specification graph.** The predicate coverage tests can be based on an expression tree or directly on the predicates. If all the logical connectors are the same (all ANDs or all ORs), it is a simple matter to modify the values for the clauses in the predicates directly. If ANDs and ORs are mixed freely, however, it is less error-prone to construct the expression tree. Most specification languages differentiate between trigger events and preconditions; in this case, the trigger events must be marked specially so that the test engineer remembers to put that input after the precondition inputs.

5. **Develop transition-pair test requirements.**

   (a) **Identify all pairs of transitions.** Transition-pair tests are ordered pairs of condition values, each representing an input to the state and an output from the state. These are formed by enumerating all the input transitions (M), all the output transitions (N), then creating $M * N$ pairs of transitions.

   (b) **Construct predicate pairs.** These pairs of transitions are then replaced by the predicates from the specification graph.

6. **Develop complete sequence test requirements.**

   (a) **Identify complete lists of states.** The complete sequence tests are created by the tester. This is done by choosing sequences of states from the specification graph to enter.

   (b) **Construct sequence of predicates.** The sequences of states are transformed into sequences of conditions that will cause those states to be entered.

   At this point, test requirements for the four levels will be in a uniform format, as truth assignments for predicates.

7. **Construct test specifications.** For each unique test requirement, generate prefix values, test case values, verify conditions, exit conditions, and expected outputs. Note that there may be a fair amount of overlap among the test requirements, thus the "unique" restriction. Generating the actual values may involve solving some algebraic equations. For example, if a condition is $A > B$, values for A and B must be chosen to give the predicate the appropriate value. It is also at this point that some "invalid" tests might be discovered. For example, it may be impossible or meaningless to pair all incoming and outgoing transitions for each state. In this case, certain test specifications will be discarded.

8. **Construct test scripts.** Each test specification is used to construct one test script. The actual scripts must reflect the input syntax of the program, so knowledge of the input syntax of the program is required for this step. (Note that this is the only step that requires any knowledge of the implementation, all preceding steps depend solely on the functional specifications.)

2.2.1 Automation Notes

It is possible to automate almost all of this derivation process. If a machine-readable form of the specification table is available, the transition conditions can be read directly from the table. The specification graph can then be automatically created from the states and transition conditions. Test requirements take the form of partial truth tables defined on transition predicates, state transition predicates, and pairs of state transition predicates. Given a formal functional specification, most if not all of these test requirements can be generated automatically. The prefix of a test case includes inputs necessary to put the system into a particular pre-state. Given the specification graph, many of these prefixes can be generated automatically. One open question is whether this problem is generally solvable (unlike the related reachability problem in general software, which is generally unsolvable), and how to solve or partially solve the problem. It is also possible to automatically refine test specifications into test scripts. Finally, algorithms for automatically generating test scripts can be developed, although the input syntax of the program will be needed. The final step, generating complete sequence tests, cannot be fully automated. But an appropriate interface could present the specification graph, and allow the tester to choose sequences of states by pointing and clicking on the screen. Each time a state is chosen, the transition from the previous state could be automatically translated into values and saved as part of the test case. This would allow the tester's job to become the purely intellectual exercise of choosing sequences of states to be entered.

2.3 Cruise Control Example

This section presents an example of applying the test data generation model to a specification for an automobile cruise control system. Cruise control is a common example in the literature [Atl94, Jin96]. Table 1 shows the specifications for the system (note that it does not model the throttle). It 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. 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 an environmental condition C does

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

Table 2: **Cruise Control Specification Predicates**

not affect a conditioned event, the table entry is marked with a hyphen "-" (don't care conditions).

Table 2 shows the transitions of the specification in predicate form, numbered P_1 through P_{12}. Figure 6 shows the specification graph, with the edges labeled with the condition numbers.

2.3.1 Full Predicate Coverage Level

There are nine transitions in the cruise control specifications, and twelve disjunctive predicates (Table 2 shows each disjunctive predicate on a separate line). For convenience, the technique is applied by considering each predicate specification separately. As stated in Section 2.1.2, both the before- and after-values of the triggering event should be tested. For SCR, this is handled by treating @ as an operator and expanding it algebraically. The relevant expansions are:

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

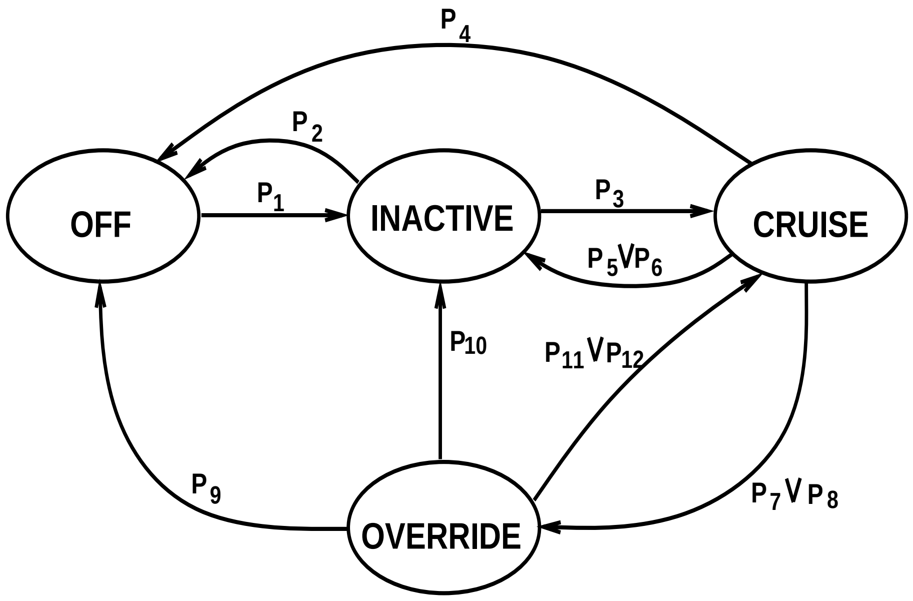


Figure 6: **Specification Graph for Cruise Control**

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

Table 3 repeats Table 2, but with the trigger events expanded appropriately.

The test case requirements for the full predicate coverage level are below with the environmental variables shown as I (Ignited) R (Running), T (Toofast), B (Brake), A (Activate), D (Deactivate), and S (Resume). 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-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_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 3: **Expanded Cruise Control Specification Predicates**

The first two transitions have only one clause, so the only test cases are based on the triggering event. The third 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 test case requirements are:

Pre State	$Activate$	$Ignited$	$Running$	$Brake$	$Activate'$	Post State
INACTIVE	F	t	t	f	T	CRUISE
INACTIVE	T	t	t	f	T	INACTIVE
INACTIVE	F	f	t	f	T	OFF
INACTIVE	F	t	f	f	T	INACTIVE
INACTIVE	F	t	t	t	T	INACTIVE
INACTIVE	F	t	t	f	F	INACTIVE

The first row is the predicate as it appears in the specification; every clause is `True`. The subsequent rows make each clause false in turn. Because there are no $\vee$ operators, the full predicate coverage criterion is satisfied by holding all other clauses `True`.

Below are the requirements for all the predicates in the cruise control program. There are 54 test cases for the 12 predicates.

	Pre State	Variable Values	Triggering Event	Post State
		I R T B A D S		
P1	OFF	F - - - - - -	$Ignited' =$ True	INACTIVE
	OFF	T - - - - - -	$Ignited' =$ True	OFF
	OFF	F - - - - - -	$Ignited' =$ False	OFF
P2	INACTIVE	T - - - - - -	$Ignited' =$ False	OFF
	INACTIVE	F - - - - - -	$Ignited' =$ False	INACTIVE
	INACTIVE	T - - - - - -	$Ignited' =$ True	INACTIVE
P3	INACTIVE	t t - f F - -	$Activate' =$ True	CRUISE
	INACTIVE	f t - f F - -	$Activate' =$ True	INACTIVE
	INACTIVE	t f - f F - -	$Activate' =$ True	INACTIVE
	INACTIVE	t t - t F - -	$Activate' =$ True	INACTIVE
	INACTIVE	t t - f T - -	$Activate' =$ True	INACTIVE
	INACTIVE	t t - f F - -	$Activate' =$ False	INACTIVE
P4	CRUISE	T - - - - - -	$Ignited' =$ False	OFF
	CRUISE	F - - - - - -	$Ignited' =$ False	CRUISE
	CRUISE	T - - - - - -	$Ignited' =$ True	CRUISE
P5	CRUISE	t T - - - - -	$Running' =$ False	INACTIVE
	CRUISE	f T - - - - -	$Running' =$ False	CRUISE
	CRUISE	t F - - - - -	$Running' =$ False	CRUISE
	CRUISE	t T - - - - -	$Running' =$ True	CRUISE
P6	CRUISE	t - F - - - -	$Toofast' =$ True	INACTIVE
	CRUISE	f - F - - - -	$Toofast' =$ True	CRUISE
	CRUISE	t - T - - - -	$Toofast' =$ True	CRUISE
	CRUISE	t - F - - - -	$Toofast' =$ False	CRUISE
P7	CRUISE	t t f F - - -	$Brake' =$ True	OVERRIDE
	CRUISE	f t f F - - -	$Brake' =$ True	CRUISE
	CRUISE	t f f F - - -	$Brake' =$ True	CRUISE
	CRUISE	t t t F - - -	$Brake' =$ True	CRUISE
	CRUISE	t t f T - - -	$Brake' =$ True	CRUISE

	CRUISE	t	t	f	F	-	-	-	*Brake*′ = False	CRUISE
P8	CRUISE	t	t	f	-	-	F	-	*Deactivate*′ = True	OVERRIDE
	CRUISE	f	t	f	-	-	F	-	*Deactivate*′ = True	CRUISE
	CRUISE	t	f	f	-	-	F	-	*Deactivate*′ = True	CRUISE
	CRUISE	t	t	t	-	-	F	-	*Deactivate*′ = True	CRUISE
	CRUISE	t	t	f	-	-	T	-	*Deactivate*′ = True	CRUISE
	CRUISE	t	t	f	-	-	F	-	*Deactivate*′ = False	CRUISE
P9	OVERRIDE	T	-	-	-	-	-	-	*Ignited*′ = False	OFF
	OVERRIDE	F	-	-	-	-	-	-	*Ignited*′ = False	OVERRIDE
	OVERRIDE	T	-	-	-	-	-	-	*Ignited*′ = True	OVERRIDE
P10	OVERRIDE	t	T	-	-	-	-	-	*Running*′ = False	INACTIVE
	OVERRIDE	f	T	-	-	-	-	-	*Running*′ = False	OVERRIDE
	OVERRIDE	t	F	-	-	-	-	-	*Running*′ = False	OVERRIDE
	OVERRIDE	t	T	-	-	-	-	-	*Running*′ = True	OVERRIDE
P11	OVERRIDE	t	t	-	f	F	-	-	*Activate*′ = True	CRUISE
	OVERRIDE	f	t	-	f	F	-	-	*Activate*′ = True	OVERRIDE
	OVERRIDE	t	f	-	f	F	-	-	*Activate*′ = True	OVERRIDE
	OVERRIDE	t	t	-	t	F	-	-	*Activate*′ = True	OVERRIDE
	OVERRIDE	t	t	-	f	T	-	-	*Activate*′ = True	OVERRIDE
	OVERRIDE	t	t	-	f	F	-	-	*Activate*′ = False	OVERRIDE
P12	OVERRIDE	t	t	-	f	-	-	F	*Resume*′ = True	CRUISE
	OVERRIDE	f	t	-	f	-	-	F	*Resume*′ = True	OVERRIDE
	OVERRIDE	t	f	-	f	-	-	F	*Resume*′ = True	OVERRIDE
	OVERRIDE	t	t	-	t	-	-	F	*Resume*′ = True	OVERRIDE
	OVERRIDE	t	t	-	f	-	-	T	*Resume*′ = True	OVERRIDE
	OVERRIDE	t	t	-	f	-	-	F	*Resume*′ = False	OVERRIDE

Test specifications

The actual test specifications and test scripts are mechanically derived from the above test requirements, and are too numerous to list. 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 (post) 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*	= `False`	– Trigger before-value
	Ignited	= `False`	– Condition variable
	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
	Running	= `False`	– 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	= `True`	– Condition variable
	Brake	= `True`	– Condition variable
	Activate	= `True`	– Triggering event
Expected outputs:	INACTIVE		

5. Test specification P3-5:

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		

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 probably 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. Her 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. 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, it seems likely that the deterministic nature of state-based systems means 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
```

2.3.2 Transition-Pair Coverage Level

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. The pairs for the OFF state are:

1. P2:P1
2. P4:P1
3. P9:P1

The pairs for the INACTIVE state are:

1. P1:P2
2. P1:P3
3. P10:P2
4. P10:P3
5. (P5 OR P6):P2
6. (P5 OR P6):P3

The pairs for the CRUISE state are:

1. P3:P4
2. P3:(P5 OR P6)

```
3. P3:(P7 OR P8)
4. (P11 OR P12):P4
5. (P11 OR P12):(P5 OR P6)
6. (P11 OR P12):(P7 OR P8)
```

The pairs for the OVERRIDE state are:

```
1. (P7 OR P8):P9
2. (P7 OR P8):P10
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. Rather than list before- and after-values for the triggering events in this table, only the after-values are shown; the before-values are assumed to be the inverse.

		I	R	T	B	A	D	S	
OFF:	1. INACTIVE	F	-	-	-	-	-	-	OFF
	OFF	T	-	-	-	-	-	-	INACTIVE
	2. CRUISE	F	-	-	-	-	-	-	OFF
	OFF	T	-	-	-	-	-	-	INACTIVE
	3. OVERRIDE	F	-	-	-	-	-	-	OFF
	OFF	T	-	-	-	-	-	-	INACTIVE
INACTIVE:	1. OFF	T	-	-	-	-	-	-	INACTIVE
	INACTIVE	F	-	-	-	-	-	-	OFF
	2. OFF	T	-	-	-	-	-	-	INACTIVE
	INACTIVE	t	t	-	f	T	-	-	CRUISE
	3. OVERRIDE	t	F	-	-	-	-	-	INACTIVE
	INACTIVE	F	-	-	-	-	-	-	OFF
	4. OVERRIDE	t	F	-	-	-	-	-	INACTIVE
	INACTIVE	t	t	-	f	T	-	-	CRUISE
	5. CRUISE	t	F	-	-	-	-	-	INACTIVE
	OR								
	CRUISE	t	-	T	-	-	-	-	INACTIVE
	INACTIVE	F	-	-	-	-	-	-	OFF
	6. CRUISE	t	F	-	-	-	-	-	INACTIVE
	OR								
	CRUISE	t	-	T	-	-	-	-	INACTIVE
	INACTIVE	t	t	-	f	T	-	-	CRUISE
CRUISE:	1. INACTIVE	t	t	-	f	T	-	-	CRUISE

		CRUISE	F - - - - - -	OFF
	2.	INACTIVE	t t - f T - -	CRUISE
		CRUISE	t F - - - - -	INACTIVE
		OR		
		CRUISE	t - T - - - -	INACTIVE
	3.	INACTIVE	t t - f T - -	CRUISE
		CRUISE	t t f T - - -	OVERRIDE
		OR		
		CRUISE	t t f - - T -	OVERRIDE
	4.	OVERRIDE	t t - f T - -	CRUISE
		OR		
		OVERRIDE	t t - f - - T	CRUISE
		CRUISE	F - - - - - -	OFF
	5.	OVERRIDE	t t - f T - -	CRUISE
		OR		
		OVERRIDE	t t - f - - T	CRUISE
		CRUISE	t F - - - - -	INACTIVE
		OR		
		CRUISE	t - T - - - -	INACTIVE
	6.	OVERRIDE	t t - f T - -	CRUISE
		OR		
		OVERRIDE	t t - f - - T	CRUISE
		CRUISE	t t f T - - -	OVERRIDE
		OR		
		CRUISE	t t f - - T -	OVERRIDE
OVERRIDE:	1.	CRUISE	t t f T - - -	OVERRIDE
		OR		
		CRUISE	t t f - - T -	OVERRIDE
		OVERRIDE	F - - - - - -	OFF
	2.	CRUISE	t t f T - - -	OVERRIDE
		OR		
		CRUISE	t t f - - T -	OVERRIDE
		OVERRIDE	t F - - - - -	INACTIVE
	3.	CRUISE	t t f T - - -	OVERRIDE
		OR		
		CRUISE	t t f - - T -	OVERRIDE
		OVERRIDE	t t - f T - -	CRUISE
		OR		
		OVERRIDE	t t - f - - T	CRUISE

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		

2.3.3 Complete Sequence Level

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 example.

2.3.4 Results

To evaluate this technique, a model of the cruise control problem was implemented in about 400 lines of C. 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 by an explicit input state value.

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]. The program, `cruise`, has five functions, 184 total blocks, and 174 decisions. The 54 test cases covered 163 of the blocks (89%) and 155 of the decisions (89%). Of the 20 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 given in Table 1. The remaining decisions were left uncovered because the variables *Activate*, *Deactivate*, and *Resume* are only used as triggering events, not condition variables. 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.

3 TEST CASE GENERATION TECHNIQUES FOR UML SPECIFICATION

This section gives a detailed discussion of how UML specification can be used to generate test cases, and how the procedure works.

3.1 Testing Model

UML can be used to specify a wide range of aspects of a system. Statecharts are the most obvious place to start with test data generation. UML statecharts are based on finite state machines using an extended Harel state chart notation, and are used to represent the behavior of an object. Behavior binds the structure of an object to its attributes and relationships so that the object can meet its responsibilities. An object's methods implement its behavior. By testing each method, we can test some elements of an object's behavior, but not the overall behavior. State machines describes overall behavior of an object, thus test cases generated from state machines test overall behavior of an object. The other advantage of UML statecharts is that they have the same semantics as the other state-based specifications. This makes it possible to generalize the test case generation model described it Chapter 2 to UML statecharts. To modify the model to UML statecharts, we explain the semantics and syntax of statecharts in UML specifications.

The *state* of an object is the combination of all attribute values and objects the object contains; a state is *static*, at a point of time, rather than dynamic. The dynamics of the object is modeled through transitions, which is a movement from a state to another. When the object is in a given state, an *event* occurs that moves it to another state (or back to the same state). During the transition from state to state, an *action* occurs. Transitions are more elaborate in the UML than in the Harel notation. The UML syntax for transition is:

```
event (arguments) [condition] ^ target.sendEvent (arguments)/operation (arguments)
```

Each of these fields is optional – even the name may be omitted if it is clear when the transition will be taken.

The *event* is the name of the transition. Often this is the only thing specified for the transition. The transition has an optional *argument* list to indicate when data is present in the transition, such as an error code or a monitored value. This argument list is enclosed within parentheses like a standard function call. A guard *condition* is shown in square brackets. A guard is a condition that must be met before the transition is taken. The *sendEvent* list is a comma- separated list of events. Each event is directed toward a *target* object, and may have *arguments*. Such events will be propagated outside of the enclosing object as a result of this transition. This is one way concurrent state machines communicate, allowing a transition in one state machine to affect other concurrent state machines. Lastly, the *operation* list specifies a comma-separated list of functions (each with possible arguments) that will be called as a result of transition being taken.

Within states, both *entry* and *exit actions*, as well as an ongoing *activity*, may be specified. An entry action is a function that is called when the state is entered (even when the transition is self-directed). An exit action is a function that is executed when the state is exited (even when the transition is self-directed). Activities denote processing that continues until completion, or until interrupted by a transition (even when the transition is self-directed).

The test objects for the state-transition model are *transition paths*, the paths through the graph that represent a full object life cycle from creation to destruction. That is, each test object represents one possible sequence of states between the birth and the death of an object. We may not be able to test the full object life cycle model with the current model, only part of it.

UML categorizes transitions into five types: *high-level transitions*, *compound transitions*, *internal transitions*, *completion transitions*, and *enabled transitions.* A *high-level* or *group transition* originates from the boundary of composite states. A composite state is a state that consists of either concurrent substates or disjoint substates. If triggered, it exits all substates of the composite state starting the exit action with the innermost states in the active state configuration. A *compound transition* originates from a set of states and targets a set of states. An *internal transition* executes without exiting or re-entering the state in which it is defined. A *completion transition* is a transition without an explicit trigger, although it may have a guard defined. When all transitions and entry actions and activities in the currently active state are completed, a *completion event* instance is generated. This event is the implicit trigger for a completion transition. An *enabled transition* is enabled by an event, and it orginates from an active state. An enabled transition is triggered when there exists at least one full path from the source state to the target state.

This research is only interested in enabled transitions. The previous model was based primarily on predicate satisfaction. In UML, the enabled transitions are similar to transitions that are based on the notion of *predicate satisfaction.* For the generalization purpose of the model, the other types of transitions are not considered.

Four kinds of events can be specified in UML: call event, signal event, time event, and change event. A *call event* represents the *reception* of a request to synchronously invoke a specific operation. The expected result is the execution of a sequence of actions that characterize

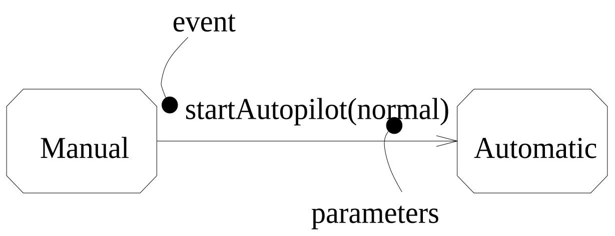


Figure 7: **Call Events**

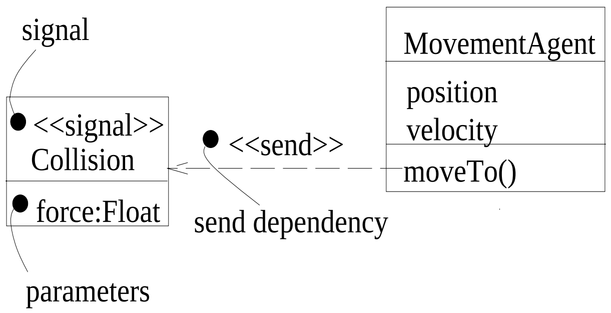


Figure 8: **Signal Events**

the operation behavior at a particular state. Object creation and object destruction are two special cases of a call event. Figure 7 illustrates a call event.

A *signal event* represents the reception of a particular (synchronous) signal. A signal event instance should not be confused with the action (e.g. send action) that generated it. Exceptions are examples of signal events. The signal events are modeled as stereotyped classes, as shown in Figure 8. The *send* dependency indicates that an operation sends a particular signal.

A *time event* represents the passage of a designated period of time after a designated event (often the entry of the current state) or the occurrence of a given date/time. In the UML, the time event is modeled by using the keyword *after* followed by some expression that

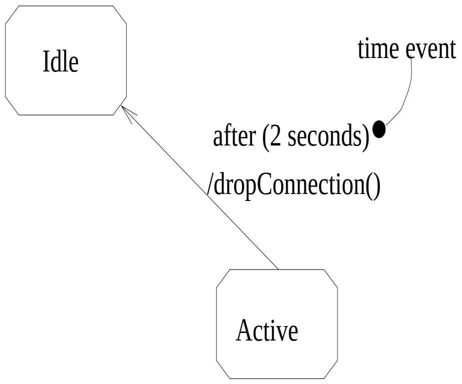


Figure 9: **Time Events**

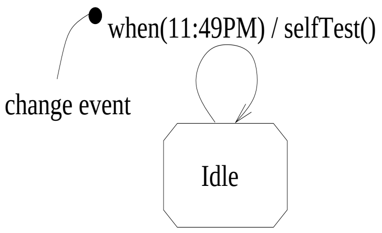


Figure 10: **Change Events**

evaluates to a period of time. Figure 9 illustrates a time event.

A *change event* models an event that occurs when an explicit boolean expression becomes true as a result of a change in value of one or more attributes or associations. A change event is raised implicitly and is not the result of an explicit change event action. The change event is different from a guard. A guard is only evaluated at the time an event is dispatched whereas, conceptually, the boolean expression associated with a change event is evaluated continuously until it becomes true. The event that is generated remains until it is consumed even if the boolean expression changes to false after that. In the UML, the change event is modeled by using the keyword *when* followed by some Boolean expression. Figure 10 illustrates a change event.

Among the four kinds of events, the change event can be expressed as a predicate. Hence, we apply the state-based specification test case generation model described in Chapter 2 to *enabled transitions* with *change events*.

3.1.1 Transition Coverage Level

Transition coverage:	**Each enabled transition in the statechart diagram is taken at least once.**

3.1.2 Full Predicate Coverage Level

Full predicate coverage:	**Each clause in turn takes the values of `True` and `False` while all other clauses in the predicate have values such that the value of the predicate will always be the same as the clause being tested.**

3.1.3 Transition-Pair Coverage Level

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.**

Note: The transition-pair coverage criterion may not be feasible in a statechart that has mixed types of transitions. The technique generates test data for only enabled transitions.

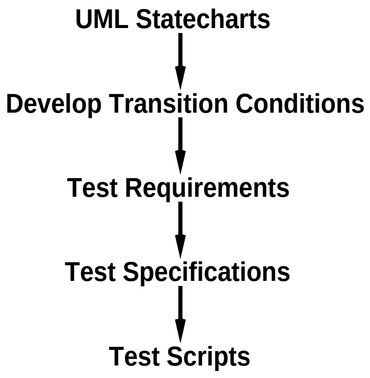


Figure 11: **General Process for Generating Test Cases**

If there are others types of transitions in a statechart, the technique cannot generate test cases for them. We may not have test cases for some of the transition pairs.

3.1.4 Complete Sequence Level

Complete sequence level:	**The test engineer must define meaningful sequences of transitions on the statechart diagram by choosing sequences of states that should be entered.**

Note: The complete sequence coverage criterion may not be feasible in a statechart that has mixed types of transitions. The reason is same as for the transition-pair coverage criterion.

3.2 Derivation Process

This section presents a process that can be used to derive test cases. Figure 11 shows the general process for generating test cases.

1. **Develop transition conditions.** The first step is to develop **transition conditions**, which are predicates that define under what conditions each transition will be taken. In UML, the transition conditions are encoded directly into the specifications.

2. **Develop transition coverage test requirements.**

    (a) **Derive transition predicates.** The conditions from step 1 are listed one at a time to form test requirements.

3. **Develop full predicate test requirements.**

    (a) **Construct truth tables for all predicates in the specification graph.** The predicate coverage tests can be based on an expression tree or directly on the predicates. If all the logical connectors are the same (all ANDs or all ORs), it is a simple matter to modify the values for the clauses in the predicates directly. If ANDs and ORs are mixed freely, however, it is less error-prone to construct the expression tree. Most specification languages differentiate between trigger events and preconditions; in this case, the trigger events must be marked specially so that the test engineer remembers to put that input after the precondition inputs.

4. **Develop transition-pair test requirements.**

    (a) **Identify all pairs of transitions.** Transition-pair tests are ordered pairs of condition values, each representing an input to the state and an output from the state. These are formed by enumerating all the input transitions (M), all the output transitions (N), then creating $M * N$ pairs of transitions.

    (b) **Construct predicate pairs.** These pairs of transitions are then replaced by the predicates from the specification graph.

5. **Develop complete sequence test requirements.**

   (a) **Identify complete lists of states.** The complete sequence tests are created by the tester. This is done by choosing sequences of states from the specification graph to enter.

   (b) **Construct sequence of predicates.** The sequences of states are transformed into sequences of conditions that will cause those states to be entered.

   At this point, test requirements for the four levels will be in a uniform format, as truth assignments for predicates.

6. **Construct test specifications.** For each unique test requirement, generate prefix values, test case values, verify conditions, exit conditions, and expected outputs. Note that there may be a fair amount of overlap among the test requirements, thus the "unique" restriction. Generating the actual values may involve solving some algebraic equations. For example, if a condition is $A > B$, values for A and B must be chosen to give the predicate the appropriate value. It is also at this point that some "invalid" tests might be discovered. For example, it may be impossible or meaningless to pair all incoming and outgoing transitions for each state. In this case, certain test specifications will be discarded.

7. **Construct test scripts.** Each test specification is used to construct one test script. The actual scripts must reflect the input syntax of the program, so knowledge of the input syntax of the program is required for this step. (Note that this is the only step that requires any knowledge of the implementation, all preceding steps depend solely on the functional specifications.)

3.2.1 Automation Notes

The UML specifications generated by Rational Rose case tool are all saved in plain text format, which are readable on any machine. Each object is an instant of a class. A class part of UML specifications has class names, class attributes, class methods, and a state machine attached to it. An object's states and transitions can be gotten directly from the class section of a specification. Since the variables used in state transition descriptions are defined in the class attributes section, the types and initial values of variables can be obtained. With an appropriate data structure for the obtained information and algorithms for the model, the test case generation process is automatable.

3.3 Cruise Control Example

This section presents an example of applying the test data generation model to the UML specification of automobile cruise control system. Figure 12 shows the UML statechart for cruise control object. It has four states: OFF (the initial state), INACTIVE, CRUISE, and OVERRIDE. The objects attributes 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*.

The transitions are shown as solid arrows from one state (the *source* state) to another state (the *target* state) labeled by a *transition string.* Each transition string specifies a conditioned event that activates the transition from the source state to the target state. A transition string `when(c)` means the value of `c` must change from *false* to *true* for the transition to be taken. A transition string `when(not c)` means the value of `c` must change from *true* to *false* for the transition to be taken. A transition string `when(c) [d]` means d must be evaluated

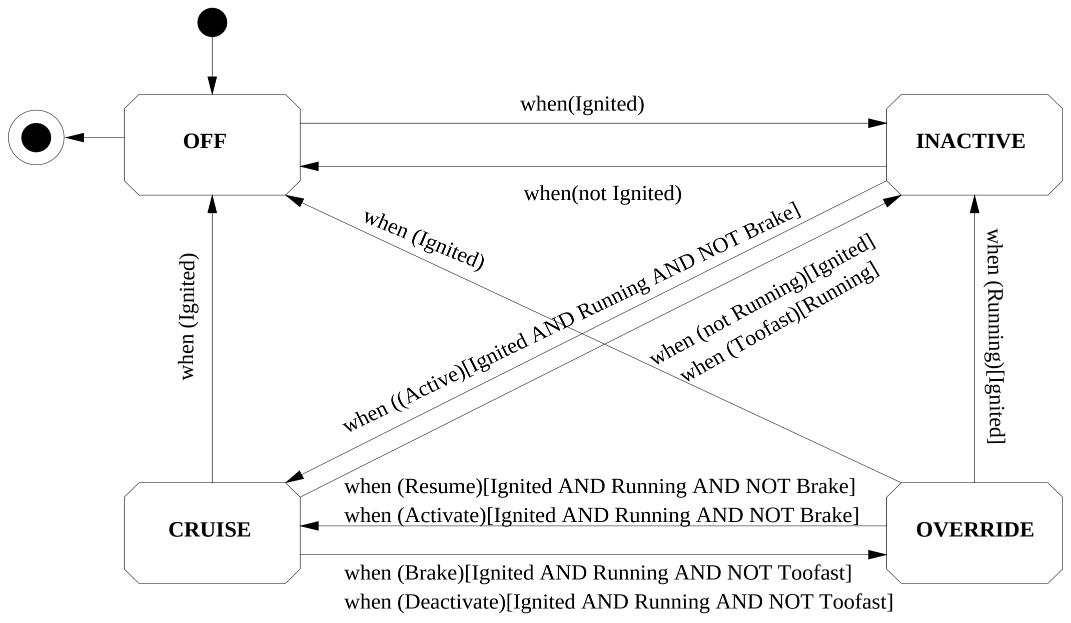

OFF
INACTIVE
CRUISE
OVERRIDE
when(Ignited)
when(not Ignited)
when (Ignited)
when (Ignited)
when ((Active)[Ignited AND Running AND NOT Brake]
when (not Running)[Ignited]
when (Toofast)[Running]
when (Running)[Ignited]
when (Resume)[Ignited AND Running AND NOT Brake]
when (Activate)[Ignited AND Running AND NOT Brake]
when (Brake)[Ignited AND Running AND NOT Toofast]
when (Deactivate)[Ignited AND Running AND NOT Toofast]

Figure 12: **UML Statechart for Cruise Control**

to *true* before and at the time `c` is changed from *false* to *true*. A transition string `when(c) [not d]` means `d` must be evaluated to *false* before and at the time `c` changed from *false* to *true*. Other combinations can be inferred from the above descriptions.

The conventions used in SCR method for event description are simple and visual, so we describe the transitions with those conventions. The `when(c)` part of the transition is mapped as @T(c), `when(not c)` is mapped as @F(c). The guard(s) `[d]` are mapped directly to `when` conditions.

Table 4 shows the transitions of the specification in predicate form, numbered P_1 through P_{12}.

From the table we can see that now we have exactly same table as we have from SCR specification of cruise control system; the rest of the process same as in Chapter 2.

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

Table 4: **Cruise Control Specification Predicates**

4 A PROOF OF CONCEPT TOOL

TCGen is a proof-of-concept tool that generates test cases from SCR and UML specifications. The SCR and UML specifications that TCGen can process are case tool specific. The SCR specifications that we considered in TCGen design are generated by SCR* Toolset [JK98], which is developed by the Naval Research Laboratory. The UML specifications were generated by Rational Software Corporation's Rational Rose, hereafter Rose.

In this chapter, we first describe the structure of the SCR and UML specification files and assumptions that we made during our design and implementation of TCGen. Then we give the architectural design for the TCGen. Finally, the algorithms that parse the specification files and generate test cases for full-predicate coverage and transition-pair coverage criteria are presented.

4.1 SCR and UML Specification Files

Both SCR* Toolset and Rose have graphical user interfaces. SCR* Toolset has consistency and type checking functions that enforce consistency of names and types among various kinds of elements of a specification. Rose cannot enforce consistency, but it has well-defined notations for all possible elements, that is, if followed, a consistent specification can be produced. Meanwhile, in both tools the specifications are saved as ASCII text files. This provided another possibility for automatically generating test cases from specification. We parse the specification file syntax to get its semantic meaning. Parsing the specification text files heavily depends on their structure. In this section, we will give a brief overview of how the specification text files were structured.

4.1.1 Structure of SCR Specification Files

SCR* Toolset supports the specification of the following items individually:

- Type Dictionary
- Mode Class Dictionary
- Constant Dictionary
- Variable Dictionary
- Specification Assertion Dictionary
- Environmental Assertion Dictionary
- Enumerated Monitored Variable Dictionary
- Controlled Variable Dictionary
- Mode Class Tables
- Term Variable Tables

The specifications for all items are saved as ASCII text files in the above order. There is no restriction on the file name, or on the extension. The structure of the text file is shown in Figure 13.

4.1.2 Assumptions for SCR Specifications

The following assumptions were made while parsing the SCR specification text file:

- @T, @F denote trigger events
- AND denotes logical and
- Only one mode class
- boolean variables
- Single variable change in event
- None/Single/Multiple variables in condition
- State transitions are deterministic

Type Dictionary

TYPE	*Type Name*
BASETYPE	*Base Type Name*
UNITS	*Unit Name*
COMMENT	*Comments for the type usage*

Mode Class Dictionary

MODECLASS	*Mode Class Name*
MODES	*List of modes separated by comma*
INITMODE	*Initial Mode*
COMMENT	*Comments for the mode class usage*

Constant Dictionary

CONSTANT	*Constant Name*
TYPE	*Type Name*
VAL	*Value*
COMMENT	*Comments for the constant*

Variable Dictionary

MON	*Name of a monitored variable*
TYPE	*Type Name*
INITVAL	*Initial value*
ACCURACY	*Accuracy*
COMMENT	*Rules for value assignment*

CON	*Name of a controlled variable*
TYPE	*Type Name*
INITVAL	*Initial value*
ACCURACY	*Accuracy*
COMMENT	*Rules for value assignment*

Specification Assertion Dictionary

ASSERTION	*Name of an assertion*
EXPR	*Expression*
COMMENT	*Explanation of assertion*

Figure 13: **The Structure of SCR Specification Text File**

Environmental Assertion Dictionary

Enumerated Monitored Variable Table

Event, Mode Transition, and Condition Functions

`EVENTFUNC`	*Event function table name*
`MCLASS`	*Mode class name*
`MODES`	*Mode name*
`EVENTS`	*Event1, Event2*
`ASSIGNMENTS`	*Value1, Value2*
`CONDFUNC`	*Condition function table name*
`CONDITIONS`	*Condition1, Condition2*
`ASSIGNMENTS`	*Value1, Value2*
`MODETRANS`	*Mode transition table name*
`FROM`	*State name*
`EVENT`	*Event*
`WHEN`	*List of Disjunctive Conditions*
`TO`	*State name*

Figure 13: **The Structure of SCR Specification Text File - continued**

4.1.3 Structure of UML MDL Files

UML specification text files generated by Rose are generally called MDL files because of the file extension "mdl". We use this convention in this thesis.

The MDL files store specification information from different perspectives. There are two main categories of information: logical and physical. The specification itself is grouped into two packages: *use case* and *object collaboration diagrams* are packed as *Use Case Package*, *Class Diagram* and *State Transition Diagram* are packed as *Logical View*. We are interested in state transition diagrams, hence, we give only the structure of the *Logical View* section of the MDL file. Figure 14 shows the internal structure of *class diagram* and *state transition diagram* in a MDL file.

```
Logical Models
    object Class
        classAttributes

        ------------- State Transition:  Logical -------------


        State Machine
            Object State /*StartState, Normal, EndState */
                State Transition
                State Machine
                    Object State /* Normal */

        ------------- State Transition:  Physical ------------


        State Diagram
            State View /* StartState, Normal, EndState */
            Transition View

    object Association
        object Role
Logical Presentations
    object ClassDiagram /* with grouping and name */
        object ClassView
            Association View
            Role View
            Inheritance View
```

Figure 14: **Structure of MDL File for Class Diagram and State Transition Diagram Section**

4.1.4 Assumptions for UML Specifications

As stated in Chapter 3, several types of transitions can be specified in UML. Because in this thesis we are interested only in transitions triggered by change events, we did not consider other types of transitions. For the UML specification file input of TCGen, we made the following assumptions:

- All transitions are triggered by change events.
- Events and conditions are expressed through boolean type class attributes.
- The specification is written strictly following the UML notations. For example, *when* denotes a change event, conditions are in solid brackets ([]), etc. Because there is no way to check whether a specification is well-formed or consistent, this assumption cannot be checked. The OCL does not have mechanism to enforce its well-formedness rules on all parts of the UML specification. Also, Rose does not have a function to write the specification in OCL.
- State transitions are deterministic.

4.2 Architectural Design

We explain the design model of the tool through a *class diagram* and three *object collaboration diagrams* generated by Rose.

4.2.1 Class Diagram

Figure 15 is a UML class diagram describing the TCGen tool. Classes are represented as boxes that have three parts, the class name, data members that are declared in the class, and methods of the class. The main entry point (TCGen) has four objects, a UML specification parser, a SCR specification parser, a full predicate test case generator, and a transition-pair test case generator.

UMLSpecParser reads a UML specification text file, parses it, and generates state transition table(s) for classes that have state machines. SCRSpecParser reads SCR specification text files, parses them, and generates state transitions tables for mode classes. FullPredicate takes a state transition table as an input, generates test cases for the full predicate coverage criterion, and saves the test cases into a file. TransitionPair takes a state transition table as input, generates test cases for the transition-pair coverage criterion, and saves the test cases in an ASCII text file.

4.2.2 Object Collaboration Diagrams

Object collaboration diagrams (OCD) for generating full predicate coverage, transition coverage, and transition-pair coverage test cases are shown in Figures 16, 17, and 18. We illustrate the OCD for generating test cases for full predicate coverage, the two figures have similar explanation.

In Figure 16, TCGen is a main program. It interacts with the user, gets the command to read a SCR or UML specification file, and invokes SCRSpecParser or UMLSpecParser sending a specification file name as a parameter. SCRSPecParser or UMLSpecParser opens the file, parses it, generates transition table, and returns the transition table to TCGen. Next, TCGen invokes *FullPredicate*, sending the state transition table as a parameter. *FullPredicate* generates test cases for full predicate coverage criteria, saves the test cases in files, and returns a message to TCGen that test cases have been generated.

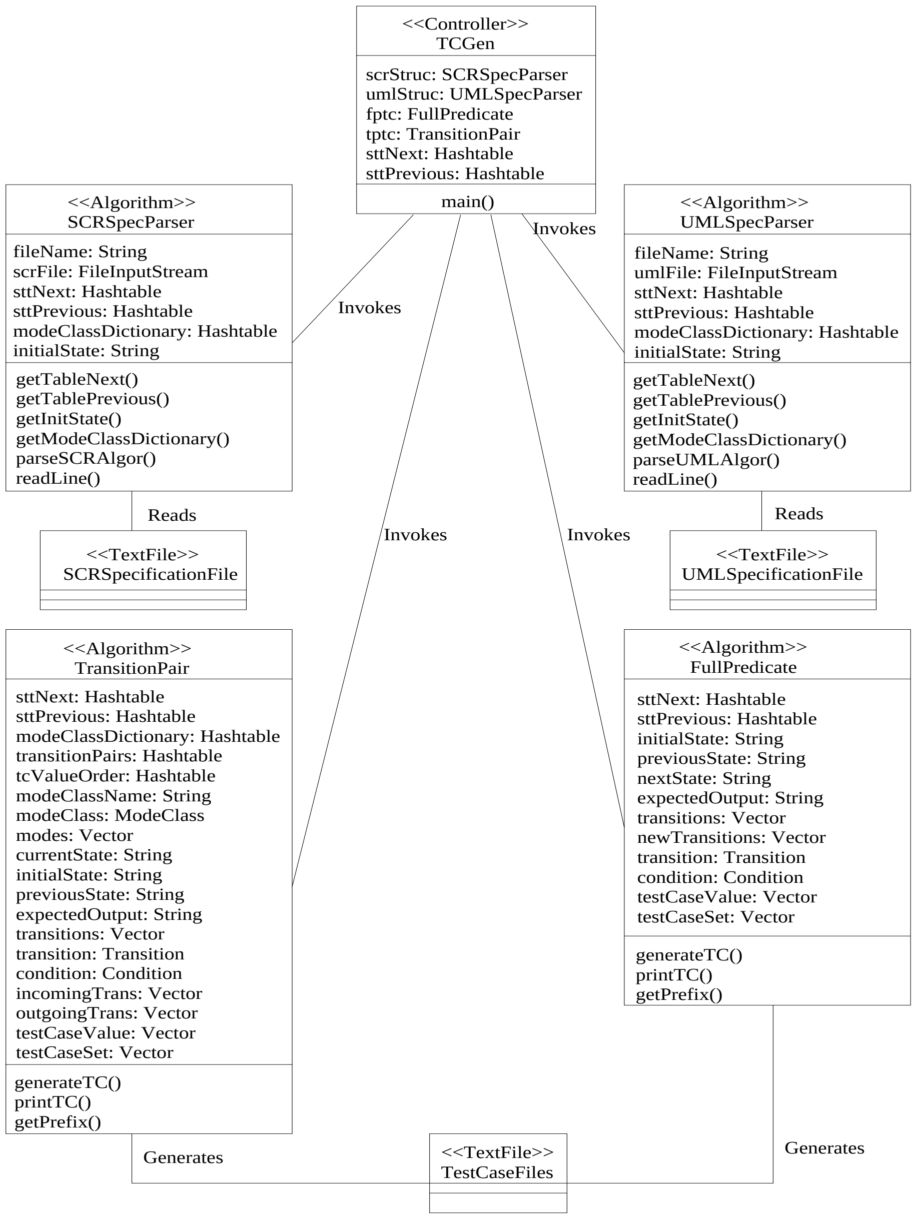


Figure 15: **Class Diagram for TCGen Tool**

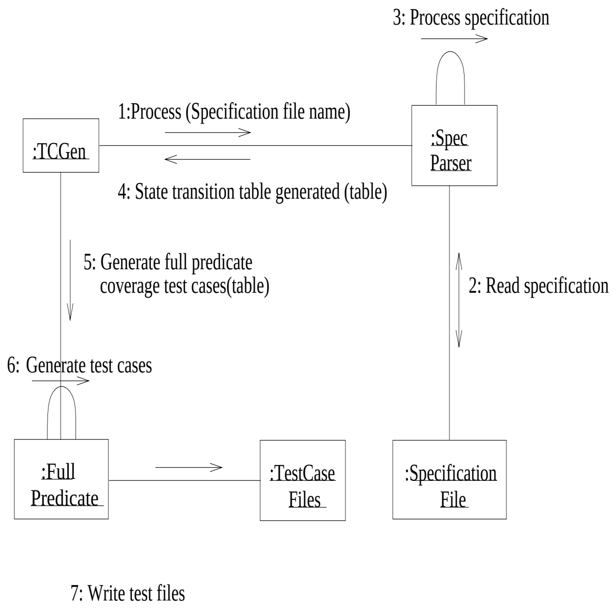


Figure 16: **OCD for Generating Full Predicate Coverage Test Cases**

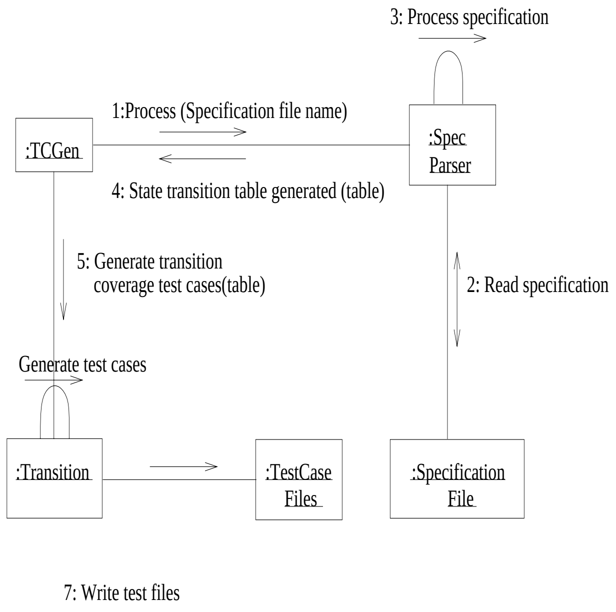


Figure 17: **OCD for Generating Transition Coverage Test Cases**

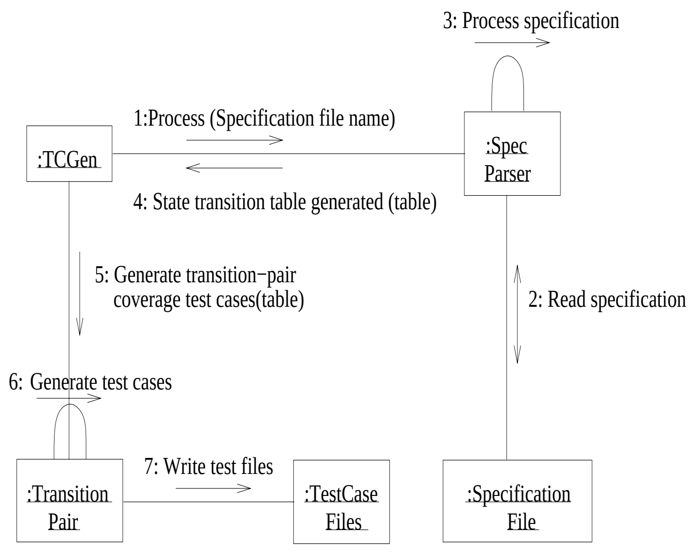


Figure 18: **OCD for Generating Transition-Pair Coverage Test Cases**

4.3 Algorithms

This section introduces algorithms used in TCGen. Algorithms were developed to parse SCR and UML specification text files and to generate test case for *full predicate coverage*, *transition coverage*, and *transition-pair coverage* criteria. A *prefix generation* algorithm was used in test data generation algorithms.

4.3.1 Parse SCR Specification Algorithm

Figure 19 gives an algorithm to parse SCR specification text files. Algorithm *SCRSpec-Parser* takes an SCR specification file as an input, then parses the file to extract the necessary information for test case generation. It extracts *mode class dictionary*, *variable dictionary*, and *state transition table*, and saves them in data structures.

```
algorithm:         ParseSCRSpecification (SCR Specification)
input:             SCR specification text file.
output:            State transition table.
output criteria:   Find all incoming and outgoing transitions for each state.


ParseSCRSpecification (SCR Specification)
BEGIN -- Algorithm ParseSCRSpecification
    -- Read until "Mode Class Dictionary" section starts
    WHILE (curLine DOES NOT CONTAIN "Mode Class Dictionary") LOOP
        curLine = NEW LINE
    END LOOP
    PARSE "Mode Class Dictionary" section; EXTRACT All mode class names
    -- Read until "Variable Dictionary" section starts
    WHILE (Currentline DOES NOT CONTAIN "Variable Dictionary") LOOP
        curLine = NEW LINE
    END LOOP
    PARSE "Variable Dictionary" section; EXTRACT All variable names,
    types, initial value, and accuracy.
    -- Read until "Event, mode transition, condition" section starts
    WHILE (Currentline DOES NOT CONTAIN "Variable Dictionary") LOOP
        curLine = NEW LINE
    END LOOP
    PARSE "Event, mode transition, condition" section; EXTRACT All previous
    states, next states, transition predicates.
    RETURN (Finished)
END Algorithm ParseSCRSpecification
```

Figure 19: **The ParseSCRSpecification Algorithm**

4.3.2 Parse UML Specification Algorithm

Figure 20 gives an algorithm to parse UML specification text files (MDL files). Algorithm *UMLSpecParser* takes a UML specification file as an input, then parses the file to extract the necessary information for test case generation. It extracts *class name*, *class attributes*, and *state machine*, and saves them in data structures. In this algorithm, UML classes are mapped to mode classes of SCR, class attributes to variable dictionaries, and state machines to state transition tables.

4.3.3 Generate Transition Coverage Test Cases Algorithm

Figure 21 gives an algorithm to generate test cases for the *transition coverage* criterion. This algorithm uses an algorithm to find a prefix for a given state. The **GetPrefix** algorithm is shown in Figure 22. Algorithm *GenerateTransitionCoverageTCs* takes a *StateTransitionTable* as input, and generates a test case for each of the outgoing transition of each source state in the table. Although we developed the *GenerateTransitionPairTCs* algorithm for the *transition pair coverage* criterion, we did not implement it in our tool TCGen because the *transition-pair coverage* criterion subsumes the *transition coverage* criterion. A criterion A *subsumes* criterion B if for every test set T that satisfies A, T also satisfies B.

4.3.4 Generate Full-Predicate Coverage Test Cases Algorithm

Figure 23 gives an algorithm for generating test cases for the *full predicate coverage* criterion. Algorithm *FullPredicateCoverageTCs* takes a *state transition table* as input, and generates test cases for the full predicate coverage criterion. It processes each outgoing transition of each source state, generates a test case that makes the transition valid, and then generates test cases that make the transition invalid. When generating a test case, a *prefix* is gotten

```
algorithm:        ParseUMLSpecification (UMLSpecificationFile)
input:            UML specification text file.
output:           State transition table.
output criteria:  Find all incoming and outgoing transitions for each state.
declare:          statemachine -- A finite state machine (FSM) that describes the
                  behavior of a class.
                  varDic -- The dictionary created for a UML class attributes.
                  outgoingTrans(s) -- The set of outgoing transitions from state s.
                  sttNext -- A table that is for outgoing transitions in the FSM.
                  sttPrev -- A table that is for incoming transitions in the FSM.
                  otr -- An outgoing transition that being processed.


ParseUMLSpecification (UMLSpecificationFile)
BEGIN -- Algorithm ParseUMLSpecification
    -- Read until "object Class category 'Logical View'" section starts
    varDic = EMPTY
    FOR EACH "object Class" section
        IF (EXISTS attributes) THEN
            WHILE (HAS MORE class attributes) LOOP
                get attribute name, type, and initial value
                varDic = varDic ∪ {(attribute name, type, initial value)}
            END LOOP
        END IF
        IF (EXISTS statemachine) THEN
            sttNext = EMPTY
            sttPrev = EMPTY
            FOR EACH state s in statemachine
                prevState = s
                get outgoingTrans(s)
                FOR EACH outgoing transition otr ∈ outgoingTrans(s)
                    get eventType
                    get eventParam
                    get condition
                    get nextState
                    sttNext = sttNext ∪ {(prevState, eventType, eventParam, condition, nextState)}
                    sttPrev = sttPrev ∪ {(nextState, eventType, eventParam, condition, prevState)}
                END FOR
            END FOR
        END IF
    END FOR
END Algorithm ParseUMLSpecification
```

Figure 20: **The ParseUMLSpecification Algorithm**

```
algorithm:         GenerateTransitionCoverageTCs (StateTransitionTable)
input:             State transition table.
output:            Test cases for transition coverage criterion.
output criteria:   Each test case consists of prefix, test case values, and expected output.
                   No redundant assignment in prefix and test case values.
declare:           prefix(s) -- Inputs to get to the state s.
                   outgoingTrans(s) -- The set of outgoing transitions from state s.
                   otr -- An outgoing transition that being processed.
                   event(otr) -- The trigger event for transition otr.
                   whenCondition(otr) -- The precondition for transition otr be enabled.
                   nextState(otr) -- The next state for transition otr.
                   expectedOutput -- The post-state of FSM after a transition take place.
                   TCValue(otr) -- Value assignments for the trigger event and when
                   condition variables for transition otr.
                   TestCaseSet -- The whole set of test cases for transition coverage citerion.


GenerateTransitionCoverageTCs (StateTransitionTable)
BEGIN -- Algorithm GenerateTransitionCoverageTCs
    TestCaseSet = EMPTY
    FOR EACH source state s in StateTransitionTable
        prefix(s) = GetPrefix(s)
        get outgoingTrans(s)
        -- Generate one test case for each transition
        FOR EACH outgoing transition otr ∈ outgoingTrans(s)
            expectedOutput = nextState(otr)
            TCValue(otr) = EMPTY
            get event(otr) and whenConditions(otr)
            -- Check for redundancy before assign a value to a variable
            IF (¬ ∃ a condition variable var ∈ prefix(s) such that
            var.name = event(otr).name ∧ var.value = event(otr).value) THEN
                TCValue(otr) = TCValue(otr) ∪ {(event(otr).name, event(otr).beforeValue)}
            END IF
            -- Assign value for clauses in when condition
            FOR EACH clause_i in whenConditions(otr)
                IF (¬ ∃ a condition variable var ∈ prefix(s) such that
                var.name = clause_i.name ∧ var.value = clause_i.value) THEN
                    TCValue(otr) = TCValue(otr) ∪ {(clause_i.name, clause_i.value)}
                END IF
            END FOR
            TCValue(otr) = TCValue(otr) ∪ {(event(otr).name, event(otr).afterValue)}
            TestCaseSet = TestCaseSet ∪ {(prefix(s), TCValue(otr), expectedOutput}
        END FOR
    END FOR
END Algorithm GenerateTransitionCoverageTCs
```

Figure 21: **The GenerateTransitionCoverageTCs Algorithm**

```
algorithm:         GetPrefix (State)
input:             The source state of a transition that is being processed.
output:            Inputs to get to the given state.
output criteria:   No redundant input.
declare:           prefix(s) -- Inputs to get to the state s.
                   incomingTrans(s) -- The set of incoming transitions into state s.
                   itr -- An incoming transition that is being processed.
                   event(otr) -- The trigger event for transition otr.
                   whenCondition(otr) -- The precondition for transition otr to be enabled.
                   nextState(otr) -- The next state for transition otr.
                   expectedOutput -- The post-state of FSM after a transition takes place.
                   TCValue(otr) -- Value assignments for the trigger event and when
                   condition variables for transition otr.
                   TestCaseSet -- The whole set of test cases for transition coverage.


GetPrefix (State)
BEGIN -- Algorithm GetPrefix
    s = state
    prefixStates = prefixStates ∪ s
    WHILE (s IS NOT initial state) LOOP
    get incomingTrans(s)
    prefix(s) = EMPTY
        IF (∃ transition itr ∈ incomingTrans(s) such that
        prevState(itr) = initialState) THEN
            s = prevState(itr)
            prefixStates = prefixStates ∪ s
            EXIT
        ELSE
            s = prevState(itr) such that itr ∈ incomingTrans(s) ∧
                prevState(itr) ∉ prefixStates
            prefixStates = prefixStates ∪ s
        END IF
    END LOOP
END Algorithm GetPrefix
```

Figure 22: **The GetPrefix Algorithm**

first to reach the source state of a transition. Then each variable in the transition predicate is assigned a test case value. To avoid redundant test case value assignments, those variables that already have assigned values in the prefixes are not considered in the test case value assignment process.

4.3.5 Generate Transition-Pair Coverage Test Cases Algorithm

Figure 24 gives an algorithm for generating test cases for the *transition-pair coverage* criterion. Algorithm *GenerateTransitionPairTCs* takes *StateTransitionTable* as an input, then generates test cases for each pair of incoming-outgoing transitions from each source state. *GenerateTransitionPairTCs* also uses the *GetPrefix* algorithm to get the input that is necessary to put the system in the source state of an incoming transition.

```
algorithm:          GenerateFullPredicateCoverageTCs (StateTransitionTable)
input:              State transition table.
output:             Test cases for full predicate coverage.
output criteria:    Each test case consists of prefix, test case values, and expected output.
assumption:         Clauses in a predicate are disjunctive.
                    No redundant assignment in prefix and test case values.
declare:            prefix(s) -- Inputs to get to the state s.
                    outgoingTrans(s) -- The set of outgoing transitions from state s.
                    otr -- An outgoing transition that is being processed.
                    event(otr) -- The trigger event for transition otr.
                    whenCondition(otr) -- The precondition for transition otr to be enabled.
                    nextState(otr) -- The next state for transition otr.
                    expectedOutput -- The post-state of FSM after a transition takes place.
                    TCValue(otr) -- Value assignments for the trigger event and when
                    condition variables for transition otr.
                    TestCaseSet -- The whole set of test cases for transition coverage.


GenerateFullPredicateCoverageTCs (StateTransitionTable)
BEGIN -- Algorithm GenerateFullPredicateCoverageTCs
    TestCaseSet = EMPTY
    FOR EACH source state s in StateTransitionTable
        prefix(s) = GetPrefix(s)
        get outgoingTrans(s)
        -- Generate one test case for each transition
        FOR EACH outgoing transition otr ∈ outgoingTrans(s)
            expectedOutput = nextState(otr)
            TCValue(otr) = EMPTY
            get event(otr) and whenConditions(otr)
            -- Check for redundancy before assign a value to a variable
            IF (¬ ∃ a condition variable var ∈ prefix(s) such that
            var.name = event(otr).name ∧ var.value = event(otr).value) THEN
                TCValue(otr) = TCValue(otr) ∪ {(event(otr).name, event(otr).beforeValue)}
            END IF
            -- Assign value for clauses in when condition
            FOR EACH clause_i in whenConditions(otr)
                IF (¬ ∃ a condition variable var ∈ prefix(s) such that
                var.name = clause_i.name ∧ var.value = clause_i.value) THEN
                    TCValue(otr) = TCValue(otr) ∪ {(clause_i.name, clause_i.value)}
                END IF
            END FOR
            TCValue(otr) = TCValue(otr) ∪ {(event(otr).name, event(otr).afterValue)}
            TestCaseSet = TestCaseSet ∪ {(prefix(s), TCValue(otr), expectedOutput}
```

Figure 23: **The GenerateFullPredicateCoverageTCs Algorithm**

```
            -- get test cases for invalid transitions
            expectedOutput = current state s
            **FOR EACH** variable var in TCValue(otr)
                TCValue(otr) = TCValue(otr) - {(var.name, var.value)}
                var.value = ¬var.value
                TCValue(otr) = TCValue(otr) ∪ {(var.name, var.value)}
                TestCaseSet = TestCaseSet ∪ {(prefix(s), TCValue(otr), expectedOutput)}
            **END FOR**
        **END FOR**
    **END FOR**
**END Algorithm GenerateFullPredicateCoverageTCs**
```

Figure 23: **The GenerateFullPredicateCoverageTCs Algorithm - continued**

```
algorithm:         GenerateTransitionPairCoverageTCs (StateTransitionTable)
input:             State transition table.
output:            Test cases for full predicate coverage criterion.
output criteria:   Each test case consists of prefix, test case values, and expected output.
assumption:        Clauses in a predicate are disjunctive.
                   No redundant assignment in prefix and test case values.
declare:           prefix(ss) -- Inputs to get to the state ss.
                   outgoingTrans(s) -- The set of outgoing transitions from state s.
                   incomingTrans(s) -- The set of incoming transitions to state s.
                   i/otr -- An incoming/outgoing transition that is being processed.
                   event(i/otr) -- The trigger event for transition i/otr.
                   whenCondition(i/otr) -- The precondition for transition i/otr to be enabled.
                   prevState(itr) -- The source state of transition itr.
                   nextState(otr) -- The target state for transition otr.
                   expectedOutput -- The post-state of FSM after a transition takes place.
                   TCValue(itr) -- Value assignments for the trigger event and when
                   condition variables for transition itr.
                   TCValue(otr) -- Value assignments for the trigger event and when
                   condition variables for transition otr.
                   TestCaseSet -- The whole set of test cases for transition-pair coverage.


GenerateTransitionPairCoverageTCs (StateTransitionTable)
BEGIN -- Algorithm GenerateTransitionPairCoverageTCs
    TestCaseSet = EMPTY
    FOR EACH state s in StateTransitionTable
        get incomingTrans(s)
        get outgoingTrans(s)
        FOR EACH incoming transition itr $\in$ incomingTrans(s)
            ss = prevState(itr)
            prefix(ss) = GetPrefix(ss)
            TCValue(itr) = EMPTY
            get event(itr) and whenConditions(itr)
            -- Check for redundancy before assign a value to a variable
            IF ($\neg$ $\exists$ a condition variable var $\in$ prefix(ss) such that
            var.name = event(itr).name $\wedge$ var.value = event(itr).value) THEN
                TCValue(itr) = TCValue(itr) $\cup$ {(event(itr).name, event(itr).beforeValue)}
            END IF
            -- Assign value for clauses in when condition
            FOR EACH clause$_i$ in whenConditions(itr)
                IF ($\neg$ $\exists$ a condition variable var $\in$ prefix(ss) such that
                var.name = clause$_i$.name $\wedge$ var.value = clause$_i$.value) THEN
                    TCValue(itr) = TCValue(itr) $\cup$ {(clause$_i$.name, clause$_i$.value)}
                END IF
            END FOR
            TCValue(itr) = TCValue(itr) $\cup$ {(event(itr).name, event(itr).afterValue)}
```

Figure 24: **The GenerateTransitionPairCoverageTCs Algorithm**

```
        FOR EACH outgoing transition otr $\in$ outgoingTrans(s)
            TCValue(otr) = EMPTY
            expectedOutput = nextState(otr)
            get event(otr) and whenConditions(otr)
            -- Check for redundancy before assign a value to a variable
                IF ($\neg$ $\exists$ a condition variable var $\in$ prefix(ss) such that
            var.name = event(otr).name $\wedge$ var.value = event(otr).value) THEN
                TCValue(otr) = TCValue(otr) $\cup$ $\{$(event(otr).name, event(otr).beforeValue)$\}$
            END IF
            -- Assign value for clauses in when condition
            FOR EACH clause$_i$ in whenConditions(otr)
                IF ($\neg$ $\exists$ a condition variable var $\in$ prefix(ss) such that
                var.name = clause$_i$.name $\wedge$ var.value = clause$_i$.value) THEN
                    TCValue(otr) = TCValue(otr) $\cup$ $\{$(clause$_i$.name, clause$_i$.value)$\}$
                END IF
            END FOR
            TCValue(otr) = TCValue(otr) $\cup$ $\{$(event(otr).name, event(otr).afterValue)$\}$
        TestCaseSet = TestCaseSet $\cup$ $\{$(prefix(ss), TCValue(itr) $\cup$ TCValue(otr), expectedOutput$\}$
        END FOR
    END FOR
END FOR
```

Figure 24: **The GenerateTransitionPairCoverageTCs Algorithm - continued**

5 RESULTS

A principal reason for implementing TCGen was to provide a tool for generating test cases automatically to improve the specification-based testing process. Experiments have been performed on SCR and UML specifications to determine the test case generation power of TCGen. Results for different coverage criteria are presented. A distribution of detected faults is given. The time to generate test cases is also presented.

The tool currently handles only one mode class in SCR specifications and one class in UML specifications. The transition predicates are limited to predicates that have only boolean clauses. The results focus on two questions: (1) can generating test cases from specifications be automated, and (2) what is the quality of test cases that are generated for the full predicate and transition-pair coverage criteria?

For the first question, we generated SCR and UML versions of specifications for the cruise control system. The specifications were generated by SCRtool* and Rose respectively, and were saved in ASCII text files. TCGen was used to generate test cases for full predicate and transition-pair test cases from both specifications. TCGen generated 34 test cases for *full predicate coverage* criterion. This was surprising at first because the number of manually generated test cases was 54. This discrepancy happened because in the test case value part of the test cases, we eliminated those variables that are already assigned values in the prefix. This makes a difference for the *full predicate coverage* criterion because it considers each of the variables in the test case value part. For purposes of evaluation , we modified the tool and generated two sets of test cases for full predicate coverage criterion.

The first set includes test cases that included redundant value assignments, which resulted in 54 test cases, and the other had no redundant value assignments, which resulted in 34 test cases.

For the *transition-pair coverage* criterion, TCGen generated 34 test cases, which is the same as the number of hand generated test cases.

Test cases are saved in separate ASCII text files. These text files can be used as direct program inputs if the implementation follows the naming conventions of the specification. When the implementation has different naming notations for variables, a processor will be needed to translate the test case files into software inputs. Also generated are summary files that describe each test case in detail – for which transition predicate or transition-pair, which part is the prefix, and what is the expected output after running a test case.

For the second question, we executed 25 faulty versions of cruise control implementation on the different coverage criteria test sets. The faults were created by inserting variable reference, variable negation, expression negation, associative shift, and operator reference faults. Besides full predicate and transition-pair test case sets, we used *all-uses* and *branch coverage* criteria test sets. The branch coverage adequate test cases were manually generated by examining each branch in the *control flow graph* of the cruise control implementation. A *control flow graph* is a graph model of a program in which conditional branch instructions and program junctions are represented by nodes and the program segments between such points are represented by links [Bei90]. The process of branch coverage adequate test cases generation was first to insert instrumention into the cruise control implementation to measure branch coverage, then initial tests were created to set each variable to each value. Then each uncovered branch was used to derive a covering test case in an iterative fashion. There are 112 branches in the cruise control implementation, and 7 of them are infeasible.

The feasible 105 branches are covered by 9 test cases.

Each faulty implementation contains one fault. This way, it is convenient to monitor which test cases find each fault, and analyze the reasons. A series of scripts were written to run the test cases and collect the results.

The evaluation results are shown in Figures 25 through 35. Figures 25 through 29 show the number of faults that each test case found. Figures 30 through 34 show, for each fault, how many test cases found it. Figures 25 and 30 show the results for the full predicate coverage test cases that do not have redundant value assignments. Figures 26 and 31 show the results for the full predicate coverage test cases that include redundant value assignments. Both test sets achieved the same fault coverage, hence, we can conclude that removing redundant test value assignments reduces the number of test cases and also the size of some test cases without affecting the efficiency of a test case set. However, we further noticed that it is necessary to remove redundant test value assignments, especially when the variables involved in this process are invariants. Changes to the values of invariant variables must cause a transition from the current state to another state because of the violation of safety invariants. Therefore, if an invariant appears as an event variable in a transition, there is no problem. However, if an invariant appears in the condition part of a transition, changes to the invariant's value should cause an immediate state transition from the current state. There may be other variables that come after the invariant variable in a test case value, if these value assignments satisfy a transition predicate of the newly changed state, they will cause another state transition. Predicting the output of this kind of test case will be hard, especially in an automated environment. However, this problem was solved in TCGEN. A state's invariants must be true when the system is in that state. This means the prefix of a test case for a transition already includes the invariant variables of the source state of the transition. By not including any variables that are in the prefix of the test case values, the problem is avoided.

Figures 27 and 32 show the results for transition-pair test cases. Figures 28 and 33 show the results for the all-uses coverage test cases. Figures 29 and 34 show the results for branch coverage test cases.

As we can see from these figures of results, both full predicate redundant and full predicate non-redundant test cases reached 56% fault coverage. Hence, we conclude that for full predicate coverage, the second type of test cases should not be generated.

Figure 35 shows the fault coverage percentage for full predicate, transition-pair, all-uses and branch coverage criteria test cases. The fault coverage percentages are 56%, 48%, 64%, and 28% respectively.

For the 25 faults, the all-uses coverage test cases found the set of faults {1, 2, 3, 4, 5, 6, 7, 9, 13, 14, 16, 19, 20, 21, 23, 25}, the full predicate coverage test cases found the set of faults { 1, 2, 5, 6, 7, 9, 11, 12, 14, 17, 19, 20, 22, 25}, the transition-pair coverage test cases found the set of faults {1, 2, 5, 6, 7, 9, 14, 17, 19, 20, 22, 25}, and the branch coverage test cases found the set of faults {5, 7, 11, 17, 19, 22, 25}. Full predicate test cases found all the faults that were found by transition-pair and branch coverage test cases. The faults that were not found by full predicate test cases but all-uses coverage test cases are {3, 4, 13, 16, 21, 23}. After examining these faults, it was found that these are specific to how the program is run.

The implementation of cruise control system has three running modes: interactive mode, file input mode for test cases with prefix, and file input mode for test cases with source state names instead of a prefix. In the interactive mode, the variable value pairs are entered in order from the terminal. In the other two modes, the test cases are read from test case files. The running modes are not specified in the specification, therefore, the full predicate and transition-pair test cases do not reflect them. The all-uses coverage test cases were run in *file input mode with source state names*, and the full predicate and transition-pair test cases were run in *file input mode with test cases that have prefixes.* The branch coverage test cases were run in all modes.

After examining the faults that were not found by each set of test cases, the characteristics of the faults are sumarized as below:

- Some faults are specific to a running mode
- The choice of prefix path is relevant to finding fault
- Faults that are in an infeasable path cannot be revealed

It can be concluded from the above analysis that to have a complete test, test cases have to be run in every running mode. Besides, for each prefix, all different paths that can form the prefix should be considered. In full predicate and transition-pair coverage criteria, the prefix of a test case is a randomly chosen path from the initial state to the source state of a transition or transition-pair. I recommend to add the following requirement to the test case generation: All the paths that can form a prefix should be considered separately for a test case value.

We summarize the results as the following:

- Test case generation is fully automated.
- The outputs of TCGen were executed on the C implementation of the cruise control system. In this example, the implementation of cruise control used the same variable names that used in the specification. Therefore, the test cases that were generated from the specification were qualified to be direct inputs to the implementation. This is only a specific case, it does not hold in general. A middle processor will be needed

between the test cases and software inputs when the implementation does not use the names in the software specification.

- 25 faults were inserted into the cruise control implementation.
- The full predicate coverage test cases reached 56% coverage.
- The transition pair coverage test cases reached 48% coverage.
- The number of full predicate coverage criterion test cases is reduced by 37%. At full predicate coverage level, the test requirement is each clause in a transition predicate in turn takes the values of *True* and *False* while other clauses have values such that the value of the predicate will always be the same as the clause being tested. Since some of the clauses are already assigned values in the *prefix*, and keep those values as a state property, their values will not (or should not) be changed. Thus, the actual number of test cases is less than the calculated number of test cases from a predicate clauses. In other word, even if we include those clauses that should not be changed in our test case generation, the result is the same as with the reduced number of test cases.

- Test case generation time for a cruise control is less than 5 seconds for both full predicate and transition-pair coverage criteria. Comparing to the manual generation of test cases, this period of time is insignificant. The cruise control system has 12 transitions, which theoretically should result in 54 full predicated test cases and 34 transition-pair test cases. Manually generating test cases for these requirements took 16 to 20 hours. The branch coverage test cases used in this experimentation took 5 hours. This includes about 1 hour of instrumentation insertion time.

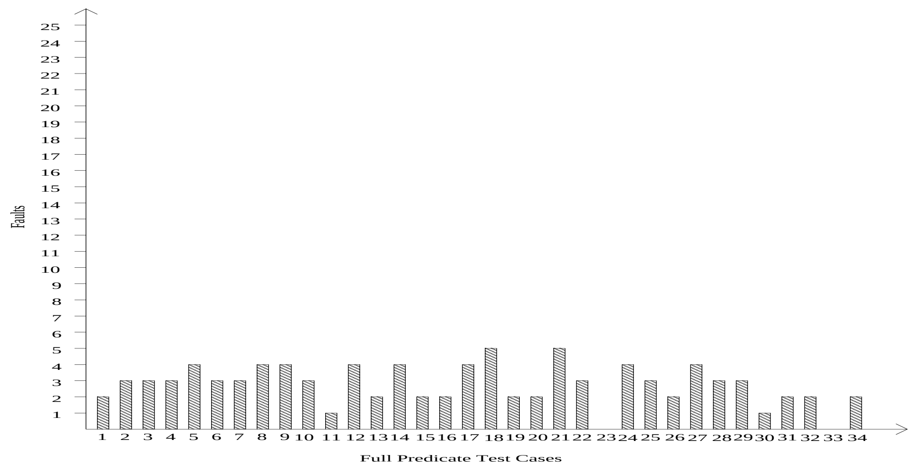


Figure 25: **Full Predicate Coverage Test Cases Results**

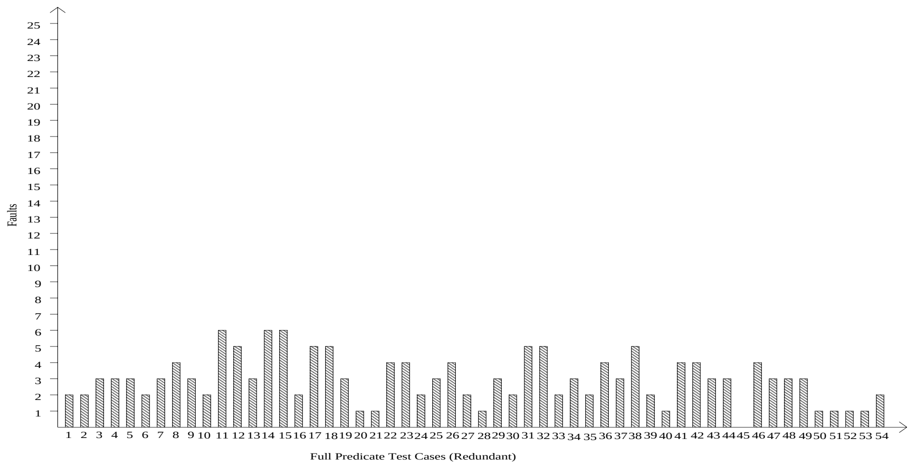


Figure 26: **Full Predicate Coverage Redundant Test Cases Results**

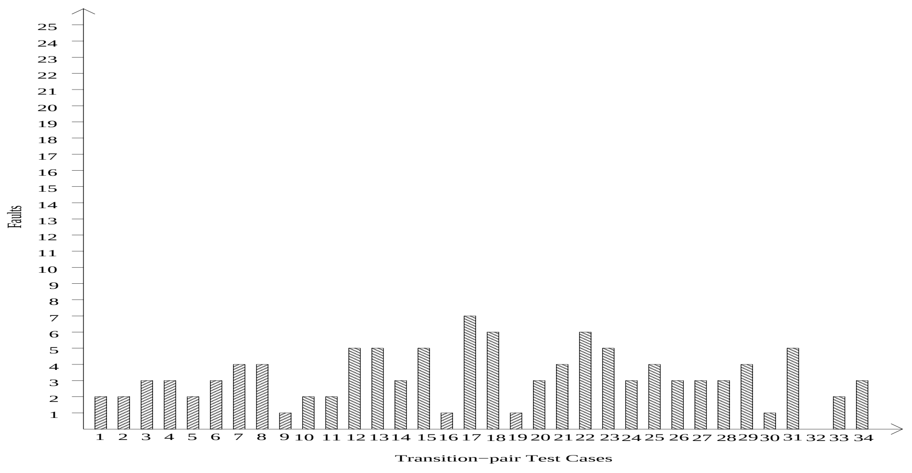


Figure 27: **Transition-pair Coverage Test Cases Results**

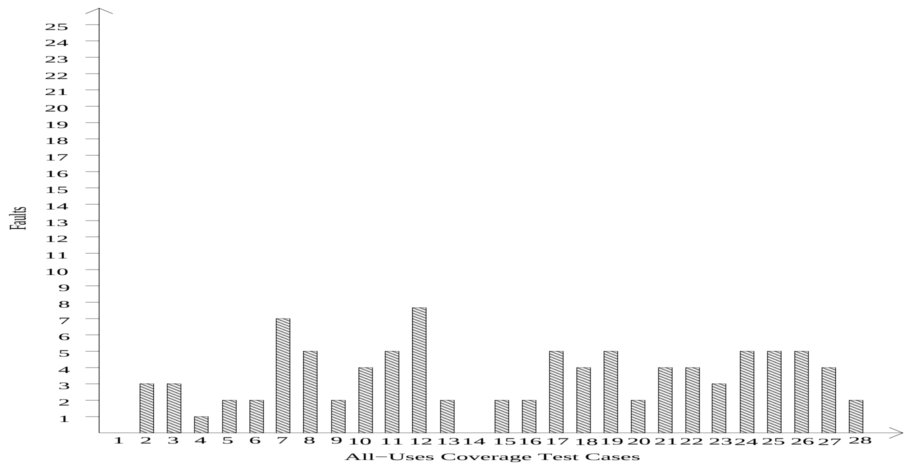


Figure 28: **All-Uses Coverage Test Cases Results**

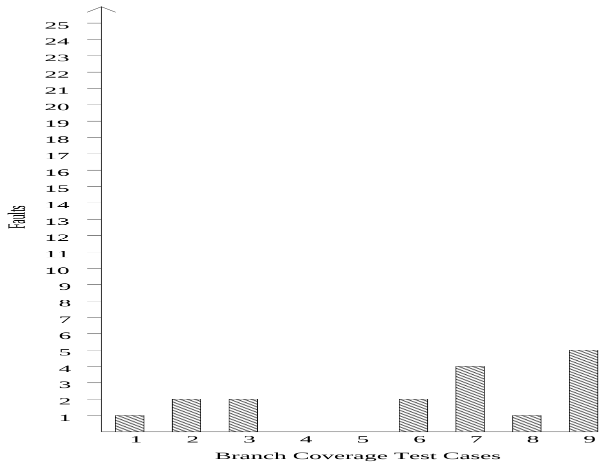


Figure 29: **Branch Coverage Test Cases Results**

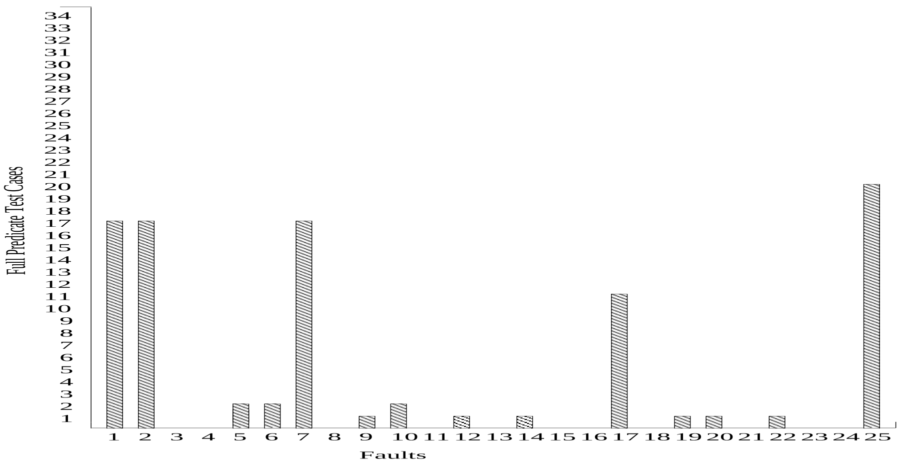


Figure 30: **Full Predicate Coverage Test Cases Results - faults found**

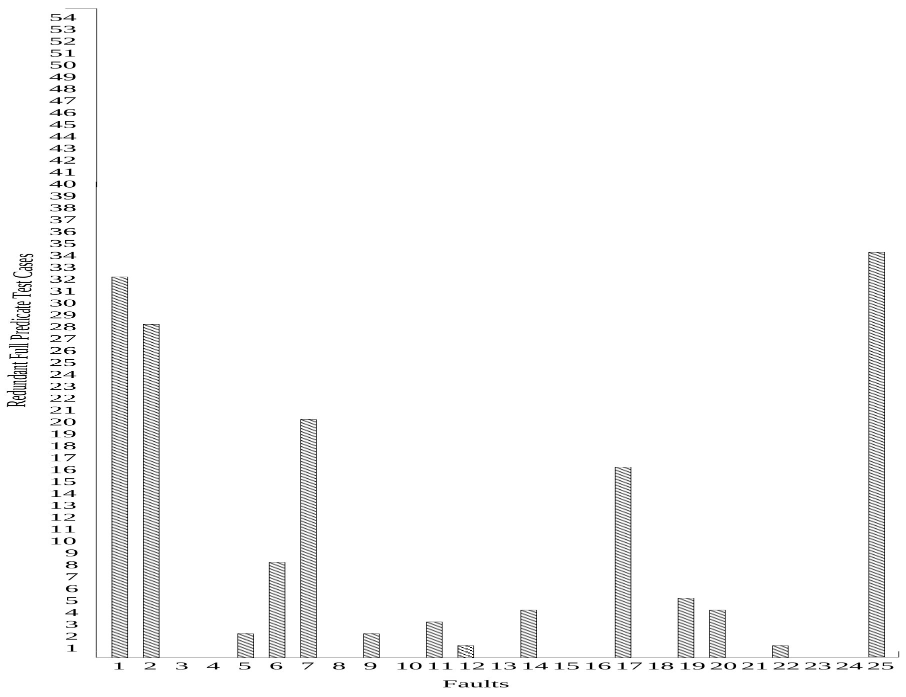


Figure 31: **Full Predicate Coverage Redundant Test Cases Results - faults found**

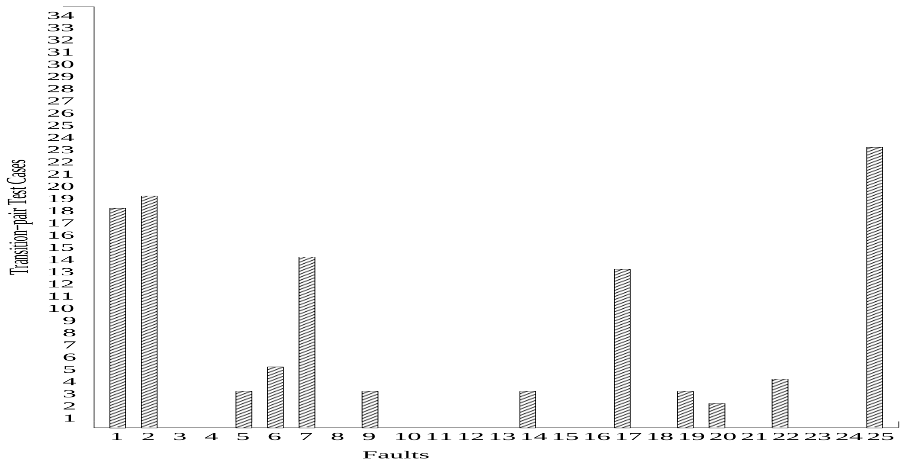


Figure 32: **Transition-pair Coverage Test Cases Results - faults found**

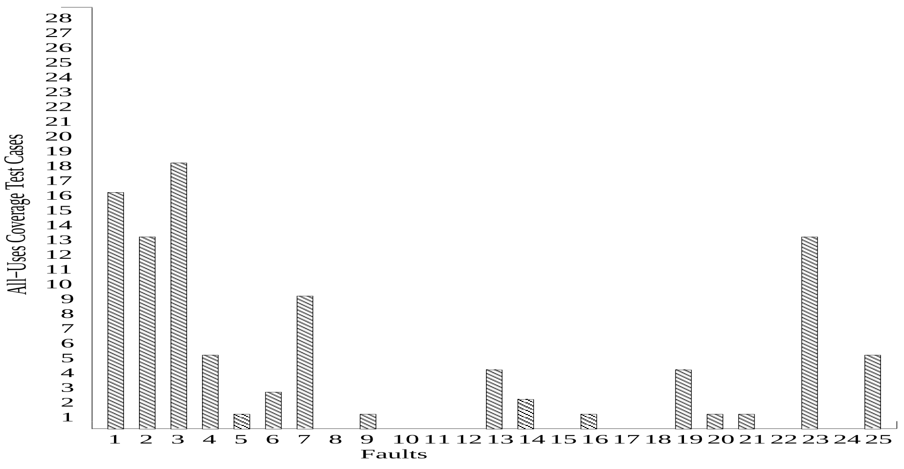


Figure 33: **All-Uses Coverage Test Cases Results - faults found**

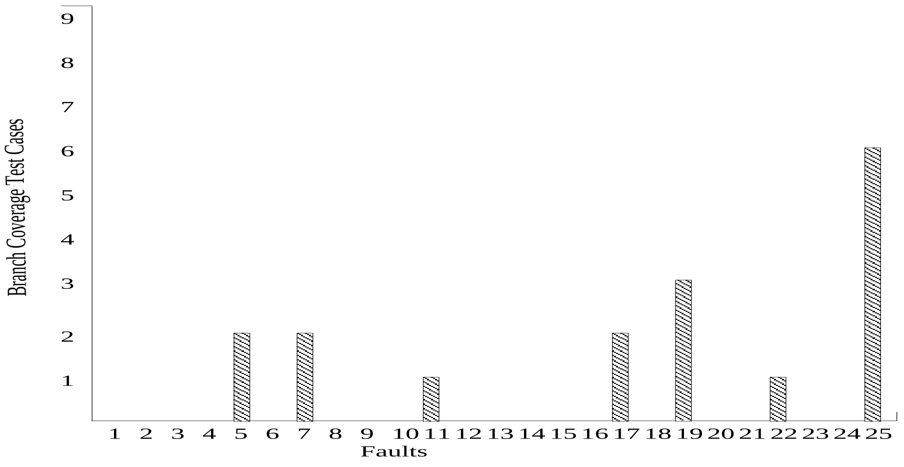


Figure 34: **Branch Coverage Test Cases Results - faults found**

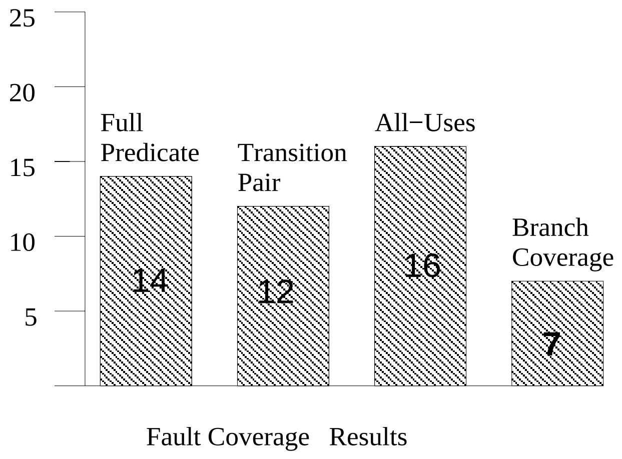


Figure 35: **Comparison of Results**

6 CONCLUSIONS AND RECOMMENDATIONS

In this section, we give the conclusions first and then present our recommendations. The recommendations are both on improving the test data generation techniques and on improving the software.

6.1 Conclusions

This thesis presents a model for automatically generating test data from state-based specifications. Specifically, SCR specifications generated by SCR* Toolset and UML specifications generated by Rose are used as a basis for generating *full predicate* and *transition-pair* coverage test cases. This thesis also presents algorithms for test data generation, and results from a proof-of-concept tool. The tool automatically generates test data for full predicate and transition-pair coverage criteria from state transition tables of SCR specifications and statecharts of UML specifications. Our results show that:

- Generating test data from state-based specifications can be fully automated.
- Full predicate coverage criterion test cases have a high capacity to catch faults at system test level. In the experiments, the fault coverage percentage is 56% for 25 faults of the cruise control program (see Figure 35 in Chapter 5). It is next to the fault coverage percentage of all-uses test cases, but full predicate test cases are less expensive to generate than all-uses test cases.
- Full predicate coverage test cases found more faults than transition-pair coverage test cases.

- The number of full predicate coverage test cases generated by the tool is less than the estimated number of test cases by the test case generation model.

6.2 Recommendations

In this subsection, we make a recommendation for improving the test case generation technique.

The state-based specification test case generation model described in this paper does not consider safety invariants of a system. This is problematic for the *full predicate* coverage criterion. For *full predicate* coverage criterion, we generate a test case that satisfies a transition predicate, and a series of test cases that do not satisfy the transition predicate (see section 2.3). The test cases that do not satisfy the transition predicate are generated by negating each clause in the transition predicate, one in a time. Changes to the values of variables that are invariants must cause a transition from current state to another state because of the violation of safety invariants. Therefore, if an invariant appears as an event variable in a transition, there is no problem. However, if an invariant appears in the condition part of a transition, changes to the invariant's value should cause an immediate state transition from the current state. There may be other variables that come after the variables of an invariant in a test case value, if these value assignments satisfy a transition predicate of the newly changed state, it causes another state transition. Predicting the output of this kind of test case will be hard, especially in automated environment. To avoid confusion, we suggest the following modification to the test case generation model: *If a state invariant appears in the condition part of a transition predicate, keep the value of the invariant unchanged during the generating test cases for invalid transitions process.*

Another recommendation is all paths of a prefix should be considered in the test case generation. As we discussed in Chapter 5, some faults can only be revealed through a specific path. Choosing only one path as a prefix cannot result in complete test.

6.3 Future Work

Although TCGen proved that we can automatically generate test cases from state-based specifications, but it is currently limited to SCR specifications that have only one mode class and UML specifications that have only one class with a statechart. Also, it can process only state transitions that have boolean type event and condition variables. This restricts TCGen's usability in industry. Processing more than one mode class in SCR or class in UML is not a problem, since it is simply a matter of repeating the process. It is more difficult to consider non-boolean type of variables in automation, and harder still to handle state transition types other than the *enabled transition* in UML.

APPENDIX A: SCR SPECIFICATION FILE OF CRUISE CONTROL SYSTEM

```
// This file contains an SCRTool system specification.
// It may have been written by the SCRTool and not changed since.

// This first definition describes the specification as a whole.
// The VERSION is the version of the SCRTool matching this file.
// The DESCRIPTION is the description field for this specification
// as it appears in the main SCRTool window.
SPECIFICATION; VERSION "1.6";

// This section contains all of the items in the
// Type Dictionary.

// This section contains all of the items in the
// Mode Class Dictionary.
MODECLASS "Cruise"; MODES "OFF, INACTIVE, CRUISE, OVERRIDE";
INITMODE "OFF";

// This section contains all of the items in the
// Constant Dictionary.

// This section contains all of the items in the
// Variable Dictionary.
MON "Activate"; TYPE "Boolean"; INITVAL "FALSE";
ACCURACY "N/A";
MON "Brake"; TYPE "Boolean"; INITVAL "FALSE"; ACCURACY
"N/A";
MON "Deactivate"; TYPE "Boolean"; INITVAL "FALSE";
ACCURACY "N/A";
MON "Ignited"; TYPE "Boolean"; INITVAL "TRUE";
ACCURACY "N/A";
MON "Resume"; TYPE "Boolean"; INITVAL "FALSE";
ACCURACY "N/A";
MON "Running"; TYPE "Boolean"; INITVAL "FALSE";
ACCURACY "N/A";
MON "Toofast"; TYPE "Boolean"; INITVAL "FALSE";
ACCURACY "N/A";

// This section contains all of the items in the
// Specification Assertion Dictionary.
// This section contains all of the items in the
// Environmental Assertion Dictionary.

// This section contains the event, mode transition, and condition functions.

MODETRANS "Cruise";
  FROM "OFF" EVENT "@T(Ignited)" TO "INACTIVE";
  FROM "INACTIVE" EVENT "@F(Ignited)" TO "OFF";
  FROM "INACTIVE"
EVENT "@T(Activate) WHEN (Ignited AND Running AND NOT Brake)"
TO "CRUISE";
  FROM "CRUISE" EVENT "@F(Ignited)" TO "OFF";
  FROM "CRUISE" EVENT "@F(Running) WHEN Ignited" TO "INACTIVE";
  FROM "CRUISE" EVENT "@T(Toofast) WHEN Ignited" TO "INACTIVE";
  FROM "CRUISE"
EVENT "@T(Brake) WHEN (Ignited AND Running AND NOT Toofast)"
TO "OVERRIDE";
  FROM "CRUISE"
EVENT
"@T(Deactivate) WHEN (Ignited AND Running AND NOT Toofast)"
TO "OVERRIDE";
  FROM "OVERRIDE" EVENT "@F(Ignited)" TO "OFF";
  FROM "OVERRIDE" EVENT "@F(Running) WHEN Ignited" TO "INACTIVE";
  FROM "OVERRIDE"
EVENT "@T(Activate) WHEN (Ignited AND Running AND NOT Brake)"
TO "CRUISE";
  FROM "OVERRIDE "
EVENT "@T(Resume) WHEN (Ignited AND Running AND NOT Brake)"
TO "CRUISE";
```

APPENDIX B: (UML SPECIFICATION) MDL FILE OF CRUISE CONTROL SYSTEM

```
(object Petal
    version     42
    _written    "Rose 4.5.8054a"
    charSet     0)

(object Design "Logical View"
    is_unit     TRUE
    is_loaded   TRUE
    defaults    (object defaults
rightMargin  0.250000
leftMargin  0.250000
topMargin   0.250000
bottomMargin  0.500000
pageOverlap  0.250000
clipIconLabels  TRUE
autoResize  FALSE
snapToGrid  TRUE
gridX       15
gridY       15
defaultFont  (object Font
    size        10
    face        "Arial"
    bold        FALSE
    italics     FALSE
    underline   FALSE
    strike      FALSE
    color       0
    default_color  TRUE)
showMessageNum  3
showClassOfObject  TRUE
notation    "Unified")
    root_usecase_package  (object Class_Category "Use Case View"

         ... (omitted)

    root_category  (object Class_Category "Logical View"
quid        "3693F7F200DF"
exportControl  "Public"
global      TRUE
subsystem   "Component View"
quidu       "3693F7F200E1"
logical_models  (list unit_reference_list
    (object Class "Cruise Control"
quid        "3693F8090267"
stereotype  "Controller"
class_attributes  (list class_attribute_list
    (object ClassAttribute "Ignited"
quid        "36F2C8330045"
type        "Boolean"
initv       "false")
    (object ClassAttribute "Running"
quid        "36F2C83C0188"
type        "Boolean"
initv       "false")
    (object ClassAttribute "Brake"
quid        "36F2C8450218"
type        "Boolean"
initv       "false")
    (object ClassAttribute "Toofast"
quid        "36F2C84D025F"
type        "Boolean"
initv       "false")
    (object ClassAttribute "Activate"
quid        "36F2C85202B6"
type        "Boolean"
initv       "false")
    (object ClassAttribute "Deactivate"
quid        "36F2C86201E7"
type        "Boolean"
initv       "false")
    (object ClassAttribute "Resume"
quid        "36F2C86A0257"
type        "Boolean"
initv       "false"))


statemachine  (object State_Machine
    quid        "3693F833031C"
    states      (list States
(object State "Off"
    quid        "3693F83A02D6"
    transitions  (list transition_list
(object State_Transition
    quid        "3693F88503E2"
    label       ""
    supplier    "Inactive"
    quidu       "3693F84101DC"
    Event       (object Event "when"
quid        "3693F88503E3"
parameters  "Ignited")
    sendEvent   (object sendEvent
quid        "3693F88503E5"))
(object State_Transition
    quid        "369A1A900135"
    supplier    "$UNNAMED$0"
    quidu       "369A1A83012C"
    sendEvent   (object sendEvent
quid        "369A1A900138")))
    type        "Normal")
(object State "Inactive"
    quid        "3693F84101DC"
    transitions  (list transition_list
(object State_Transition
    quid        "369A1AB5017E"
    label       ""
    supplier    "Off"
    quidu       "3693F83A02D6"
    Event       (object Event "when"
quid        "369A1AB5017F"
parameters  "not Ignited")
    sendEvent   (object sendEvent
quid        "369A1AB50181"))
(object State_Transition
    quid        "369A1B380136"
    label       ""
    supplier    "Cruise"
    quidu       "3693F874027F"
    Event       (object Event "when"
quid        "369A1B380137"
parameters  "Activate")
    condition   "Ignited AND Running AND NOT Brake"
    sendEvent   (object sendEvent
quid        "369A1B380139"))
(object State_Transition
    quid        "369A1DCB0397"
    supplier    "Cruise"
    quidu       "3693F874027F"
    sendEvent   (object sendEvent
quid        "369A1DCB039A"))
(object State_Transition
    quid        "369A1E510349"
    supplier    "Cruise"
    quidu       "3693F874027F"
    sendEvent   (object sendEvent
quid        "369A1E51034C")))
    type        "Normal")
(object State "Override"
    quid        "3693F8440059"
    transitions  (list transition_list
(object State_Transition
    quid        "369A1AF202F8"
    label       ""
    supplier    "Off"
    quidu       "3693F83A02D6"
    Event       (object Event "when"
quid        "369A1AF202F9"
parameters  "not Ignited")
    sendEvent   (object sendEvent
quid        "369A1AF202FB"))
(object State_Transition
    quid        "369A1EC10387"
    label       ""
    supplier    "Cruise"
    quidu       "3693F874027F"
    Event       (object Event "when"
quid        "369A1EC10388"
parameters  "Activate")
    condition   "Ignited AND Running AND NOT Brake"
    sendEvent   (object sendEvent
quid        "369A1EC1038A"))
(object State_Transition
    quid        "369A1EFF0295"
    label       ""
    supplier    "Inactive"
    quidu       "3693F84101DC"
    Event       (object Event "when"
quid        "369A1EFF0296"
parameters  "not Running")
    condition   "Ignited"
    sendEvent   (object sendEvent
quid        "369A1EFF0298"))
(object State_Transition
    quid        "369A470500D6"
    label       ""
    supplier    "Cruise"
    quidu       "3693F874027F"
    Event       (object Event "when"
quid        "369A470500D7"
parameters  "Resume")
    condition   "Ignited AND Running AND NOT Brake"
    sendEvent   (object sendEvent
quid        "369A470500D9")))
    type        "Normal")
(object State "Cruise"
    quid        "3693F874027F"
    transitions  (list transition_list
(object State_Transition
    quid        "369A1B06027F"
    label       ""
    supplier    "Off"
    quidu       "3693F83A02D6"
    Event       (object Event "when"
quid        "369A1B060280"
parameters  "not Ignited")
    sendEvent   (object sendEvent
quid        "369A1B060282"))
(object State_Transition
    quid        "369A1DB102D1"
    supplier    "Inactive"
    quidu       "3693F84101DC"
    sendEvent   (object sendEvent
quid        "369A1DB102D4"))
(object State_Transition
    quid        "369A1DBF00D3"
    supplier    "Inactive"
    quidu       "3693F84101DC"
    sendEvent   (object sendEvent
quid        "369A1DBF00D6"))
(object State_Transition
    quid        "369A1DC702D3"
    supplier    "Inactive"
    quidu       "3693F84101DC"
    sendEvent   (object sendEvent
quid        "369A1DC702D6"))
(object State_Transition
    quid        "369A1DD20206"
    label       ""
    supplier    "Inactive"
    quidu       "3693F84101DC"
    Event       (object Event "when"
quid        "369A1DD20207"
parameters  "not Running")
    condition   "Ignited"
    sendEvent   (object sendEvent
quid        "369A1DD20209"))
(object State_Transition
    quid        "369A1E4B0369"
    label       ""
    supplier    "Inactive"
    quidu       "3693F84101DC"
    Event       (object Event "when"
quid        "369A462701A6"
parameters  "Toofast")
    condition   "Running"
    sendEvent   (object sendEvent
quid        "369A1E4B036C"))
(object State_Transition
    quid        "369A1E6401F2"
    label       ""
    supplier    "Override"
    quidu       "3693F8440059"
    Event       (object Event "when"
quid        "369A1E6401F3"
parameters  "Brake")
    condition   "Ignited AND Running AND NOT Toofast"
    sendEvent   (object sendEvent
quid        "369A1E6401F5"))
(object State_Transition
    quid        "369A46BF023E"
    label       ""
    supplier    "Override"
    quidu       "3693F8440059"
    Event       (object Event "when"
quid        "369A46BF023F"
parameters  "Deactivate")
    condition   "Ignited AND Running AND NOT Toofast"
    sendEvent   (object sendEvent
quid        "369A46BF0241")))
    type        "Normal")
(object State "$UNNAMED$1"
    quid        "369A1A630299"
    transitions  (list transition_list
(object State_Transition
    quid        "369A1A72011E"
    supplier    "Off"
    quidu       "3693F83A02D6"
    sendEvent   (object sendEvent
quid        "369A1A720121")))
    type        "StartState")
(object State "$UNNAMED$0"
    quid        "369A1A83012C"
    type        "EndState")))
statediagram  (object State_Diagram ""

                     ... (physical description - omitted)
```

REFERENCES

REFERENCES

References

[AG93] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

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

[Bei90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[BH95] Jonathan P. Bowen and Micheal G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4), April 1995.

[BHO89] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.

[Boo94] Grady Booch. *Object Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998. ISBN 0-201-57168-4.

[CM94] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[FBWK92] S. Faulk, J. Brackett, P. Ward, and J. Kirby. The CoRE method for real-time requirements. *IEEE Software*, pages 22–33, September 1992.

[Hen80] K. Henninger. Specifiying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[HKL97] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of the 1997 Annual Conference on Computer Assurance (COMPASS 97)*, pages 35–47, Gaithersburg MD, June 1997. IEEE Computer Society Press.

[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.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 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.

[JK98] Jr. James Kirby. Scr* toolset: The user guide. Technical report, Naval Research Laboratory, April 1998.

[MH97] S. P. Miller and K. F. Hoech. Specifiying the mode logic of a flight guidance system in core. Release – 1 – 1, Collins Commercial Avionics, Rockwell Collins, Inc., Cedar Rapids, IA, 1997.

[MM92] D. Mandrioli and B. Meyer. Design by contract. In *Advances in Object-Oriented Software Engineering*, pages 1–49. Prentice-Hall, New York, 1992.

[Off98a] A. J. Offutt. Generating test data from requirements/specifications: Final report. Technical report ISSE-TR-98-01, Department of Information and Software Engineering, George Mason University, Fairfax VA, 1998.

[Off98b] 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.

[PC89] Andy Podgurski and Lori A. Clarke. The implications of program dependences for software testing, debugging, and maintenance. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 168–178, Key West, Florida, December 1989. ACM SIGSOFT'89.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[RAO92] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118. IEEE Computer Society, May 1992.

[RBP^{+}91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[SC-92] RTCA Committee SC-167. Software considerations in airborne systems and equipment certification, Seventh draft to Do-178A/ED-12A, July 1992.

[SC91] P. Stocks and D. Carrington. Deriving software test cases from formal specifications. Technical report 199, The University of Queensland, Department of Computer Science, May 1991.

[SC93] P. Stocks and D. Carrington. Test Templates: A Specification-Based Testing Framework. In *Proceedings of the 15th International Conference on Software Engineering*, pages 405–414, Baltimore, MD, May 1993.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.