Generating Test Data from SOFL Specifications [*]

A. Jefferson Offutt
ISSE Department, 4A4
George Mason University
Fairfax, VA 22030-4444 USA
email: ofut@isse.gmu.edu

Shaoying Liu
Faculty of Information Sciences
Hiroshima City University
Asaminami-ku, Hiroshima 731-31 Japan
email: shaoying@cs.hiroshima-cu.ac.jp

Abstract

Software testing can only be formalized and quantified when a solid basis for test generation can be defined. Tests are commonly generated from the source code, control flow graphs, design representations, and specifications/requirements. Formal specifications represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can be easily manipulated. This paper presents a new method for generating tests from formal specifications. This method is comprehensive in specification coverage, applies at several levels of abstraction, and can be highly automated. The paper applies the method to SOFL specifications, describes the technique, and demonstrates the application on a case study. A preliminary evaluation using a code-level coverage criterion (mutation testing), indicates that the method can result in very effective tests.

Keywords: Formal Methods, Specification-based Testing, Software Testing.

1 Introduction

There is an increasing need for effective testing of software for safety-critical applications, such as avionics, medical, and other control systems. This paper presents results from an ongoing project to improve the ability to test software for critical systems by developing techniques for generating test cases from formal specifications of the software. Formal specifications represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can easily be manipulated by automated means.

This paper presents a model for developing test inputs from model-based specifications, and a derivation process for obtaining the test cases. The test data generation model includes techniques for generating tests at several levels of detail. These techniques provide coverage criteria that are based on the specifications, and the test generation process details steps for transforming specifications to tests.

[*]The work is supported in part by the Ministry of Education of Japan under Joint Research Grant-in-Aid for International Scientific Research FM-ISEE (08044167).

Specification-based test data generation has several advantages over code-based generation. Requirements/specifications can be used as a basis for output checking, significantly reducing one of the major costs of testing. The process of generating tests from the specifications will often help the test engineer discover problems with the specifications themselves; if this step is done early, the problems can be eliminated early, saving time and resources. Generating tests during development also allows testing activities to be shifted to an earlier part of the development process, allowing for more effective planning and utilization of resources. Another advantage is that the test data is independent of any particular implementation of the specifications.

1.1 Using Specifications in Testing

Software functional specifications have been incorporated into testing in several ways. They have been used as a basis for test case generation, to check the output of software on test inputs, and as a basis for formalizing *test specifications* (as opposed to functional specifications). This paper is primarily concerned with the first use, that of generating test cases from specifications. An immediate goal is to develop *mechanical* procedures to derive test cases from formal specifications; long term goals include automated tool support to transform formal functional specifications into effective test cases.

Figure 1 provides an abstract view of part of the test process. A program P, along with a set of test cases T, is submitted to a computer C, which runs T on P to produce some results R. A primary concern for testers is how to produce T; the set of test cases should be effective at finding faults in the program, adequate at providing some information about the quality of the program, and preferably satisfy some requirements or criterion for testing that is repeatable, automatable, and measurable.

A common source for tests is the program code. In *code-based test generation*, a testing criterion is imposed on the software to produce test requirements. For example, if the criterion of branch testing is used, then the tests are required to cover each branch in the program. Figure 2 provides an abstract view of part of a typical test process that might be used for code-based test generation. The specification S (which can be formal or informal) is used as a basis for writing the program P, which is used to generate the tests T, according to some coverage criterion such as branch or data flow. Execution of T on P creates the *actual output*, which must be compared with the *expected output*. The expected output is produced from the test case with some knowledge of the specifications. Thus, code-based generation uses the specifications to generate the code and check the output of the tests.

This is in contrast to *specification-based testing*, an abstract view of which is shown in Figure

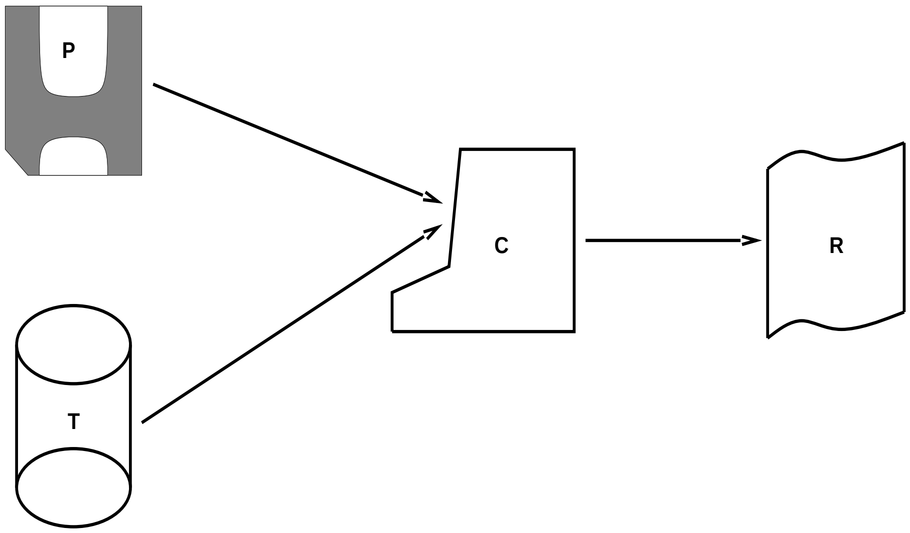

Figure 1: Specifications and Programs in Testing

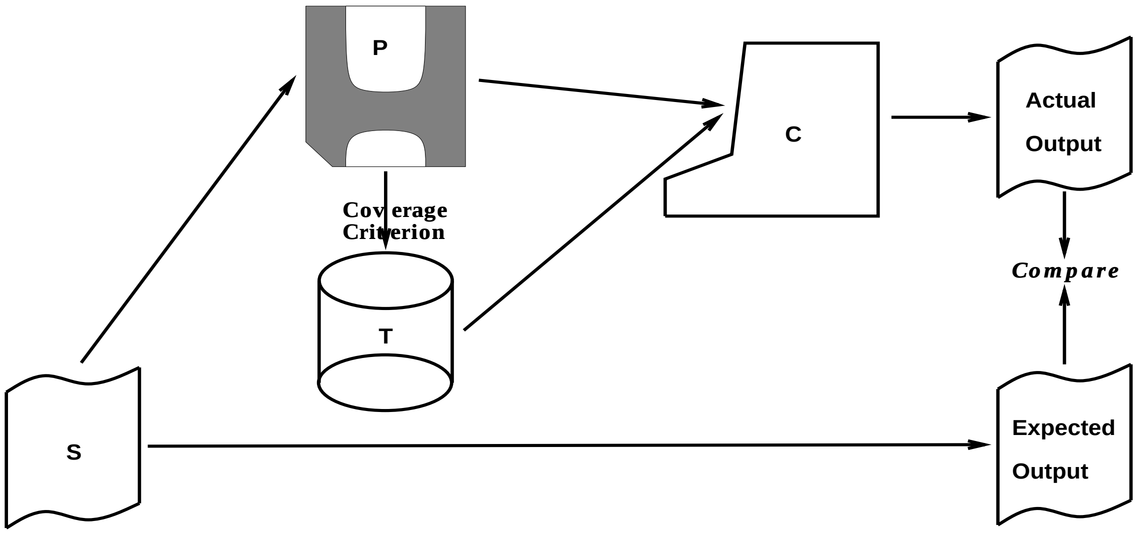

Figure 2: Code-based Test Generation

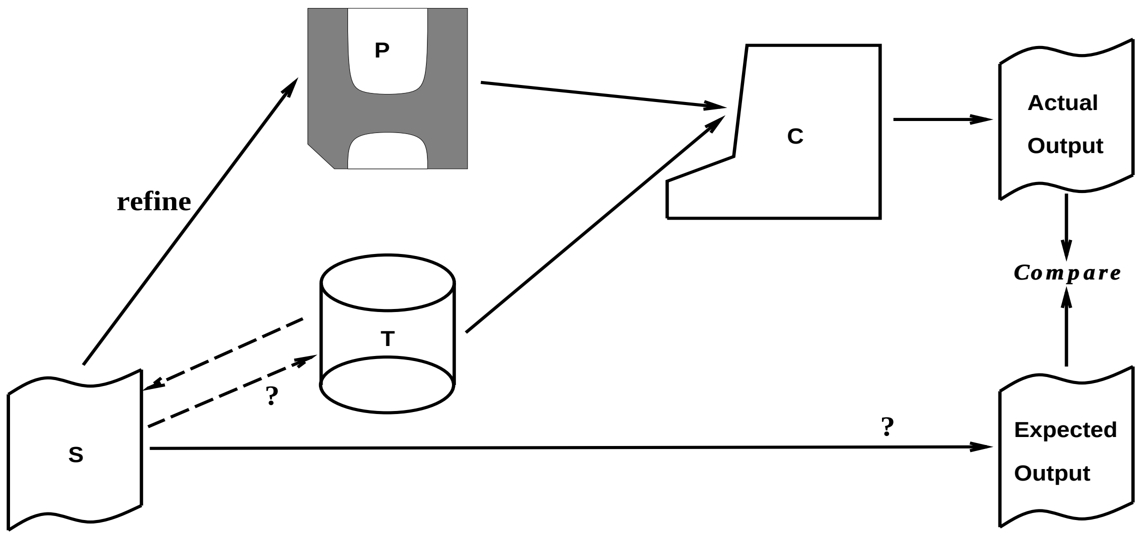

Figure 3: Specification-based Test Generation

3. Here the specifications are used to produce test cases, as well as to produce the program. In this scenario, the specifications are more likely to be formalized, so the arc from S to P is labeled as a “refinement”, a process that is often used to create code from formal specifications. One significance of producing tests from specifications is that the tests can be created earlier in the development process, and be ready for execution **before** the program is finished. Additionally, when the tests are generated, the test engineer will often find inconsistencies and ambiguities in the specifications, allowing the specifications to be improved before the program is written. The arcs from S to T and from S to $ExpectedOutput$ are labeled with a “?”, because these are currently areas of active research. This project is looking at ways to generate tests from specifications; others, such as Li et al. have been developing techniques for creating expected output from specifications [LS96, HKL$^+$95, LYZ94].

Both specification-based and code-based test generation have strengths, and both are used in practice. Both methods have been criticized (sometimes unfairly), and both have been supported (sometimes too strongly). The authors have carried out research involving both approaches, and would like to present the strengths and weaknesses of both approaches in a scientific, unbiased manner. While some of these are accidental differences of the current state of the research and available technology (using the philosophical distinction drawn by Brooks [Bro87]), some are also essential. This paper takes the position that specification-based test data generation is complementary to code-based test data generation, and that both are necessary.

Because specification-based testing only considers an external view of the software, it can be said to test the products, but not the design decisions, and it may not test all of the program. By deriving tests from the specifications, engineers are often able to find problems in the specifications. While this effect has not been formalized or quantified, experience has provided strong anecdotal evidence. An example is a project, called Mistix, that has been used in several classes and research projects [AO94, OI95]. It is a simplified file system program, used to illustrate trees, lists, type parameterization, and inheritance. The first author had supplied informal specifications to several classes before using it as an assignment in a graduate testing class. The students applied a modification of the category-partition method [OB88, BHO89] to the specifications to produce test cases. During the exercise, they identified many inconsistencies and ambiguities in the specifications, and found several points of incompleteness. These problems allowed faults to be found in existing implementations that had previously gone undiscovered.

Specification-based testing is also currently immature, which means there is a scarcity of formalizable criteria and automated tool support. It is this problem that this research is attempting to address. Specification-based testing has the potential to benefit from formal specifications, by using the formal specifications as input to an formalizable, automatable test generation process. Another advantage of specification-based testing is that it can support the automation of testing result analysis, by using specifications as test oracles.

A major disadvantage of code-based generation is that it tests what was built, rather than what was intended. Code-based tests also may not cover the entire input domain. On the other hand, code-based generation technology is very mature, and there are many formalized criteria for testing, and many tools available.

There are several things that are not known about generating tests from code and from specifications. If specification-based testing is used, engineers do not know how well those tests cover the program code; likewise, it is not known how well code-based tests might cover the input domain.

The two approaches are sometimes used in combination. The most common way is to generate tests based on the specifications, and then use *code-based coverage analysis* to measure the quality of the tests. For example, the tests might be measured by how many branches in the software are covered. There has been no published data concerning how effective this combination is. It is widely agreed, however, that it is difficult to construct tests that are inputs to the top level system that cover detailed code-level requirements (such as branches). This is why code-based test generation is typically thought of as useful for *unit testing*, where individual functions or modules are tested, and specification-based test generation is typically thought of as useful for *system testing*, where an entire working system is tested.

As said, this paper addresses the problem of a lack of formalizable, measurable criteria for generating test cases from specifications. Specifically, a model for generating tests from *SOFL specifications* is presented. SOFL is a specification language and methodology that is intended to be *usable* and *graphical*, and to *combine* the object-oriented design methodology with the structured design methodology. This paper first reviews the small but growing body of work on using formal specifications as a basis for producing test cases, then overviews the SOFL language and methodology. Then a model for generating tests from SOFL specifications is described, and results from a case study of testing a small program unit are presented.

2 Approaches to Specification-based Testing

The current research literature reports on specific tools for specific formal specification languages [BGM91, BCFG86, GMH81, Jal92, OSW86, TVK90, WGS94], manual methods for deriving tests from specifications [AA92, AO94, Ber91, DF93, Hay86, SCS97], case studies on using specifications to check the output of the software [DF91, Kem85, Lay92, SC93a], and formalizations of test specifications [SC96, SC93b, BHO89, Cho86]. This paper uses the term *specification-based testing* in the narrow sense of using specifications as a basis for deciding what tests to run on software. Some of these techniques are reviewed, dividing them into approaches that use model-based, state-based, and property-based specifications.

2.1 Model-based Approaches

Model-based specification languages, such as Z [Spi89] and VDM [Jon86], support formal specifications of the software based on set-theoretic models of real-world objects. Dick and Faivre [DF93] suggested using specifications to produce predicates, and then using predicate satisfaction techniques to generate test data. Given a set of predicates that reflect preconditions, invariants, and postconditions, test cases are generated to satisfy individual clauses. Their work was for VDM specifications, and primarily focused on state-based specifications, using finite state automata representations. Dick and Faivre discussed straightforward translation of the specifications into disjunctive normal form predicates, and presented solutions to the problem of predicate satisfaction by using prolog theorem proving techniques.

Stocks and Carrington [SC93b, SC93a] and Amla, Ammann, and Offutt [AA92, AO94] proposed using a form of domain partitioning to generate test cases. Given a description of an input domain, the idea is to use specifications to partition the input domain into subsets. The Amla, Ammann, and Offutt approach is based on a modification to the category-partition method for test generation [BHO89, OSW86]. Hierons [Hie97] presents algorithms that rewrite Z specifications into a form that can be used to partition the input domain. From this, states of a finite state automaton are derived, which are then used to control the test process.

Hayes [Hay86] has suggested a dynamic scheme that uses run-time verification of the program. The idea is to add code to the program to check predicates from the specifications, such as type invariants, preconditions, and input-output pairs.

Singh et al. [SCS97] used a method called "classification-tree" to generate test cases from Z specifications. The classification-tree is used to organize the input predicates, which are put into a disjunctive normal form for generation of values.

2.2 State-based Approaches

State-based specifications are a variant of model-based specifications with greater emphasis on states and 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. For example, SCR [Hen80, AG93] calls these WHEN conditions and triggering events. The values of the triggering events before the transitions are sometimes called before-values, and the values after the transition are sometimes called after-values. The state immediately preceding the transition is called the pre-state, and the post-state is the state after the transition.

Blackburn and Busser [BB96] used state-based functional specifications of the software, expressed in the language `T-Vec`, to derive disjunctive normal form constraints, similarly to Dick and Faivre's method. These constraints are then solved to generate test cases. There is a strong similarity between Blackburn and Busser's algorithms and the algorithms used by Offutt's test data generator [DGK$^+$88, DO91]; the key difference being that Blackburn and Busser's is specification-based, whereas Offutt's constraints are code-based.

Weyuker, Goradia, and Singh [WGS94] present a method to generate test data from boolean logic specifications of software. They applied their techniques to the FAA's Traffic Collision and Avoidance System (TCAS), and used mutation-style faults to measure the quality of the test cases.

2.3 Property-based Approaches

Property-based approaches attempt to specify software without reference to an explicit model, but instead with axioms that describe the relationships between functions. Algebraic specifications are the typical property-based approach, and Gannon, McMullin and Hamlet [GMH81] used a script derivation approach for deriving test cases from algebraic specifications. They treated the axioms as a language description and generated strings on that language to serve as test cases. Doong and

Frankl [DF91] used a similar approach to test object-oriented software.

Bernot [Ber91] proposed a similar scheme, with more formalization of the process and the test cases. Bougé et al. [BCFG86] suggested a logic programming approach to generating test cases from algebraic specifications. Tsai, Volovik, and Keefe [TVK90] used a similar approach, but started with relational algebra queries.

While property-based approaches are certainly worth pursuing, this paper focuses on the model-based approach to specification. The main reason is that model-based approaches enjoy a wider currency in industrial application.

2.4 Summary

Most of the current specification-based testing techniques use manual methods that cannot be easily generalized or automated. The goals of this research include generalizing currently known techniques, defining measurable criteria, and developing automated tools. This research is part of an ongoing project into the practical applications of formal methods.

3 Generating Tests from SOFL Specifications

An initial result from this research project is the specification language SOFL. SOFL uses a graphical representation to integrate structured methods, an object-oriented methodology and formal methods. This section first provides a brief introduction to the SOFL languages, then presents a technique for generating test data from SOFL specifications.

3.1 Brief Introduction to SOFL

This overview focuses only on the constructs of SOFL that are used for the test generation; a full description can be found elsewhere [LS95, LSOHS98]. This section starts by describing the overall structure of a SOFL system and then gives a rather precise definition of the constructs that are used for test data generation.

3.1.1 SOFL System Structure

Construction of a software system using SOFL starts with the construction of a hierarchical condition data flow diagram, as illustrated in Figure 4. A *condition data flow diagram*, or CDFD, is a directed graph consisting of data flows, data stores, and condition processes. A *data flow* is labeled with a variable, which represents a data transition between condition processes. A *data store* is a variable that represents data at rest. All variables and data objects are typed. A *condition process* is like a process in DeMarco data flow diagrams [DeM78]; it is a transition between two states that uses preconditions to specify when the transition can be taken, and postconditions to state what is true if the transition is taken. For each CDFD, a specification module, called *S-module*, is provided to define types and variables and to specify the functionality of all the condition processes in the CDFD by giving their pre and postconditions. The specification module and the CDFD are complementary in the sense that the CDFD describes the relationships between condition processes in terms of data flows and the module describes the precise functionality of each condition processes (without explicit indication of their relationships). They are also complementary in the sense that the CDFD provides a graphical view of the system at the current level while the module supplies the details of the system in a textual form.

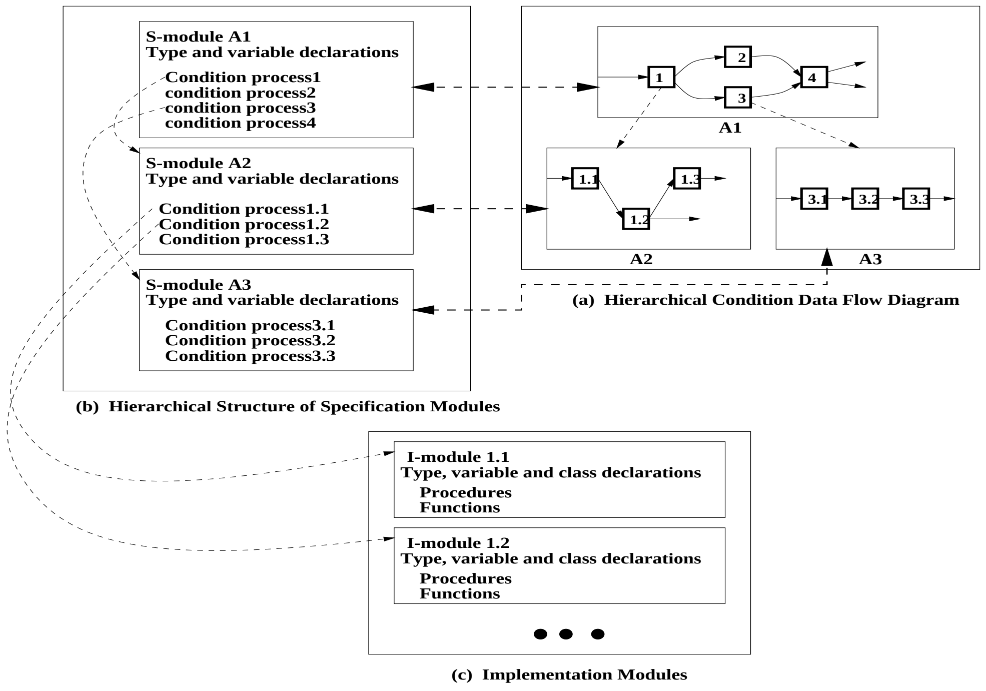

Figure 4: The Structure of a SOFL System

An important refinement of a high level condition process is its decomposition into a lower level CDFD under its functional constraints, as expressed by the pre and postconditions. The decomposed CDFD describes the details of how the functionality of the high level condition process is realized by lower level condition processes, and the details about all the condition processes in the lower level CDFD are described in their corresponding specification modules. The relationship between a high level condition process and its decomposed CDFD is reflected by building a link in the S-module. This decomposition process proceeds until all the lowest level condition processes are simple enough for implementation. Then each lowest level condition process is implemented (or prototyped) in an executable implementation module, called *I-module*. The structure of an I-module is similar to a C++ module, including type, variable and class declarations (optional), and procedure and/or function definitions. However, since the same abstract data types used in the specification modules (e.g. sequences, sets, and maps) are adopted, the implementation is done on a relatively abstract level compared with C++ or Pascal-like programs. This helps prototyping and facilitates the verification of the implementation against its specification. Since every high level condition process is realized in terms of a lower level CDFD and every lowest level condition process is implemented using an I-module, the whole system is executable.

3.1.2 Relevant Constructs of SOFL Specifications

As our interest is that of using SOFL specifications for testing, we focus on the specification part. Specifically, three kinds of constructs are tested: invariants, condition processes, and condition data flow diagrams (CDFDs).

In SOFL, an invariant is given in the declaration part of an S-module and is used to describe a property of some type or variable that remains unchanged throughout the whole system. Its format is:

forall [x1 **in** D1, x2 **in** D2, ..., xn **in** Dn | P(x1, x2, ..., xn)]

where each xi is a variable and each Di is a type or a variable of type set. A condition process is specified in an S-module with the following form:

`CP-name (x1: T1, x2: T2, ..., xn: Tn) y1: TT1, y2: TT2, ..., ym: TTm`

pre `P1`

post `P2`

decomposition `M-name`

In this condition process specification, CP-name is the name of the condition process, the xi's represent the input data flows to the condition process, and the yj's represent the output data flows. The Ti's are the types of the input data flows, and the TTi's are types of the output data flows. P1 is the precondition of the condition process that constrains the inputs, and P2 is the postcondition that is required to be satisfied by the outputs after an execution of the condition process with the precondition. M-name indicates the name of the S-module whose corresponding CDFD is the decomposition of the condition process.

As described before, a CDFD is a directed graph describing data transitions between condition processes. Its graphical constructs are given in Figure 5. Figure 5(a) describes a sequential structure. It means that after A1, A2, ..., An fire and x1, x2, ..., xn are produced, then B is enabled to fire, and y is produced as the result. Figure 5(b) indicates that when the condition C(x) is satisfied by x, then data from x will flow along the upper arc to B1, otherwise, it will flow along the lower arc to B2. Figure 5(c) shows a multiple selection structure. When x satisfies Ci(x) (i=1...n), then data from x will flow along the corresponding arc to the condition process Bi. Otherwise, it will flow along the lowest level arc to Bn+1. Figure 5(d) presents a loop structure. When x is available, B will be fired and x1 will be produced as the result. Then, because x1 is available, B is fired again and x1 is produced again. This repetition continues until y is produced.

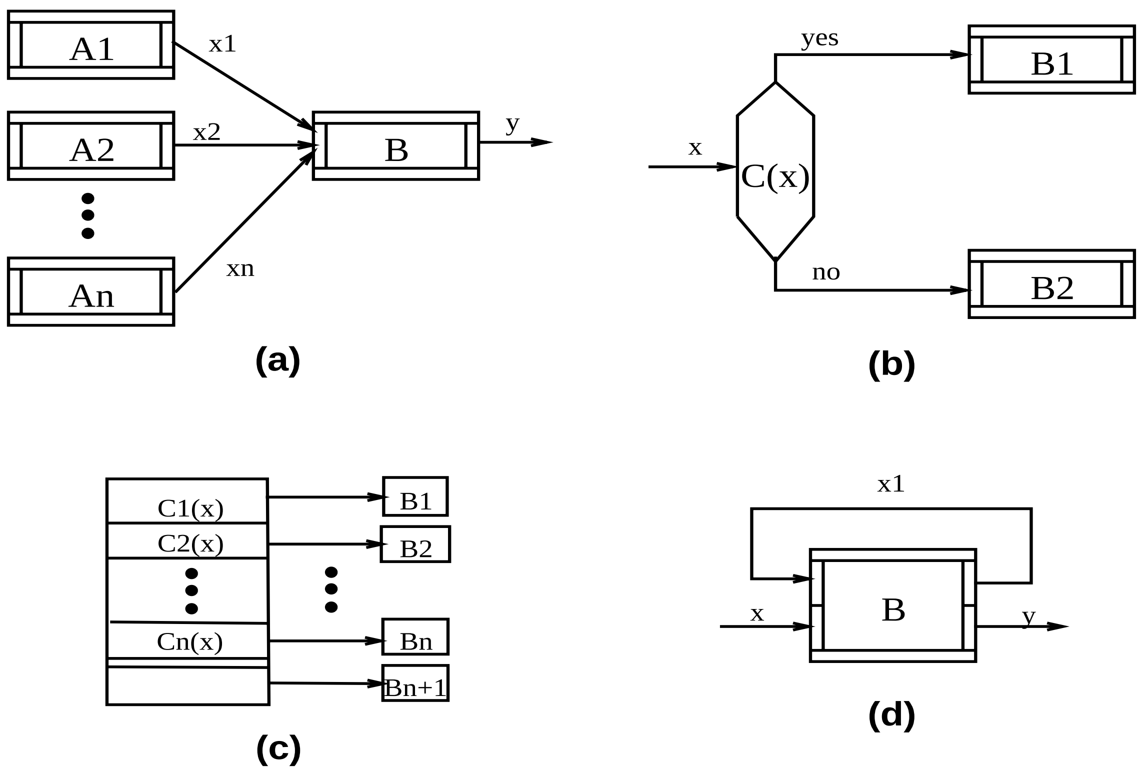

Figure 5: The Graphical Constructs of CDFDs

One advantage of SOFL for testing is that different representations are used for the specifications at different abstraction levels of the software. The next section discusses how to use the SOFL specifications to generate tests at different levels. S-modules are used to generate system level tests, and I-modules to generate tests for module components.

3.2 SOFL-based Testing

Section 2 discusses the notion of *predicate satisfaction.* 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 relate the pre to postconditions with the tests. Tests should also be created to find faults, as well as to cover the input domain. For SOFL specifications, it is important to test both S-modules and I-modules; different approaches can be used to generate tests at these different levels. A remaining problem is that of finding missing conditions.

SOFL specifications are at two principle levels, the S-module level and the I-module level. These descriptions take on different forms, so the goals of testing will be different. This paper adapts a system level testing technique for S-modules, a data flow-based testing approach for integration testing among S-modules, and extends the predicate-based testing approaches for testing I-modules.

3.2.1 Generating Tests from S-Modules

S-Modules specify condition data flow diagrams (CDFDs) in textual form. These can be viewed as a definition of **types**, **objects**, and **functions** that operate on objects of those types. Category-partitioning is directly applied as in Ammann and Offutt [AO94]. Test specifications are constructed as an intermediate between functional specifications and actual tests. A minimal coverage criterion for category-partition testing is used, and a mechanical process to produce tests that satisfy the criterion is applied.

The category-partition method [BHO89, OB88] is a specification-based testing strategy that uses an informal functional specification to produce formal test specifications. The category-partition method offers the test engineer a general procedure for creating test specifications. The test engineer's key job is to develop *categories*, which are defined to be the major characteristics of the input domain of the function under test, and to partition each category into equivalence classes of inputs called *choices*. By definition, choices in each category must be disjoint, and together the choices in each category must cover the input domain.

The steps in the category-partition method that lead to a test specification are as follows:

1. Analyze the specification to identify the individual **functional units** that can be tested separately.

2. Identify the input domain, that is, the **parameters** and **environment variables** that affect the behavior of a functional unit.

3. Identify the **categories**, which are the significant characteristics of parameter and environment variables.

4. Partition each category into **choices**.

5. Specify **combinations** of choices to be tested, instantiate **test cases** by specifying actual data values for each choice, and determine the corresponding **results** and the changes to the environment.

Each specified combination of choices results in a *test frame*. The category-partition method relies on the test engineer to determine constraints among choices to exclude certain test frames. There are two reasons to exclude a test frame from consideration. First, the test engineer may decide that the cost of building a test script for a test frame exceeds the likely benefits of executing that test. Second, a test frame may be infeasible, in that the intersection of the specified choices is the empty set.

The developers of the category-partition method have defined a test specification language called TSL [BHO89]. A test case in TSL is an operation and values for its parameters and relevant environment variables. A test script in TSL consists of the operations necessary to create the environmental conditions (called the SETUP portion), the test case operation, 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 test script. The test engineer may optionally give specific representative *values* for any given choice to aid the test generation tool in deriving specific test cases. The category-partition method supplies little explicit guidance as to which combinations of choices are desirable – the task is left mostly to the test engineer's judgment.

Ammann and Offutt had several differences in their use of the category-partition method. First, the derivation is based on formal specifications of the software, since, as has been demonstrated in a variety of papers [AA92, Lay92, SC93b, SC93c], the formality of the functional specification helps to simplify and organize the production of a test specification. Second, the TSL syntax is not followed. Specifically, as has been done by others [SC93b, SC93c], a formal specification notation is used to describe aspects of the tests themselves as well as to describe functional behavior.

3.2.2 Integration Tests for S-Modules

Although S-Modules specify condition data flow diagrams (CDFDs) in textual form, they only define the condition processes of CDFDs separately, giving no relationships between the condition processes. It is the CDFDs (graphical notation) that describe their relationships in terms of data flows. In other words, CDFDs describe how the condition processes defined in S-modules are integrated to form an entire system.

Since condition processes in a CDFD are related in terms of data flows, it is natural and easy to conduct data flow based testing for integration testing. The essential idea is to generate test data to cover all the possible data flows. To this end, test data (input data flows) are generated to test every possible graphical constructs given in Figure 5.

Testing sequential structures. Figure 5(a) shows a sequential structure in a CDFD. This is a sequential structure in the sense that condition process B cannot fire until after condition processes A1, A2, ..., An fire. Therefore, generating test data for the variables x1, x2, ..., xn is necessary to generate values for y.

Criterion CDFD-1:

(1) Every input data flow of a condition process must be used in a test.

(2) Every output data flow of a condition process must be generated in a test.

Testing selection structures. There are two selection constructs in CDFDs, as shown in Figure 5(b) and (c). To test such selection structures, the following criterion is defined.

Criterion CDFD-2:

Generate test data for x so that each branch of the selection can be used once.

Testing iteration structures. An iteration structure is possible in a CDFD, as shown in Figure 5(d). To test such a structure, it must be ensured that when condition process B is fired that consumes the input data flow x, the output data flows x1 and y are generated once, respectively. This will also test whether a loop like the one in Figure 5(d) terminates. This is actually consistent with criterion CDFD-2, so a separate criterion is not defined.

3.2.3 Generating Tests for I-Modules

As an I-module provides an implementation of a lowest level condition process specification, a predicate-based approach is used to test I-modules, but the previous approaches are extended in several ways. Primarily, the predicates are measured at three different levels; the disjunctive, conjunctive, and the relational level. For convenience, it is assumed the predicates are in disjunctive normal form (DNF). The primary intent is that each clause in each predicate is tested independently.

At the disjunctive level, the predicates are in the form $(A \vee B \vee C \vee ...)$. These are tested by holding all disjuncts but one `False`, and varying each one to be `True` in turn. That is, tests are generated by finding values that satisfy the following partial truth table:

A	$\vee$	B	$\vee$	C	$\vee$	...
T		F		F		...
F		**T**		F		...
F		F		**T**		...
		⋮				

These tests cases sample from the **valid** parts of the input space. In addition, the test engineer may include combinations of two or more `True` disjuncts that are **semantically** meaningful. Because the program will be tested from the specification, it is only necessary to generate test data for the input parameters and state variables in the pre and postconditions.

At the conjunctive level, the predicates are in the form: $(A \wedge B \wedge C \wedge ...)$ and each clause is tested in turn. First the entire expression is forced to be `True` by finding values that cause each clause to be `True`. Then **invalid** parts of the input space are sampled by holding all conjuncts but one `True`, and varying each one to be `False` in turn. That is, tests are generated by finding values that satisfy the following partial truth table:

A	$\wedge$	B	$\wedge$	C	$\wedge$	...
T		T		T		...
F		T		T		...
T		**F**		T		...
T		T		**F**		...
		⋮				

Additionally, the test engineer may include combinations of two or more `False` conjuncts that are **semantically** meaningful.

At the relational level, a limited form of domain analysis [WC80] is applied to the expressions. This serves to test the boundaries of the relations. Assume that there are unary operators `SUCC` and `PRED` defined for each linearly ordered type, which returns the successor and predecessor respectively of a given value. These operators are provided in Ada for the built-in types. For example, `SUCC (X)` is equivalent to `X+1` for integer types. Then for each relational expression `X rop Y`, where `X` and `Y` are arbitrary expressions and `rop` is some relational operator, replace the relational expression by:

- `X = Y`

- `X = SUCC (Y)`
- `X = PRED (Y)`

and generate values to satisfy the predicates.

3.3 I-Module Level Example

`Triang` is a small program that has been widely used in the testing literature [ABD+79, CR83, DLS78, RHC76]. `Triang` inputs three integers that represent the relative lengths of the sides of a triangle and classifies the triangle as equilateral, isosceles, scalene or illegal. SOFL specifications for `Triang` are given in Figure 6.

Classifications = {EQUILATERAL, ISOSCELES, SCALENE, INVALID}
Triang (S1, S2, S3 : **integer**) : Classifications

pre: S1>0 **and** S2>0 **and** S3>0;

post: S1=S2 **and** S2=S3 **and** S1=S3 **implies** Triang=EQUILATERAL **and**
S1+S2 $\leq$ S3 **or** S1+S3 $\leq$ S2 **or** S2+S3 $\leq$ S1 **implies** Triang=INVALID **and**
S1=S2 **or** S2=S3 **or** S1=S3 **implies** Triang=ISOSCELES **and**
S1$\neq$S2 **and** S2$\neq$S3 **and** S1$\neq$S3 **and**
S1+S2 > S3 **and** S1+S3 > S2 **and** S2+S3 > S1 **implies** Triang = SCALENE

Figure 6: SOFL Specifications for Triang

If the preconditions are considered at the conjunctive level, all disjuncts but one must be held `True`, and each one is varied to be `False` in turn. This is done by appending values to the sides of the following partial truth table:

S1>0	$\wedge$	S2>0	$\wedge$	S3>0	S1	S2	S3
T		T		T	1	1	1
F		T		T	0	1	1
T		**F**		T	1	0	1
T		T		**F**	1	1	0

To generate values for the relational level, the relations are satisfied separately, resulting in the following table. This is a conjunctive clause, therefore the predicates that are not being considered must be held constant at `True`, so as not to affect the value of the full predicate. For example, for the first three rows of this table, where S1 is being considered, the values for S2 and S3 are held constant at 1.

S1>0	$\wedge$	S2>0	$\wedge$	S3>0	S1	S2	S3
S1=0					0	1	1
S1=0+1					1	1	1
S1=0−1					−1	1	1
		S2=0			1	0	1
		S2=0+1			1	1	1
		S2=0−1			1	−1	1
				S3=0	1	1	0
				S3=0+1	1	1	1
				S3=0−1	1	1	−1

The postconditions are more complicated. There are four separate conditions, which represent four possible cases. Only one condition can be true at a time, so for the purposes of testing, these are disjunctive clauses. For convenience, these are labeled as follows:

A: `S1=S2 and S2=S3 and S1=S3`

B: `S1+S2` $\leq$ `S3 or S1+S3` $\leq$ `S2 or S2+S3` $\leq$ `S1`

C: `S1=S2 or S2=S3 or S1=S3`

D: `S1`$\neq$`S2 and S2`$\neq$`S3 and S1`$\neq$`S3 and S1+S2` $>$ `S3 and S1+S3` $>$ `S2 and S2+S3` $>$ `S1`

The disjunctive level tests are:

A	$\vee$	B	$\vee$	C	$\vee$	D	S1	S2	S3
T		F		F		F	1	1	1
F		**T**		F		F	1	1	2
F		F		**T**		F	1	1	0
F		F		F		**T**	2	3	4

At the conjunctive level, case A contains three clauses, so tests should be generated from it by holding all conjuncts but one `True`, and varying each one to be `False` in turn. Unfortunately, none of these conditions can be satisfied; if S1=S2 and S2=S3, then by transitivity, S1=S3, so it is impossible for only two of the three clauses to be `True`. This is illustrated in the following table. From inspection, it is reasonable to relax the requirement, and allow two of the three clauses to be `False`. This kind of reasoning is difficult to introduce algorithmically, but the following scheme will provide an adequate approximation. If holding one clause `False` leads to an infeasible constraint system, the engineer can attempt to cause a second clause to be `False`. Of course, it is difficult to recognize whether a constraint system is infeasible. This problem is generally undecidable, but heuristics have been developed that can find infeasibility in a large number of cases [OP97]. A reasonable extension that handles this problem is to add one more test case at the conjunctive level: require **all** conjuncts in the expression to be `False`. The bottom line in the table below reflects this extenstion.

S1=S2	∧ S2=S3	∧ S1=S3	S1	S2	S3
T	T	T	1	1	1
F	T	T	*infeasible*		
T	**F**	T	*infeasible*		
T	T	**F**	*infeasible*		
F	**F**	**F**	2	3	4

Case B again contains three disjunctive clauses. The resulting test cases are shown in the following table.

$S1+S2 \leq S3$	$\vee$ $S1+S3 \leq S2$	$\vee$ $S2+S3 \leq S1$	S1	S2	S3
T	F	F	1	1	2
F	**T**	F	1	2	1
F	F	**T**	2	1	1

Case C contains three disjunctive clauses, so tests should be generated from it by holding all disjuncts but one `False`, and varying each one to be `True` in turn. These test cases are shown in the following table.

S1=S2	∨ S2=S3	∨ S1=S3	S1	S2	S3
T	F	F	1	1	0
F	**T**	F	1	0	1
F	F	**T**	0	1	1

Case D contains six conjunctive clauses, test cases for which are shown in the following table.

$S1 \neq S2$	$\wedge$ $S2 \neq S3$	$\wedge$ $S1 \neq S3$	$\wedge$ $S1+S2 > S3$	$\wedge$ $S1+S3 > S2$	$\wedge$ $S2+S3 > S1$	S1	S2	S3
T	T	T	T	T	T	3	4	5
T	T	T	T	T	**F**	8	3	4
T	T	T	T	**F**	T	3	8	4
T	T	T	**F**	T	T	3	4	8
T	T	**F**	T	T	T	3	4	3
T	**F**	T	T	T	T	4	3	3
F	T	T	T	T	T	3	3	4

At the predicate level, the postconditions have a total of 15 predicates. But most of them are duplicated, particularly for the purposes of testing. For testing, the relational operator ignored, so, for example, S1=S2 is equivalent to S1≠S2. There are only six distinct predicates, and all of the test case derivation tables for these predicates are not listed here. The total number of test cases generated by this method is 68, 31 of which are unique. These test cases are listed in Appendix A.

3.3.1 Triang Mutation Coverage Results

As an initial evaluation of the specification-based test generation technique, the quality of the tests was measured using a coverage criterion. Mutation testing is considered one of the strongest testing techniques, and is commonly used as a method for evaluating tests [WGS94, TFWC91, Nta84, RZ89]. The test sets are evaluated on the basis of their *mutation score*, which is the ratio of the number of mutants killed over the total number of mutants. Experience has shown that mutation scores of over 90% are difficult to achieve, and mutation scores of over 95% are very difficult.

The Mothra mutation system was used [DGK+88, DO91], and all mutants were generated for an implementation of `Triang`. Mothra created 842 non-equivalent mutants, and the 31 test cases killed 817 mutants, for a mutation score of 97.03%. Additionally, analysis of the relevant mutants showed that the test data set is completely adequate for the extended branch coverage criterion (also known as multiple condition coverage). While these numbers are only for one small program, they are extremely encouraging, and lead us to hope that this kind of specification-based test data generation scheme can yield tests that do very well at structural code coverage.

4 Conclusions

This paper has introduced a new technique for generating test data from formal software specifications. Formal specifications represent a significant opportunity for testing because they precisely describe the functionality of the software in a form that can be easily manipulated by automated means. This paper addresses the problem of developing formalizable, measurable criteria for generating test cases from specifications. A model for generating tests from *SOFL specifications* was presented, and a case study of a software unit was presented. This case study was evaluated using mutation analysis, and it was found that the technique can be very effective. This result indicates that this technique can benefit software developers who construct formal specifications during development.

4.1 Future Work

Our immediate goal is to develop formal criteria for specification-based test data generation. A subsequent goal is to develop *mechanical* procedures to derive test cases from formal specifications; long term goals include automated tool support to transform formal functional specifications into effective test cases. A future goal is to build an automatic test data generation tool for this technique. We also expect to provide more evidence of the effectiveness of this technique, and attempt to apply it at the system level.

One disadvantage of the current technique is that the predicates must be assumed to be in disjunctive normal form. While it is true that all predicates can be put into DNF, it is sometimes not desirable to change the form of the specifications. The advantage of DNF is that the predicates have a simple form, and the algorithms for satisfying them are relatively straightforward. We are currently considering other representation forms that will allow tests to be generated without modifying the form of the expressions.

5 Acknowledgments

We would like to thank Steve Miller and Paul Ammann for insightful discussions on this work. We should also thank the Ministry of Education of Japan for funding this project under Joint Research Grant-in-Aid for International Scientific Research FM-ISEE (08044167).

References

[AA92] N. Amla and P. Ammann. Using Z specifications in category partition testing. In *Proceedings of the Seventh Annual Conference on Computer Assurance (COMPASS 92)*, Gaithersburg MD, June 1992. IEEE Computer Society Press.

[ABD+79] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.

[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.

[AO94] P. Ammann and A. J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg MD, June 1994. IEEE Computer Society Press.

[BB96] M. Blackburn and R. Busser. T-VEC: A tool for developing critical systems. In *Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96)*, pages 237–249, Gaithersburg MD, June 1996. IEEE Computer Society Press.

[BCFG86] L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *The Journal of Systems and Software*, 6(4):343–360, November 1986.

[Ber91] G. Bernot. Testing against formal specifications: A theoretical view. Technical report LIENS-91-1, LIENS, Départment de Mathématiques et d'Informatique, January 1991.

[BGM91] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[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.

[Bro87] F. B. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[Cho86] N. Choquet. Test data generation using a prolog with constraints. In *Proceedings of the Workshop on Software Testing*, pages 51–60, Banff Alberta, July 1986. IEEE Computer Society Press.

[CR83] L. A. Clarke and D. J. Richardson. The application of error-sensitive testing strategies to debugging. In *Symposium on High-Level Debugging*, pages 45–52. ACM SIGSOFT/SIGPLAN, March 1983.

[DeM78] Tom DeMarco. *Structured Analysis and System Specification.* Yourdon Inc., New York, 1978.

[DF91] R. K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 165–177, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of FME '93: Industrial-Strength Formal Methods*, pages 268–284, Odense, Denmark, 1993. Springer-Verlag Lecture Notes in Computer Science Volume 670.

[DGK$^+$88] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[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.

[GMH81] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[Hay86] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.

[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.

[Hie97] Robert M. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification, and Reliability*, 7:19–33, 1997.

[HKL+95] M. Hlady, R. Kovacevic, J. J. Li, B. R. Pekilis, D. Prairie, T. Savor, R. E. Seviora, D. Simser, and A. Vorobiev. An approach to automatic detection of software failures. In *Proceedings of the IEEE 6th International Symposium on Software Reliability Engineering (ISSRE)*, pages 314–323, Toulouse-France, October 1995.

[Jal92] P. Jalote. Specification and testing of abstract data types. *Computer Language*, 17(1):75–82, 1992.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Kem85] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.

[Lay92] G. Laycock. Formal specification and testing: A case study. *The Journal of Software Testing, Verification, and Reliability*, 2:7–23, 1992.

[LS95] Shaoying Liu and Yong Sun. Structured methodology + object-oriented methodology + formal methods: Methodology of SOFL. In *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems*, pages 137–144, Ft. Landerdale, Florida, U.S.A., November 1995. IEEE Computer Society.

[LS96] J. J. Li and R. E. Seviora. Automatic failure detection with conditional-belief supervisors. In *Proceedings of the IEEE 7th International Symposium on Software Reliability Engineering (ISSRE 96)*, pages 4–13, White Plains, NY, October 1996.

[LSOHS98] Shaoying Liu, Yong Sun, A. J. Offutt, and Chris Ho-Stuart. SOFL: A formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering*, 24(1), January 1998. Special Issue on Formal Methods.

[LYZ94] Luqi, H. Yang, and X. Zhang. Constructing an automated testing oracle: An effort to produce reliable software. In *Proceedings of IEEE Conference on Computer Software and Applications (COMPSAC)*, 1994.

[Nta84] S. C. Ntafos. An evaluation of required element testing strategies. In *Proceedings of the Seventh International Conference on Software Engineering*, pages 250–256, Orlando FL, March 1984. IEEE Computer Society.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[OI95] A. J. Offutt and A. Irvine. Testing object-oriented software using the category-partition method. In *Proceedings of the Seventeenth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA '95)*, pages 293–303, Santa Barbara, CA, August 1995.

[OP97] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.

[OSW86] T. J. Ostrand, R. Sigal, and E. J. Weyuker. Design for a tool to manage specification-based testing. In *Proceedings of the Workshop on Software Testing*, pages 41–50, Banff Alberta, July 1986. IEEE Computer Society Press.

[RHC76] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.

[RZ89] J. Rowland and Y. Zuyuan. Experimental comparison of three system test strategies preliminary report. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 141–149, Key West Florida, December 1989. ACM SIGSOFT 89.

[SC93a] P. Stocks and D. Carrington. The ISDM case study: A dependency management system (specification-based testing). Technical report, The University of Queensland, Department of Computer Science, 1993.

[SC93b] P. Stocks and D. Carrington. Test template framework: A specification-based testing case study. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 11–18, Cambridge MA, June 1993.

[SC93c] 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.

[SC96] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[SCS97] H. Singh, M. Conrad, and S. Sadeghipour. Test case design based on Z and the classification-tree method. In *Proceedings of the First International Conference on Formal Engineering Methods*, pages 81–90, Hiroshima, Japan, November 1997. IEEE Computer Society.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall Publishing Company Inc., 1989.

[TFWC91] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: Deterministic versus random input generation. In *Fault-Tolerant Computing: The Twenty-First International Symposium*, pages 410–417, Montreal, Canada, June 1991. IEEE Computer Society.

[TVK90] W. T. Tsai, D. Volovik, and T. F. Keefe. Automated test case generation for programs specified by relational algebra queries. *IEEE Transactions on Software Engineering*, 16(3), March 1990.

[WC80] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6(3):247–257, May 1980.

[WGS94] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

A Appendix: Test Cases for Triang

S1	S2	S3
-1	1	1
0	1	1
1	-1	1
1	0	1
1	1	-1
1	1	0
1	1	1
1	1	2
1	1	3
1	2	1
1	2	3
1	3	1
1	3	2
2	1	1
2	2	3
2	3	2
2	3	4
2	3	5
2	3	6
2	4	3
2	5	3
2	6	3
3	1	1
3	1	2
3	2	2
3	2	4
3	4	2
4	2	3
4	3	2
5	2	3
6	2	3