Generating Test Cases From Software Documentation

By
Shilei Liu, B. Eng.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Engineering
Department of Electrical and Computer Engineering
McMaster University

MASTER OF ENGINEERING(2001) McMaster University
(Computer) Hamilton, Ontario

TITLE: Generating Test Cases From Software Documentation

AUTHOR: Shilei Liu, B. Eng.(Hunan University, China)

SUPERVISOR: Dr. Martin von Mohrenschildt

NUMBER OF PAGES: ix, 59

Abstract

In this thesis, we develop tools and methods to assist in verification of large software systems. In particular, we will apply the complete coverage principle to generate test cases automatically from tabular expressions used in software documentation.

Multi-dimensional expressions called tables are used in software documentation so that formal documentation can replace conventional documentation to achieve unambiguousness and precision and to ensure that the software will be trustworthy. Tables represent relations [11]. Certain cells contain predicate expressions that partition the domain. According to the complete partition coverage principle, we choose a single test case as representative of a partition. Since there is no particular reason to choose one element over another as a class representative, we will choose the test case randomly within the partition.

In order to develop a general algorithm to effectively generate test cases, all the numeric expressions in the guard cells have to be linear expressions, because there is no efficient method of solving the nonlinear programming problem in full generality, nor a collection of methods such that at least one of them could be assigned to any given NLP problem [13].

Acknowledgments

I wish to express my sincere appreciation to my supervisor, Dr. Martin von Mohrenschildt, for his guidance and support throughout this research work. His ideas and thoughts helped in formalizing the basis of this research, and his invaluable suggestions and patience made it possible for me to finish this thesis.

I am grateful to Dr. Mohrenschildt and Dr. Tom Luo for their suggestions in the test case generation method proposed in this thesis. I would also thank Dr, David L. Parnas, for his guidance in software documentation and testing.

Finally, I would like to thank my husband, Weimin, for his support and encouragement. I would also thank Mr. Ronald Kolthoff for his many suggestions in preparing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computer systems are increasingly incorporated into all aspects of our lives and business. They are also widely used in safety-critical applications including medical devices, automobiles, aircraft, and industrial process controls. In these type of applications, software failures can cause injury or loss of life. The correct behavior of software is crucial to the safety and well being of people and business. Consequently, there is an increasing need for the application of strict engineering discipline to the development of software systems [23].

Human beings, however, are fallible. Even if they adopt the most sophisticated and thoughtful design techniques, erroneous results can never be avoided *a priori*. Consequently, software products, like the products of any engineering activity, must be *verified* against its requirements throughout its development [5].

One fundamental approach to verification is experimenting with the behavior of a product to see whether the product performs as expected. It is common practice to input a few sample cases (test cases), which are usually randomly generated or empirically selected, and then verify that the output is correct. However, it cannot provide enough evidence that the desired behavior will be exhibited in all remaining cases. The only testing of any system that can provide absolute certainty about the

correctness of system behavior is *exhaustive* testing, i.e., testing the system under all possible circumstances. Unfortunately, such testing can never be performed in practice [5].

We need to perform testing by a systematic and practical approach, theoretically called the *complete coverage principle*. This testing criterion attempts to group elements of the input domain into classes such that the elements of a given class behave in exactly the same way. This way, we can choose a single test case as representative of each class. If we divide the input domain into disjoint classes, we say the classes constitute a *partition* [5] or an *equivalence partitioning* [16] of the domain, then there is no particular reason to choose one element over another as a class representative [5].

In this thesis, we will develop tools and methods to assist in verification of large software systems. In particular, we will apply the complete coverage principle to generate test cases automatically from tabular expressions used in software documentation advocated by the Software Engineering Research Group (SERG) of McMaster University [22, 11, 1] and others [10, 2].

As in any other engineering disciplines, it is important to document the design of software products, from the initial development through the maintenance period that follows. Documentation is used in design reviews, to guide the programmers, to guide the users and to save cost when the software has to be extended or modified [11]. Multi-dimensional expressions called tables are used in documentation so that formal documentation can replace conventional documentation to achieve preciseness and unambiguousness and to ensure that the software will be trustworthy.

Tables represent relations [11]. Certain cells contain predicate expressions that partition the domain. These cells determine which final cell(s) specifies the value of the table [1]. According to complete coverage principle, we choose a single test case as representative of a partition. Since there is no particular reason to choose one element over another as a class representative, we will choose the test case randomly within the partition.

Although it is well known that "Programming testing can be used to show the presence of bugs, but never to show their absence" [19], it is widely agreed that testing is an important step in the software process. Unfortunately, testing is time consuming and costly – as much as 50% of the development costs for a project can be attributed to testing. It seems natural, therefore, that any set of tools intended to improve the software development process will include tools to aid testing [23].

The method discussed above is in the field of Computer Aided Software Engineering (CASE). The main prerequisite for tools that automatically generate test cases is the availability of a formal description. Tools of this type are still in an early, experimental stage in industry, and few test case generators are available commercially. Hopefully, this thesis can contribute to improving this situation.

1.2 Introduction to Testing

1.2.1 Concepts

Software Testing is the execution of code using combinations of input and state selected to reveal bugs. Software testing can be viewed as a problem in systems engineering. It is the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs [4]. *Testing tool* is a software application that supports testing.

The objects that we test can be requirements and design specifications, modules of code, data structures, and any other objects that are necessary for the correct development and implementation of our software. In this chapter, We will use the term "product" to refer to the objects.

We view a product as a representation of a relation. The function describes the relationship of an input to an output. The *domain* is the set of values that input may take and the *range* is the set of values that output may take.

1.2.2 How Much Testing Is Enough Testing?

We observe that, in general, the only testing of any system that can provide absolute certainty about the correctness of system behavior is *exhaustive* testing, i.e., testing the system under all possible circumstances [5]. Unfortunately, such testing can never be performed in practice. Thus, we need *statistical* testing, i.e., using testing strategies - some criterion for selecting significant test cases.

Although we could not exhaust all the possible input values, we can group the elements of the input domain into *equivalence classes*. An equivalence class is a set of input values such that any value within the class behaves the same way as all the other values in that class. This way, we can choose a single test case as representative of each class.

If we can divide the input domain into equivalence classes, we say the class constitute a *partition* or an *equivalence partitioning* of the domain. This approach is called the *complete partition coverage principle*, because it covers all the partitions exhaustively.

Besides equivalence relation partitioning, there are other methodologies such as boundary-value analysis [16] and binomial testing [3] that are complimentary.

1.2.3 Why Testing Should Be Automated?

Test automation is software that automates any aspect of testing of an application system. It includes capabilities to generate test inputs and expected results, to run test suites without manual intervention, and to reveal pass/no pass.

How can we run these tests in a timely and efficient manner? Our only hope is to automate testing as much as possible. Automated testing offers many significant advantages such as [4]:

- Every time the software is changed, the automated test can be generated accordingly. On the other hand, the manual testing is not repeatable and rewriting is time consuming, tedious and error prone.

- The test process information produced by manual testing is often inconsistent and fragmentary.
- Automated test is the only repeatable and efficient way to generate a large quantity of input and to evaluate a large quantity of output.
- Randomly generated test can greatly improve tester productivity.
- The cost of test automation is typically recovered after two or three projects from increased productivity and the avoided costs associated with buggy software.

1.2.4 The Limits of Testing

Limited by some fundamental properties of software systems, there are limitations for testing such as [4]:

- In advocating proof of correctness, Dijkstra observed that "Programming testing can be used to show the presence of bugs, but never to show their absence"[19].
- We can never be sure that a testing system is correct. That is, bugs in test design, an oracle, or test drivers can produce spurious results.

1.3 Test Case Generation and Related Work

Test case generation is the critical step in testing. A *test case* specifies the pretest state of the product under test and its environment, as well as the test inputs or conditions. A *test suite* is a collection of test cases, typically related by a testing goal or implementation dependency.

The test case generation can generate not only input to exercise the software, sometimes the properties of the corresponding output can be specified too. In this thesis, only the generation of input data is concerned. However, the method and algorithm as well as the tool developed in this thesis can be easily extended to

determine whether an <input, output> pair satisfies the relation described by the specification.

For determining whether the output from a program is correct, there is some mechanism, an oracle, which can also be automated. In [23], an automated Test Oracle Generator (TOG) tool is developed that, given a relational program specification using tabular expressions, the tool will produce a program that will act as an oracle. The oracle program will take an <input, output> pair as input and will return *true* if the pair satisfies the relation described by the specification, or *false* if it does not.

An analysis of black-box testing techniques is described in [26], such as boundary-value analysis [16]. Later binomial method was proposed in [3] and a tool based on this method was developed in [18]. While our method is selecting test cases randomly within a partition, they select test data according to rules based upon certain assumptions. Both of these are complimentary to, but different from, the work described in this thesis.

The equivalence partitioning is usually regarded as black box testing. However, because tabular expressions can be used to document all the software documentations involved in all stages of software life cycle [21], including software requirements as well as module internal design and implementations [8], this method can also be used to test system specification, i.e., consistency, and to test module internal design. For automating the table completeness and consistency checking process, see [12].

1.4 Assumptions

Although test case generation method works for almost all the table types that fit in the formal semantics of tabular expressions [11], the current implementation of the Test Case Generation Tool (TCAT) can only handle one or two dimensional tables. However, the algorithms can be easily extended to handle higher dimensional tables as well as nested tables.

In order to develop a general algorithm to effectively generate test cases, all the numeric expressions in the guard cells have to be linear expressions, because there is no efficient method of solving the nonlinear programming problem in full generality, nor a collection of methods such that at least one of them could be assigned to any given NLP problem [13]. However, for value cells, it could be anything including Nonlinear systems.

Although quantifiers can be handled by our method, the current implementation of TCAT only deals with quantifier-free expressions.

1.5 Thesis Outline

Chapter 2 introduces the notation and terminology used in this thesis. Chapter 3 describes the method of test case generation. Chapter 4 discusses the design of a Test Case Generation Tool (TCAT). Chapter 5 presents the results, limitations, further discussions, future work and conclusions.

Chapter 2

Notation and Terminology

2.1 Predicate Logic

The first-order predicate logic used for software documentation is defined in [20]. To deal with domains of discourse containing objects of different types, a many-sorted logic used for tabular notations is defined in [25]. The following is a brief description of the concepts and definitions used in later chapters. Readers can refer to [27] for more information.

2.1.1 Basic Concepts

A *relation* is a set of tuples. A *binary relation* R is a set of ordered pairs (2-tuples). The *domain* of a binary relation R is the set of values that appear as the first element of R. The *range* of a binary relation R is the set of values that appear as the second element of R.

A *function* is a binary relation with one additional property: for any given simple tuple, x, in its domain, there is only one pair (x, y) in the function.

A *predicate* is a function whose range contains only two members: *true* and *false*.

2.1.2 The Syntax of Predicate Logic

The many-sorted Algebra A_Σ

Given a signature Σ. The many sorted algebra has for each sort s a *carrier* set A_s and each Σ-function symbol f of type $s_1 \times s_2 \times \ldots \times s_m \to s$ a function f^A: $A_{s1} \times A_{s2} \times \ldots \times A_{sm} \to A_s$, the interpretation of f in A.

Definition 2.1 (Term of Σ)

Given a set of typed variables $x_1 :: s_{i1}$, $x_2 :: s_{i2}$, ..., $x_n :: s_{in}$, terms of type s of the many sorted signature Σ are:
(a) A constant of Σ is a term of type s.
(b) A variable $x_i :: s$ is a term of type s.
(c) If f is a Σ-function symbol of type $s_{i1} \times s_{i2} \times \ldots \times s_{in} \to s$ and t_k are terms of type s_{ik} then f(t_1, t_2, ..., t_n) is a term of type s.

Definition 2.2 (Substitution, Constant Substitution)

A substitution is a set of expressions of the form $x_i \mapsto p_i$ where x_i is a variable of type s_j and p_i is a term of the same type s_j of the algebra A_Σ. A constant substitution is a set of expressions of the form $x_i \mapsto a$ where x_i is a variable of type s_j and $a \in A_{sj}$.

2.1.3 The Semantics of Predicate Logic

Definition 2.3 (State)

Let V be a set of variables. A state σ over V is a mapping $\sigma = \{x_i \mapsto a_i\}$ where x_i is a variable of type s_j and $a \in A_{sj}$.

States, sometimes called constant substitution, gives a semantical meaning to the terms.

Definition 2.4 (Evaluation of Function)

A function of FA is a substitution of A_Σ. *A constant function is a constant substitution of* A_Σ. *A function* F *is evaluated under the constant substitution* V *with* $\mathbf{var}(f) \subseteq \mathbf{rangevar}(\mathbf{V})$ *to* $val(F, V)$ *resulting in a constant function.*

Note: the mapping $val(t, V)$ is defined by structure induction on t:
(a) If $var(t) = \{\}$ then $val(t, V) = t$.
(b) If $t = x_i$, then $val(t, V)$ is the right hand side of the substitution $x_i \mapsto p_i$ of V.
(c) $val(f(t_1, t_2, \ldots, t_{si})) = f(val(t_1, V), val(t_2, V), \ldots, val(t_{si}, V))$.

val represents *simultaneous substitution*, the right hand side of any expression in the substitution is not affected by any other substitution in the set of substitutions V. For Example, $val(x_1, \{x_1 \mapsto x_2, x_2 \mapsto x_3\})$ is x_2 and not x_3.

2.2 Tabular Expressions

Tabular notation is one of the cornerstones of the relational model for documenting the intended behaviors of programs [11]. This section describes the structure and the formal semantics of tables within a general framework [1].

A tabular expression T contains the following elements.

- A raw table skeleton (H_1,...,H_n,G).
- A table predicate rule, p_T.
- A table relation rule, r_T.
- A cell connection graph, CCG.
- A mapping ψ, from cells to expressions.
- A table composition rule, C_T.

		H2	
	h_1^2	h_2^2	h_3^2
h_1^1	$g_{1,1}$	$g_{1,2}$	$g_{1,3}$
h_2^1	$g_{2,1}$	$g_{2,2}$	$g_{2,3}$
H1			G

Table 2.1: Raw Table Skeleton

Raw Table Skeleton

The shape of a table can be described independently of the contents of the table cells. Tables contain sets of cells composed of two kinds of grids: one-dimensional *headers* and a multi-dimensional *main grid*. A *raw table skeleton* is a collection of headers and a main grid, all containing empty cells.

Table 2.1 illustrates a raw table skeleton. Here the headers are H_1 and H_2 where $H_1 = \{h_i^1 | i \in \{1,2\}\}$, $H_2 = \{h_j^2 | j \in \{1,...,3\}\}$. The main grid is $G = \{g_{ij} | i \in \{1,2\}, j \in \{1,...,3\}\}$, and the raw table skeleton is a tuple $T = (H_1, H_2, G)$, where (H_1, H_2) are headers and G is the main grid.

Cell Connection Graph

Cell connection graph provides information about how to read a table. Certain cells contain predicate expressions that partition the domain. These cells are usually read first, and determine which final cells contain the value of the table. The flow of information within a table can be characterized by a *cell connection graph* (CCG). Figure 2.1 illustrates the CCG of two kinds of tables: case 1 is a normal table and case 2 is an inverted table. For more CCGs for other tables, please refer to [1] for more information.

Table Rules

Table rules are used to build larger expressions from the expressions con-

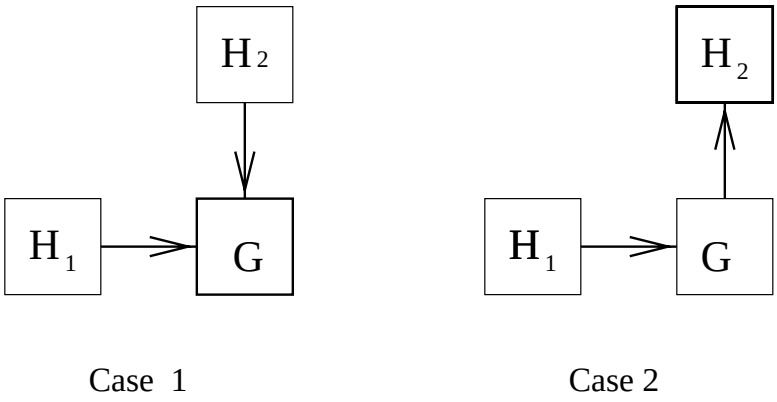

Figure 2.1: Examples of Cell Connection Graphs

tained in the cells. There are two table rules:

- The *table predicate rule*, P_T, determines the domain of the expression, and is composed of the table components containing *guard* cells (the cells containing predicates that restrict the domain of the relation).
- The *table relation rule*, r_T, determines the value of the expression, and is composed of the table components containing *value* cells (the cells containing relation definitions).

For the tables showed in figure 2.1, the table predicate rules are:

- P_T: case 1: $H_1 \wedge H_2$, case 2: $H_1 \wedge G$
- r_T: case 1: G, case 2: H_2

The table rules for various kinds of tables used in different projects and the corresponding examples are summarized in reference [1].

Mapping ψ

ψ is a mapping which assigns a predicate expression, or part of it, to each guard cell, and a relation expression, or part of it, to each value cell. The mapping

ψ describes the contents of all cells.

The Table Composition Rule C_T

The tabular expression T is interpreted as a relation. The table composition rule represents the input / output relation defined by the table.

Chapter 3

Test Case Generation Method

3.1 Overview

Prior to discussing the design of a test case generation tool (chapter 4), we must explain the method of test case generation.

First, all the partitions of the input domain should be generated. As illustrated in section 3.2, they can be derived from the *table predicate rule* R_T.

The symbolic manipulation is then applied to simplify the conditions and to transform the conditions into a formula in disjunctive normal form (DNF). This is stated in section 3.3.

We, then, will generate at least one test suite for each partition. This is carried out by substituting each occurrence of a variable with a value associated with that variable in the partition. Because the conditions are grouped in DNF and each conjunction part is independent of each other, we can generate one test suite for each conjunction part to have a more complete coverage. The methods and algorithms for this step are described in section 3.4.

	H2		
	y=10	y>10	y<10
x≥1	0	y^2	-y^2
x<1	x	x+y	x-y
H1			G

Table 3.1: Normal Function Table

3.2 Partition Generation

3.2.1 Tabular Notation Survey and Methods of Generating Partitions

This section is a survey of tabular notations that are used to denote software behaviors and the survey covers six projects from five different organizations [1]. The discussion is based on the *table predicate rule (P_T)* of the generalized model of table semantics introduced in Chapter 2.

1. $H_1 \wedge H_2$

For Normal Function Table and Predicate Expression Table used by Software Engineering Research Group (SERG); State Transition Table used by Ontario Hydro - Safety Critical Software (AECL).

Table 3.1 is an example of a Normal Function Table used by SERG. The table predicate rule P_T is $H_1 \wedge H_2$, H_1 and H_2 are two headers.

The partitions of Table 3.1 are illustrated in Table 3.2. Here the two headers $H_1 = \{h_i^1 | i \in \{1,2\}\}$ *and* $H_2 = \{h_j^2 | j \in \{1,2,3\}\}$ determinate the domain of the table. There are six partitions in this table:

		H2		
		h_1^2	h_2^2	h_3^2
		↓	↓	↓
h_1^1	→	$h_1^1 \wedge h_1^2$	$h_1^1 \wedge h_2^2$	$h_1^1 \wedge h_3^2$
h_2^1	→	$h_2^1 \wedge h_1^2$	$h_2^1 \wedge h_2^2$	$h_2^1 \wedge h_3^2$
H1				G

Table 3.2: Partitions of Table 3.1

$$
\begin{aligned}
&h_1^1 \wedge h_1^2 : (x \geq 1 \wedge y = 10),\\
&h_1^1 \wedge h_2^2 : (x \geq 1 \wedge y > 10),\\
&h_1^1 \wedge h_3^2 : (x \geq 1 \wedge y < 10),\\
&h_2^1 \wedge h_1^2 : (x < 1 \wedge y = 10),\\
&h_2^1 \wedge h_2^2 : (x < 1 \wedge y > 10),\\
&h_2^1 \wedge h_3^2 : (x < 1 \wedge y < 10).
\end{aligned}
$$

If we define the *Cartesian product* of two sets X and Y, as $X \times Y = \{(x \wedge y) \mid x \in X \wedge y \in Y\}$, then the partitions of Table 3.1 can be written as the Cartesian product of two sets H_1 and H_2: $H_1 \times H_2$, where $H_1 = \{h_i^1 | i \in \{1,2\}\}$ *and* $H_2 = \{h_j^2 | j \in \{1,2,3\}\}$. That is, Partition$(H_1 \wedge H_2) = \{(h_1^1 \wedge h_1^2)\ , (h_1^1 \wedge h_2^2),\ (h_1^1 \wedge h_3^2),\ (h_2^1 \wedge h_1^2),\ (h_2^1 \wedge h_2^2),\ (h_2^1 \wedge h_3^2)\}$.

Generally, if the table predicate rule (P_T) of a table T is $H_1 \wedge H_2$, where $H_1 = \{h_i^1 | i \in \{1,...,m\}\}$ and $H_2 = \{h_j^2 | j \in \{1,...,n\}\}$ are headers, the partitions of table T are the combinations of H_1 and H_2, denoted by $H_1 \wedge H_2$; or the Cartesian product of set H_1 and H_2, denoted by $H_1 \times H_2$.

Table 3.3 shows the generated partitions of Table 3.1.

	y=10	$y > 10$	$y < 10$
	↓	↓	↓
$x \geq 1$ →	$x \geq 1 \wedge y = 10$	$x \geq 1 \wedge y > 10$	$x \geq 1 \wedge y < 10$
$x < 1$ →	$x < 1 \wedge y = 10$	$x < 1 \wedge y > 10$	$x < 1 \wedge y < 10$

Table 3.3: Generated Partitions for Table 3.1

	H2	
	$switch = open$	$switch = closed$
$(x \geq 5) \vee (mode = A)$	$valve \neq high$	$x > 10$
$(x < 5) \vee (mode = B)$	$high = low$	$x \leq 10$
H1		**G**

Table 3.4: Inverted Function Table

2. $H_1 \wedge G$

For Inverted Function Table and Inverted Predicate Expression Table used by SERG; Condition Table, Event Table (1 and 2) and Mode Transition 1 Table used by Software Cost Reduction (SCR) project.

Table 3.4 is an example of an Inverted Function Table used by SERG. The table predicate rule is $H_1 \wedge G$, H_1 is a header and G is the main grid.

The partitions of Table 3.4 are illustrated in Table 3.5. Here the two headers $H_1 = \{h_i^1 | i \in \{1, 2\}\}$ *and* $G = \{g_{ij} | i \in \{1, 2\}, j \in \{1, 2\}\}$ determinate the domain of the table.

If we define $G_i = \{g_{ij} | j \in \{1, \ldots, n\}\}$, then here $G_1 = \{g_{1,1}, g_{1,2}\}$ and $G_2 = \{g_{2,1}, g_{2,2}\}$. Thus the partitions of Table 3.4 can be written as the union of $h_i^1 \times G_i$: $\cup_{i=1}^{m}(h_i^1 \times G_i)$. That is, Partition$(H_1 \wedge G) = \{(h_1^1 \wedge g_{1,1}) \vee (h_2^1 \wedge g_{2,1})$,

		$(h_1^1 \wedge g_{1,1}) \vee (h_2^1 \wedge g_{2,1})$	$(h_1^1 \wedge g_{1,2}) \vee (h_2^1 \wedge g_{2,2})$
		$\uparrow$	$\uparrow$
h_1^1	$\rightarrow$	$g_{1,1}$	$g_{1,2}$
h_2^1	$\rightarrow$	$g_{2,1}$	$g_{2,2}$

Table 3.5: Partitions of Table 3.4

		$((x \geq 5) \vee (mode = A) \wedge valve \neq high) \vee ((x < 5) \vee (mode = B) \wedge high = low)$	$((x \geq 5) \vee (mode = A) \wedge x > 10) \vee ((x < 5) \vee (mode = B) \wedge x \leq 10)$
		$\uparrow$	$\uparrow$
$(x \geq 5) \vee (mode = A)$	$\rightarrow$	$valve \neq high$	$x > 10$
$(x < 5) \vee (mode = B)$	$\rightarrow$	$high = low$	$x \leq 10$

Table 3.6: Generated Partitions for Table 3.4

$(h_1^1 \wedge g_{1,2}) \vee (h_2^1 \wedge g_{2,2})\}$.

Generally, if the table predicate rule (P_T) of a table T is $H_1 \wedge G$, where $H_1 = \{h_i^1 | i \in \{1, ..., m\}\}$ and $G_i = \{g_{ij} | j \in \{1, ..., n\}\}$ are headers, the partitions of table T are the combinations of H_1 and G, denoted by $H_1 \wedge G$; or the union of set H_1 and G, denoted by $\cup_{i=1}^{m}(h_i^1 \times G_i)$.

Table 3.6 shows the generated partitions of Table 3.4.

3. H_2

For Vector Function Table, Vector Relation Table and Mixed Vector Table used by SERG; Horizontal Condition Table used by AECL, etc.

Generally, if the table predicate rule (P_T) of a table T is H_2, where $H_2 = \{h_j^2 | j \in \{1, ..., n\}\}$ is a header, the partitions of table T are H_2; or the members of set H_2, denoted by $H_2 = \{h_j^2 | j \in \{1, ..., n\}\}$.

4. H_1

For Selector Table (1, 2 and 3) used by SCR; Vertical Condition Table used by AECL, Condition Table used by Ontario Hydro - Control Program (OHCP), etc.

Generally, if the table predicate rule (P_T) of a table T is H_1, where $H_1 = \{h_i^1 | i \in \{1, ..., m\}\}$ is a header, the partitions of table T are H_1; or the members of set H_1, denoted by $H_1 = \{h_i^1 | i \in \{1, ..., m\}\}$.

5. Other Partitioning Rules

The methods can be easily extended to other tables as long as they fit the formal semantics of the tabular expression model introduced in chapter 2.

3.2.2 Partition Generation Summary

The summary of the *table predicate rule (P_T)* for different kinds of tables and the partitions generated are shown in Table 3.7.

P_T	Partitions	Table Types
$H_1 \wedge H_2$	$H_1 \times H_2$, where	Normal Function
	$H_1 = \{h_i^1 \mid i \in \{1, ..., m\}\}$	Predicate Expression
	$H_2 = \{h_j^2 \mid j \in \{1, ..., n\}\}$	State Transition
$H_1 \wedge G$	$\cup_{i=1}^{m}(h_i^1 \times G_i)$, where	Inverted Function
	$H_1 = \{h_i^1 \mid i \in \{1, ..., m\}\}$	Inverted Predicate
	$G_i = \{g_{ij} \mid j \in \{1, ..., n\}\}$	Expression
H_2	H_2, where	Vector Function
	$H_2 = \{h_j^2 \mid j \in \{1, ..., n\}\}$	Vector Relation
H_1	H_1, where	Selector
	$H_1 = \{h_i^1 \mid j \in \{1, ..., m\}\}$	Condition

Table 3.7: Partition Generation for Different Kinds of Tables

3.3 Symbolic Manipulation

As discussed in section 3.2, the partitions we generated are expressions in the form of the intersections of several conditions. We shall manipulate the expressions so that they can be interpreted in a more efficient way for our test case generation purpose. In particular, we will construct equivalent formulas in Disjunctive Normal Form (DNF).

In order to find a generic method that guarantees to solve the problem effectively under all the circumstances, all the quantifiers in the original formulas should be eliminated so that the original formulas can be constructed to equivalent quantifier-free formulas. The reason is described as follows:

Theorem 3.1

For every quantifier-free formula φ, we can effectively construct equivalent formulas δ and γ of the following form:

δ is a finite disjunction of finite conjunctions of literals:

$$\delta = \bigvee_{1 \le i \le n} \chi_i, \text{ with } \chi_i = \bigwedge_{1 \le j \le m(i)} L_{ij} \text{ with literals } L_{ij}$$

γ is a finite conjunction of finite disjunctions of literals:

$$\gamma = \bigwedge_{1 \le i \le n} \chi_i, \text{ with } \chi_i = \bigvee_{1 \le j \le m(i)} L_{ij} \text{ with literals } L_{ij}$$

δ is called a formula in **disjunctive normal form** and γ is called a formula in **conjunctive normal form**.

Proof It can be proved by structural induction over φ [27].

A *literal* is either an atomic formula $p(t_1, \ldots, t_n)$ (also called a positive literal) or a negated atomic formula $\neg p(t_1, \ldots, t_n)$ (also called a negative literal).

We have both positive and negative literals in this thesis. The atomic formulas include $Equal(=)$, $Less(<)$, $Greater(>)$, $LessEqual(\leq)$ and $GreaterEqual(\geq)$.

Remarks on quantifiers:

In order to effectively construct an expression in DNF, we should first transform the expression to a quantifier-free expression. The following is an informal description about how to eliminate the quantifiers for expressions containing variables over finite types.

There are two kinds of quantifiers, one is the *existential quantifier($\exists$)* and the other one is the *universal quantifier($\forall$)*.

If x is of type s and all the elements in s are finite ($s1$, $s2$, ..., sn), then $\{\exists \mathrm{x}|p(x)\}$ is equivalent to $p(s1) \vee p(s2) \vee \ldots \vee p(sn)$. Similar, $\{\forall \mathrm{x}|p(x)\}$ is equivalent to $p(s1) \wedge p(s2) \wedge \ldots \wedge p(sn)$. Clearly such formulas could be larger.

Quantifiers were not considered originally in the design of the Test Case Generation Tool as described in Chapter 4. Fortunately, with the design of modules and the separation of concerns, the tool can be easily extended to handle quantifiers without changing the interfaces.

3.4 Test Case Generation

3.4.1 Algebra Analysis

For test case generation purpose, we will generate one test suite for each conjunction part of the DNF derived from the previous sections

$$\chi_i = \bigwedge_{1 \leq j \leq m(i)} \mathrm{L}_{ij} \text{ with literals } L_{ij}$$

and χ_i itself is a *Conjunctive Normal Form*, which we will call *condition* c:

$$c = c_1 \wedge c_2 \wedge \ldots \wedge c_m, \quad c_1, c_2, \ldots, c_m \text{ are literals.}$$

A condition c is a function whose range contains only two members: *true* and *false*.

3.4.2 Data Type Analysis

Of all the data types, we will divide them into two groups: *numeric* type and *finite set* type. The reason why we divide this way will be more clear in the subsequent sections.

1. Numeric Type

The data types *Integer, Fraction, Rational, Float or Real*, etc. all represent numeric information. Although most numeric types are infinite (e.x. *Real*), they can also be finite (e.x. *Subrange* of integers).

An example of the numeric type variables and their operations is: $(x \leq 3.6) \wedge (y = 7)$, or *And(LessEqual(x,3.6), Equal(y,7))* in prefix notation where x is a real number and y is an integer.

2. Finite Set Type

Finite set type consists of finite possible data values except numeric ones (e.x. *subrange* of integers). *Enumerated, set* and *list* types defined in some programming languages are examples of finite set type. Finite set type cannot be numeric and must be finite.

An example of finite set type variables and their operations is: *(switch = on) or (mode* $\neq$ *B)* or *Or(Equal(switch,on), NotEqual(mode,B))* in prefix notation where $switch \in \{on, off\}$ and $mode \in \{A, B, C, D, E\}$.

Note: *boolean* type can be considered as a special finite set type.

For example: a boolean type variable b can be considered as a finite set type variable with $b \in \{\ true, false\ \}$.

3. Group Numeric Type and Finite Set Type Variables

From the formula of *condition* c in section 3.4.1,

$$c = c_1 \wedge c_2 \wedge \ldots \wedge c_m, \quad c_1, c_2, \ldots, c_m \text{ are literals,}$$

We can split numeric type and finite set type variables and group them together to transform to the following form:

$$c = (\ N_1 \wedge \ldots \wedge N_p\ ) \wedge (\ S_1 \wedge \ldots \wedge S_q) \text{ where}$$

$N = N_1 \wedge \ldots \wedge N_p$, $N_1 \ldots, N_p$ are literals and $var(N)$ are numeric type variables.
$S = S_1 \wedge \ldots \wedge S_q$, $N_1, \ldots, N_q$ are literals and $var(S)$ are finite set type variables.

3.4.3 Generating Test Cases for Numeric Type Variables

The following is a typical example of generating test cases for numerical type variables:

$$(x_1 + 1 < 5) \wedge (x_2 + x_3 > 3.7\ ) \wedge \ldots \wedge (x_n \geq 0).$$

If all the expressions in the tables are linear expressions, the problem of generating test cases is in fact solving the following linear system and is very similar to a *linear programming*(LP) problem:

$$\begin{cases} \sum_{j=1}^{n} a_{1j}x_j \leq b_1 \\ \\ \sum_{j=1}^{n} a_{2j}x_j \leq b_2 \\ \\ \ldots \\ \\ \sum_{j=n}^{n} a_{1j}x_j \leq b_n \end{cases} \qquad i.e. \quad Ax \leq b$$

If some of the expressions in the tables are not linear expressions, it becomes an *nonlinear programming* problem (NLP). There is no efficient method of solving the nonlinear programming problem in full generality, nor a collection of methods

such that at least one of them could be assigned to any given NLP problem [13]. The major hurdle is that the problem cannot be regarded as a sequence of unconstrained problems in a linear subspace. The curvature of the nonlinear constraints must be considered. It is a nontrivial matter to follow the boundary of a region defined by nonlinear constraints [14]. Finding a universally applicable (and efficient) method for NLP seems to be beyond the scope of our knowledge at present [13].

In the absence of general algorithms for nonlinear programming, it lies near at hand to explore the possibilities of approximate solution by linearization [7]. We now have very efficient algorithms for solving linear programming problems and very high-quality software for these algorithms. In the following sections, we will develop a general algorithm to effectively generate test cases for linear systems.

3.4.3.1 Linear Programming and Simplex Method

Linear Programming model is one of *optimization* or *mathematical programming* models, in which a linear objective function is to be optimized subject to linear equality and inequality constraints and sign restrictions (or lower and/or upper bounds) on the decision variables [15]. The following is a brief introduction to *Linear Programming* (LP) and the *simplex method*.

1. Problem

$$\text{Minimize (or Maximize) } z(x) = cx$$

$$\text{subject to} \qquad Ax \leq b$$

where A, b, c are given matrices or vectors.

2. Transform the Problem into the Standard Form

Before applying the simplex method on an LP, all the constrains in it on which pivot operations are carried out must be transformed into equality constrains. Then, the simplex method is used to solve LPs in a particular form known as *the standard form*:

x_1	x_2	$\cdots$	x_n	b
a_{11}	a_{12}	$\cdots$	a_{1n}	b_1
a_{21}	a_{22}	$\cdots$	a_{2n}	b_2
$\vdots$	$\vdots$		$\vdots$	$\vdots$
a_{m1}	a_{m2}	$\cdots$	a_{mn}	b_n

Table 3.8: Original Table

x_{m+1}	$\cdots$	x_n	x_1	x_2	$\cdots$	x_m	$\overline{b}$
$\overline{a}_{1,m+1}$	$\cdots$	$\overline{a}_{1n}$	1	0	$\cdots$	0	$\overline{b}_1$
$\overline{a}_{2,m+1}$	$\cdots$	$\overline{a}_{2n}$	0	1	$\cdots$	0	$\overline{b}_2$
$\vdots$		$\vdots$	$\vdots$		$\vdots$		$\vdots$
$\overline{a}_{m,m+1}$	$\cdots$	$\overline{a}_{mn}$	0	0	$\cdots$	1	$\overline{b}_m$

Table 3.9: Canonical Table

$$Ax = b$$

with all variables and right-hand side are nonnegative.

The standard form $Ax = b$ (nonsingular) can be described as the Original Table 3.8.

3. Canonical Form

The original table, 3.8, is in Canonical form if there is an identity matrix of size $m \times m$ among the first m rows of the original table. When this occurs it is easy to read out a feasible solution of the problem known as a *basic feasible solution* (BFS) from the table. The significance of BFSs will be explained later.

If the original table is not in canonical form, it is called *the Phase I problem* and should be transformed into canonical form. For the transformation method, see [15].

4. Initial Basic Feasible Solutions (BFS)

When the table is in canonical form, it is called *the Phase II problem.* An initial Basic Feasible Solution (BFS) can be obtained directly:

Initial stage: BFS:

$$\text{All non basic variables } x_i = 0,\ i = m+1, ..., n$$
$$i\text{-th basic variable} = \overline{b}_i,\ i = 1, ..., m$$

The BFS can be used as a starting feasible basic vector to solve any LP problem in the standard form to optimize the object function. The details can be found in [15].

5. Integer Programming

If the variables are integer type, the problem becomes the *integer linear programming* problem, and it belongs to NP hard problems [24].

3.4.3.2 Algorithms For Test Case Generation

1. Problem

Randomly sample the solution spaces of $Ax \leq b$, with x being a vector, A and b being given matrices and vectors.

This problem is solved using methods of Linear Programming. For the purpose of our tool, x should be *randomly* generated.

<table>
<tr><td>No Feasible Solution</td><td colspan="3">Has Feasible Solution</td></tr>
<tr><td rowspan="5">Message “Conditions contradict.”</td><td>One Solution</td><td colspan="2">More Than One Solution</td></tr>
<tr><td rowspan="4">Solve $Ax = b$</td><td>Sign Restrictions Only</td><td>Otherwise</td></tr>
<tr><td rowspan="3">Generate directly</td><td colspan="2">Transform into the Standard Form $Ax = b$</td></tr>
<tr><td>In Canonical Form</td><td>Not In Canonical Form: Transform to Canonical Form</td></tr>
<tr><td colspan="2">1. Get initial BFS
2. Obtain the adjacent extreme points of a BFS
3. Randomly Generate a feasible point</td></tr>
</table>

Table 3.10: The Approach for Solving $Ax \leq b$

2. The Mathematical Approach

Table 3.10 describes the approach using vertex solution to randomly generate an x such that $Ax \leq b$.

Figure 3.1 illustrates how to randomly generate a feasible point once two of the adjacent extreme points of an initial BFS is given. Let the solution space be K, then the line segment joining any pair of points in K lies entirely in K. Suppose an initial BFS is A, and any two adjacent extreme points of A are B and C. The method of randomly generating a feasible point P in K is as follows:
(1). Get the middle point of B and C, denoted by D.
(2). Get the line segment joining A and D.
(3). Get any point on the line segment, denoted by P.

$$P = \lambda(D - A),\ 0 < \lambda < 1 \text{ is randomly generated.}$$

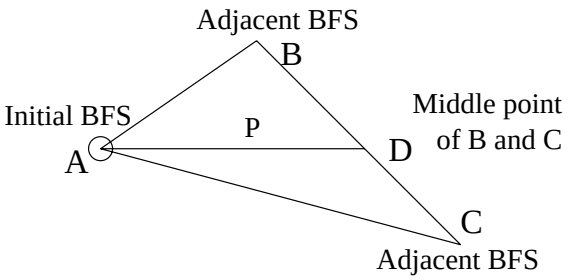

Figure 3.1: Randomly Generate a Feasible Point

3. The Approach Based on the Maple's Simplex Package

Using the routines, libraries, packages and especially the simplex package from the Maple software product [6], we can greatly simplify our algorithm and make it more efficient. Table 3.11 is an overview of the method used to solve the problem using Maple.

<table>
<tr><td>No Feasible Solution</td><td colspan="3">Has Feasible Solution</td></tr>
<tr><td rowspan="3">Message “Conditions Contradict.”</td><td>One Solution</td><td colspan="2">More Than One Solution</td></tr>
<tr><td rowspan="2">Solve $Ax = b$</td><td>Sign Restrictions Only</td><td>Otherwise</td></tr>
<tr><td>Generate directly</td><td>1. Simplification
2. Transform into the Standard Form $Ax = b$
3. Randomly generate slack variables
4. Substitute slack variables to $Ax = b$ and solve
5. Substitute variables to $Ax \leq b$ to double check</td></tr>
</table>

Table 3.11: Randomly Generate x for $Ax \leq b$ Using Maple

3.4.4 Generating Test Cases for Finite Set Type Variables

The following is a typical example for generating test cases for finite set type variables:

$$(switch = on) \wedge (mode \neq B) \wedge (high = high) \wedge \ldots \wedge (valve = level).$$

Requirements

Before applying the algorithm of generating test cases for finite set type variables, we must first check to see if all the operators are binary (either $Equal(=)$ or $NotEqual(\neq)$). Also all the operands should be finite set type. Otherwise an error message will be given.

Algorithm of Generating Test Cases for Finite Set Type Variables

Table 3.12 is an overview of the algorithm for generating test cases for Finite Set type Variables. It covers all the possibilities.

The details of randomly generating test cases for finite set variables need to be fully explained here. From all the cases in table 3.12, the main case will be $f(a, b)$, where $f \in \{Equal, NotEqual\}$, one of a or b will be a variable, and the other one will be a constant. For example, $Equal(mode, A)$ and $NotEqual(valve, low)$.

There are two steps for this algorithm.

1. A hash table needs to be established. For example, the hash table of the previous example will be: table(["$mode'' = A$
"$valve'' = high, normal$])

How to get one entry of the table:

(1) If $f = Equal$, then the left hand side will be the variable name, and the right hand side will be the constant name.

<table>
<tr><td colspan="6">$f(a,b),\ f \in \{Equal, NotEqual\}$, a and b :</td></tr>
<tr><td colspan="2">both are constants</td><td colspan="2">one constant
one variable</td><td>both are
variables</td><td>Otherwise</td></tr>
<tr><td>True</td><td>False</td><td>False</td><td>True</td><td rowspan="3">“Not
In
Yet”</td><td rowspan="3">Error
Message
“Not
Boolean”</td></tr>
<tr><td>Purge</td><td>Message
“Conditions
Contradict”</td><td>Message
“Conditions
Contradict”</td><td>1. Look up in hash table
2. Randomly generated
(main)</td></tr>
<tr><td colspan="4">Examples:</td></tr>
<tr><td>$Equal(A,A)$

$NotEqual(A,B)$</td><td>$Equal(A,B)$

$NotEqual(A,A)$</td><td>$Equal(mode,A)$
and
$Equal(mode,B)$</td><td>$Equal(mode,A)$
and
$NotEqual(valve,low)$</td><td></td><td></td></tr>
</table>

Table 3.12: Overview of the Algorithm of Generating Test Cases for Finite Set Type Variables

(2) If $f = NotEqual$, then the left hand side will also be the variable name and the right hand side needs a little bit calculation. For example, if the type for "$valve$" is $ModeT$ (defined in a separate input type file), and all the possible values fos for $ModeT$ are "$high$", "$normal$" and "low", then all the possible values for variable "$valve$" will be "$high$", "$normal$" and "low". Therefore,

the right hand side = {all the possible values} "$minus$" {constant name}

For example, if $f(a, b) = NotEqual(valve, low)$, then the entry of the table will be "$valve$" $= \{high, normal, low\} - \{low\} = \{high, normal\}$.

2. Randomly get a test case from the hash table.

(1) Get the number of elements (Noi) from the right hand side of the table entry. For example, in the previous hash table example, $No1 = 1$ and $No2 = 2$.

(2) Randomly generate an integer (tmp) from 1 to Noi.

(3) Select the tmp-th element as the generated test case. For example, if $tmp1 = 1$, $tmp2 = 2$, then the generated test suite will be ("$mode$" $= A$) and ("$valve$" $= normal$)

Chapter 4

Test Case Automation Tool Design (TCAT)

This chapter now describes the Test Case Automation Tool (TCAT). Developed as part of this thesis, TCAT generates test cases automatically from the input table and is implemented using Maple [6].

Figure 4.1 illustrates the overall design of the tool.

4.1 Requirements

4.1.1 Input

There are two input files for TCAT. The first file contains the table from which the test cases are to be generated. The format of the table file should be in the format of Appendix A [17]. It consists of the table type, dimension and table cells. The type of the input table is one of NORMAL, INVERTED, VECTOR or SELECTOR. Otherwise the program prints "Wrong Table Type". The dimension of the table is one or two. Otherwise the program prints "Wrong Dimension". All the headers of the input table have to be boolean expressions. Otherwise the program prints "Not Boolean".

The second file contains type definitions and variable declarations. The

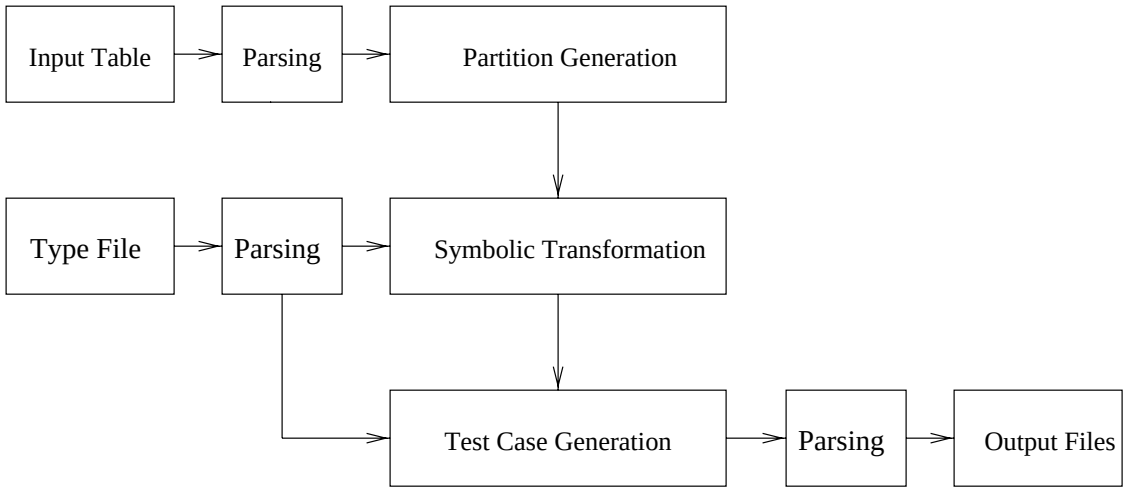

Figure 4.1: Overall Design

format of the type file should be in the format of Appendix B[17]. Otherwise there will be an error message "File Format Error". All the variables have to be declared before they can be used. Otherwise there will be error messages "Not A Type" or "'Unknown Type'.

4.1.2 Output

There is one output file for TCAT. The generated test cases together with their table indexes are stored in an output file (the name of the file is provided by the user). There are two kinds of format for output file, one is table format and the other one is text format. As illustrated in Appendix D, there are three types of test cases:

1. {} means the partition is true. There is no test case generated in this situation.

2. {contradiction} means there is a contradiction in the partition. This can be used to indicate there is an inconsistency in the headers.

3. $\{var = constant, \ldots, var = constant\}$ is the test cases. The variables and the

constants are of the same type.

4.1.3 Limitations

Currently TCAT can only handle one or two dimensional tables. It cannot handle nested tables. However, the tool can be extended to handle higher dimension tables and nested tables as long as they fit the formal semantics of tabular expression model introduced in chapter 2 [1, 11, 17].

There is no quantifier in the cells that contain conditions and all the numeric expressions in the cells are linear expressions.

4.2 Module Guide

The TCAT is implemented by a set of *modules,* each of which encapsulates a set of design decisions. Thus the design is easier to understand because of the separation of concerns. It is also easier to change since the decisions affected by the changes are likely to be isolated.

Figure 4.2 illustrates the overall modular structure of TCAT and states the relationships among the individual modules using the USES relation [5, 23, 17].

For any two distinct modules A and B, we say that A USES B if and only if at least one access program in A requires the correct execution of at least one access program in B.

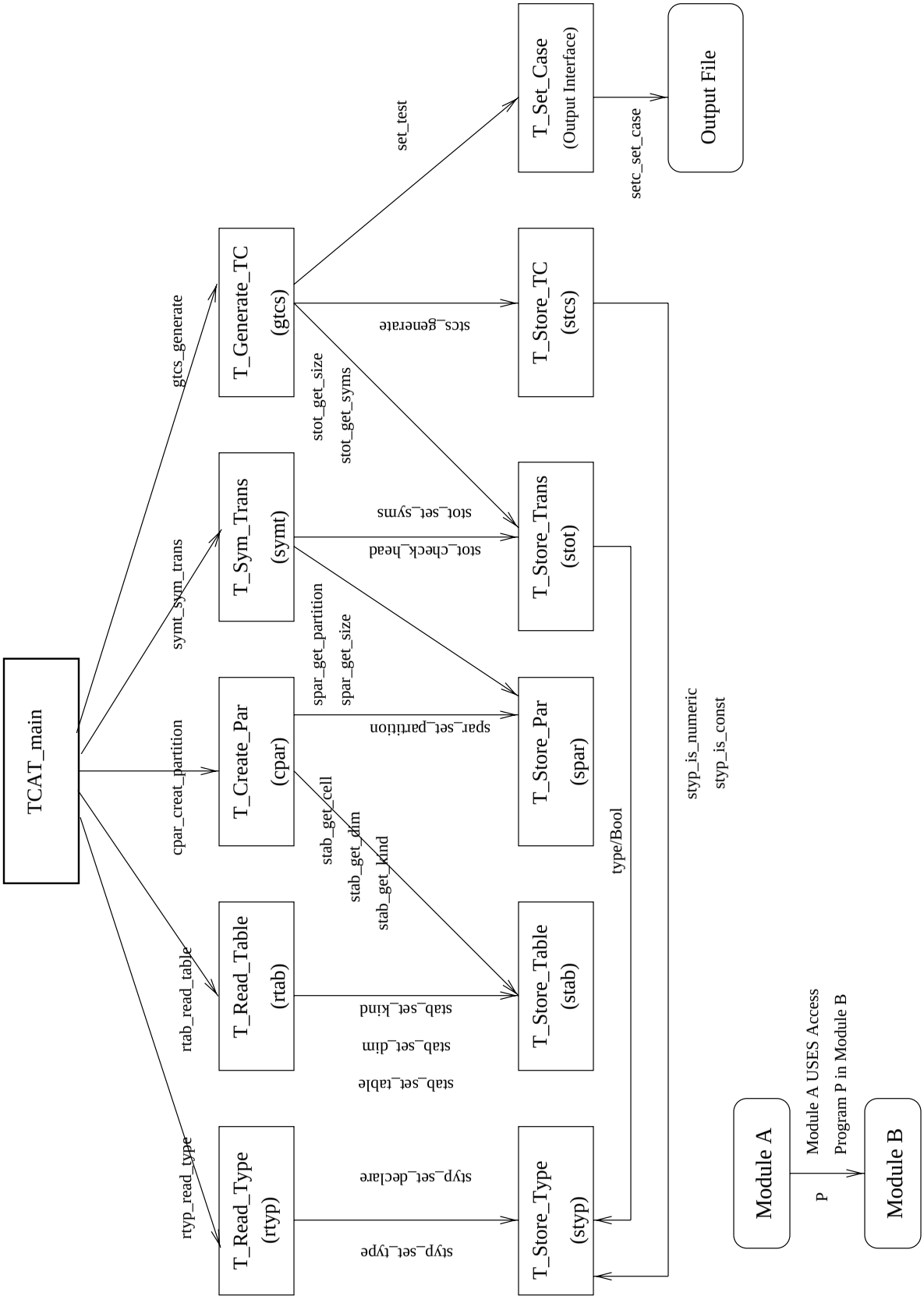

Figure 4.2: The Overall Modular Structure and Design

4.2.1 Read Table (T_Read_Table)

Uses module: *T_Store_Table*

This module reads from an input file consisting of table information. The format of the input *table* file is the secret of the module.

Access Program	Description
rtab_read_table(file)	Reads the input table in *file* and calls the access programs in T_Store_Table to store the table

4.2.2 Store Table (T_Store_Table)

This module encapsulates the semantic information of the input table. The table semantic contains the kind of the table, the dimension and the contents of the table. The kind of a table is one of NORMAL, INVERTED, VECTOR or SELECTOR which represents one of the four table predicate rules: $H_1 \wedge H_2$, $H_1 \wedge G$, H_2 and H_1. The secret of the module is the data structure used to store tables. The likely change is the addition of new kinds of tables.

Access Program	Description
stab_set_table(table)	Stores the contents of the input table in *table*
stab_set_dim(dim)	Stores the dimension of the input table in *dim*
stab_set_kind(kind)	Stores the kind of the input table in *kind*
stab_get_cell(i,j)	Returns the element at the i th row, j th column of the table.
stab_get_dim()	Returns the dimension of the table.
stab_get_kind()	Returns the kind of the table.

4.2.3 Read Type (T_Read_Type)

Uses module: *T_Store_Type*

This module reads from an input file consisting of type definitions and variable declarations. The format of the input *type* file is the secret of this module.

Access Program	Description
rtyp_read_type(typefile)	Reads the input type information in *typefile* and calls the access programs in T_Store_Type to store the type information

4.2.4 Store Type (T_Store_Type)

This module stores the type information of the input type file. It provides access programs to store type definitions and variable declarations. It also checks whether a given expression is boolean. Furthermore, it determines whether an expression is constant or numeric. The data structure used to store the type information is the secret of this module. The likely changes are the data structure used to store the type information and the addition of boolean type (for example, quantifiers).

Access Program	Description
styp_set_type(tname,tdef)	Stores user defined types and its definition
stab_set_declare(var,tname)	Stores the variable declaration: the variable name and its type
styp_is_const(expr)	Returns *true* if *expr* is a mathematical constant, or a member of a user defined type
styp_is_numeric(expr)	Returns *true* if *expr* is a mathematical constant, or a variable defined by the user as an *Integer* or *Real*, or an expression with numeric operands and numeric operators (defined recursively)
type/Bool(expr)	Extends the maple built in function *type* to determine whether the given expression is of type *boolean*. Returns *true* if *expr* is a: 1. Boolean constant or Boolean variable defined by user 2. Boolean operator (And, Or, Not) and its operands are of type Bool (recursive definition) 3. Relational operator (Less, Greater, Equal, GreatEqual, LessEqual) and operands are *Numeric* type 4. A relational operator (Equal, NotEqual) and operands are of the same *Finite Set* type
styp_get_type(var)	If the variable *var* is a user defined type (defined in *typefile*), it returns the corresponding type. If the variable is a numeric variable, it returns "Numeric". Otherwise returns "Unknown Type".

4.2.5 Create Partition (T_Create_Par)

Uses module: *T_Store_Table*

This module creates all the partitions of the input table. It is able to generate partitions from four kinds of table: NORMAL, INVERTED, VECTOR or SELECTOR. The algorithm used to generate partitions is the secret of the module. The likely changes is the addition of table kinds.

Access Program	Description
cpar_creat_partition()	Generates all the partitions of the input table and calls the access programs in T_Store_Par to store the partitions.

4.2.6 Store Partition (T_Store_Par)

This module is used to store all the partitions generated from the input domain. In order to distinguish different partitions, each partition is associated with an index. The secretes of the module is the data structure used to store the partitions and the index method. The likely changes are the encapsulated data structure and the index method.

Access Program	Description
spar_set_partition(ind,par)	Stores partition *par* with its associated index *ind.*
spar_get_size()	Returns the total number of the partitions generated from the input table.
spar_get_partition(ind)	Returns the partition associated with index *ind.*

4.2.7 Symbolic Transformation (T_Sym_Trans)

Uses module: *T_Store_Par, T_Store_Trans*

This module invokes other access programs to get the partitions, to check the partitions, to do transformation and to store the results. If one of the partitions is not a boolean expression, the program prints an error message and stops execution. Otherwise it will invoke access programs in module T_Store_Trans to do the transformation and to store the results.

Access Program	Description
symt_sym_trans()	Invokes other access programs to get the partitions, to check the partitions, to do transformation and to store the results.

4.2.8 Store Transformation (T_Store_Trans)

This module encapsulates the methods and algorithms for symbolic simplification and transformation. As a "syntax" simplification, it first simplifies all the expressions using *predicate logic*, then transforms the simplification to a DNF. The secrets of this module are the algorithms for symbolic transformation and the data structure used to store the transformation.

Access Program	Description
stot_check_head(par)	Checks if the partition is a boolean expression. Returns *true* if it is a boolean expression, otherwise returns *false*.
stot_set_sym(ind,par)	Simplifies and transforms the partition to a DNF with *Numeric* type variables and *Finite Set* variables group together. Stores the transformation with its associate index.
stot_get_size()	Returns the total number of the transformations stored.
styp_get_syms(ind)	Returns the transformation associated with index *ind*.

4.2.9 Generate Test Case (T_Generate_TC)

Uses module: *T_Store_Trans, T_Store_TC, T_Set_Case*

This module gets the transformed DNF, generates test cases for them and invokes access programs of output interface to put the generated test cases with their associated indices into an output file.

Access Program	Description
gtcs_generate()	Gets transformations, generates test cases and puts the generated test cases with their associated indices into an output file.

4.2.10 Store Test Case (T_Store_TC)

This module encapsulates the methods and algorithms for storing test cases. First the DNF derived from other modules is transformed to another DNF with *Numeric* type variables and *Finite Set* variables grouped together. Then different algorithms are applied to different groups to generated test cases. The secrets of this module are the algorithms used to generate *Numeric* type and *Finite Set* type test cases and the data structure used to store the test cases.

Access Program	Description
stcs_generate(par)	Groups *Numeric* type and *Finite Set* type variables and generate test cases separately. If the conditions contradict each other, then error "Conditions Contradict", otherwise test cases will be generated.

This module is divided into two sub modules: numeric_case and set_case for generating Numeric type and Finite Set type variables separately. The generating methods are described in detail in Chapter 3.

4.2.11 Output Test Case (T_Set_Case)

This module is used to write the generated test cases to an output file for further retrieval. It serves as an interface of the program and the output file.

Access Program	Description
set_set_case(ind,list)	Writes a list of test cases and their associated index to an output file. Note there will be more than one set of test suites associate with one index.

Chapter 5

Conclusions

5.1 Results

We have developed methods and algorithms for test case generation as described in chapter 3. TCAT successfully applied the methods and implemented the algorithms as described in chapter 4. We generated some test cases using a couple of tables. Appendix C is the result of two of them.

5.2 Future Work

As described in the previous section, the TCAT has been tested and evaluated using some small tables which has shown that the methods are viable in these cases. If we can apply the TCAT to a wide variety of industrial software applications, we could draw more general conclusions about the viability and usefulness of the methods.

Experience has shown that there are some auxiliary predicates and functions that frequently appear in specifications of the form used in this work. For example, it is often the case that a specification states that some set of variables are not changed-denote by the auxiliary predicate "NC" in [8]. In this work the specific form of this predicate must be specified in the documentation. It would be convenient if this definition could be produced automatically from a shorthand notation as used in [8] [23].

At present, TCAT can not handle quantifiers including universal ($\forall$) and existential ($\exists$) quantifiers. In the future, quantifiers could be handled using quantifier elimination method as given by *Trash and Storm* [9].

In TCAT, all the numeric expressions in the guard cells have to be linear expressions. More work could be done to handle some classes of nonlinear expressions.

5.3 Conclusions

In this thesis, we developed tools and methods to assist in verification of large software systems. In particular, we applied the complete coverage principle to generate test case automatically from tabular expressions used in software documentation.

The method discussed here is in the field of Computer Aided Software Engineering (CASE). The main prerequisite for tools that automatically generate test cases is the availability of a formal description. Tools of this type are still in an early, experimental stage in industry, and few test case generators are available commercially. Some work has been done in SERG [23, 18] to contribute to improving this situation.

Appendix A

Input Table Format

Note: This format is based on the table format defined in [17].

A.1 Format

DIM:=[$\langle N1 \rangle \langle N2 \rangle$];
T_TYPE:=$\langle kind \rangle$;
cell[i,j]:=$\langle exp \rangle$;
...
...

N1 and N2 are positive integers, represent dimensions.
kind is one of NORMAL, INVERTED, VECTOR, or SELECTOR.
i and j are positive integers such that
$0 \leq$ i $\leq$ N1 and $0 \leq$ j $\leq$ N2
expr is any expression.

A.2 Example of Two Input Tables

The normal function table 3.1 in chapter 3.

```
DIM:=[2,3];
T_TYPE:=NORMAL;
cell[1,0]:=GreatEqual(x,1);
cell[2,0]:=Less(x,1);
cell[0,1]:=Equal(y,10);
cell[0,2]:=Greater(y,10);
cell[0,3]:=Less(y,10);
cell[1,1]:=Equal(f,0);
cell[1,2]:=Equal(f,y*y);
cell[1,3]:=Equal(f,-y*y);
cell[2,1]:=Equal(f,x);
cell[2,2]:=Equal(f,x+y);
cell[2,3]:=Equal(f,x-y);
```

The inverted function table 3.4 in chapter 3.

```
DIM:=[2,2];
T_TYPE:=INVERTED;
cell[1,0]:=Or(GreatEqual(x,5),Equal(mode,A) );
cell[2,0]:=Or(Less(x,5),Equal(mode,B));
cell[0,1]:=Equal(switch,open);
cell[0,2]:=Equal(switch,closed);
cell[1,1]:=NotEqual(valve,high);
cell[1,2]:=Greater(x,10);
cell[2,1]:=Equal(high,low);
cell[2,2]:=LessEqual(x,10);
```

Appendix B

Input Type File Format

Note: This format is based on the type format defined in [17].

B.1 EBNF Grammar for Type Definitions

$$
\begin{array}{lcl}
Syntax & \longrightarrow & \{TypeDef\}\ \{VarDef\}^{1} \\
TypeDef & \longrightarrow & \text{“}TYPE\text{”}\ TypeName\ :\ SetDef \\
SetDef & \longrightarrow & \text{“}Set\text{”}\ \text{“}\{\text{”}\ SetEle\ \{\ \text{“,”}\ SetEle\ \}\ \text{“}\}\text{”} \\
SetEle & \longrightarrow & String \\
TypeName & \longrightarrow & String \\
VarDef & \longrightarrow & VarName\ \text{“}:\text{”}\ Type \\
Type & \longrightarrow & TypeNmae|\text{“}Integer\text{”}|\text{“}Real\text{”}|\text{“}Bool\text{”}
\end{array}
$$

B.2 Example of an Input Type File

```
TYPE ModeT : Set {'high','low','normal'}
TYPE Mode2 : Set {A,B}
TYPE STYPE : Set {'open','closed'}
valve : ModeT
level : ModeT
switch : STYPE
mode : Mode2
a : Bool
c : Integer
lower : Real
upper : Real
x : Integer
y : Integer
f : Integer
pressure : Real
```

Appendix C

Sample Output of Test Case Automation Tool

The sample output for the normal function table 3.1 in chapter 3 (also in appendix A).

```
    |\^/|     Maple V Release 5 (WMI Campus Wide License)
._|\|   |/|_. Copyright (c) 1981-1997 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
# For test use
>
> printlevel:=1;
                                printlevel := 1

>
> read(‘T_Read_Table‘):
> read(‘T_Store_Table‘):
> read(‘T_Create_Par‘):
> read(‘T_Store_Par‘):
> read(‘T_Read_Type‘):
> read(‘T_Store_Type‘):
```

```
> read(‘T_Sym_Trans‘):
> read(‘T_Store_Trans‘):
> read(‘T_Generate_TC‘):
> read(‘T_Store_TC‘):
Warning, new definition for dual
Warning, new definition for maximize
Warning, new definition for minimize
> read(‘test_cases.txt‘):
>
> rtab_read_table(‘tabn‘):
> rtyp_read_type(‘type1‘):
>
> cpar_creat_partition():
> symt_sym_trans():
>
> gtcs_generate():
     Test cases to be generated: , {[{GreatEqual(x, 1), Equal(y, 10)}, {}]}

bytes used=1031676, alloc=851812, time=0.24
                       "final result: ", [{y = 10, x = 2}]

         Test cases to be generated: , {[{Less(x, 1), Less(y, 10)}, {}]}

                       "final result: ", [{y = 9, x = -1}]

    Test cases to be generated: , {[{GreatEqual(x, 1), Greater(y, 10)}, {}]}

                       "final result: ", [{y = 10, x = 4}]

      Test cases to be generated: , {[{GreatEqual(x, 1), Less(y, 10)}, {}]}

                        "final result: ", [{y = 9, x = 3}]
```

```
        Test cases to be generated: , {[{Less(x, 1), Equal(y, 10)}, {}]}

                        "final result: ", [{y = 10, x = 1}]

       Test cases to be generated: , {[{Less(x, 1), Greater(y, 10)}, {}]}

bytes used=2033408, alloc=1441528, time=0.56
                        "final result: ", [{x = 0, y = 11}]

>
>
>
>
>
> quit
bytes used=2120596, alloc=1441528, time=0.60
```

Output File please see appendix D.

Appendix D

Output File Format

D.1 Table Format Output File

1. Normal function table 3.1

Suppose the input table is the following normal function table:

H2

	y=10	y>10	y<10
x≥1	0	y^2	-y^2
x<1	x	x+y	x-y

H1 **G**

Then the Table Format Output File should be:

	y=10	$y > 10$	$y < 10$
	↓	↓	↓
$x \geq 1$ →	$\{x = 2, y = 10\}$	$\{x = 4, y = 11\}$	$\{x = 3, y = 9\}$
$x < 1$ →	$\{x = 1, y = 10\}$	$\{x = 0, y = 11\}$	$\{x = -1, y = 9\}$

There are three types of test cases:

1. {} means the partition is true. There is no test case generated in this situation.

2. {contradiction} means there is a contradiction in the partition. This can be used to indicate there is an inconsistency in the headers.
3. {var = constant, ..., var = constant} is the test cases. The variables and the constants are of the same type.

1. Inverted function table 3.1

Suppose the input table is the following inverted function table:

		H2
	switch=open	switch=closed
(x≥5) ∨ (mode=A)	valve≠high	x>10
(x<5) ∨ (mode=B)	high=low	x≤10
H1		**G**

Then the Table Format Output File should be:

		{contradiction}; {x=6, valve=low}; {mode=A, valve=low}	{x=12}; {mode=A, x=11}; {x=4}; {mode=B, x=9}
		↑	↑
(x≥5) ∨ (mode=A)	→	valve≠high	x>10
(x<5) ∨ (mode=B)	→	high=low	x≤10

D.2 Text File Format Output File

Format:
Index: [i,j]
case k: CasePairs
...
The following is the text format of the test case output file for the input table 3.2:

```
Index: [1,1]
case 1: {y = 10, x = 2}
Index: [2,3]
case 1: {y = 9, x = -1}
Index: [1,2]
case 1: {y = 11, x = 4}
Index: [1,3]
case 1: {y = 9, x = 3}
Index: [2,1]
case 1: {y = 10, x = 1}
Index: [2,2]
case 1: {x = 0, y = 11}
```

The following is the text format of the test case output file for the input table 3.5:

```
Index: [0, 1]
case 1: {contradiction}
case 2: {x = 6, valve = normal}
case 3: {mode = A, valve = low}
Index: [0, 2]
case 1: {x = 12}
case 2: {mode = A, x = 11}
case 3: {x = 4}
case 4: {mode = B, x = 9}
```

Bibliography

[1] R.F. Abraham. *Evaluating Generalized Tabular Expressions in Software Documentation.* CRL Report 346, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, February 1997.

[2] T.A. Alspaugh. *Software Requirement for the A-7E Aircraft.* Tech. Rep. NRL/FR/5546-92-9194, Naval Research Lab., Washington DC, 1992.

[3] Roman Viveros Balwant Singh and David L. Parnas. *Estimating Software Reliability Using Inverse Sampling.* CRL Report 351, McMaster University, Hamilton, Ontario, Canada, August 1997.

[4] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison Wesley Longman, Inc., 2000.

[5] M. Jazayeri C. Ghezzi and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, NJ., 1991.

[6] Bruce W. Char. *Maple V Language References Manual.* Waterloo Maple Publishing, 1991.

[7] Sven Dano. *Nonlinear and Dynamic Programming.* Springer-Verlag, 1975.

[8] Jan Madey David Lorge Parnas and Michal Iglewski. *Precise Documentation of Well-Structured Programs.* IEEE Transactions on Software Engineering, VOL.20, No. 12, December 1994.

[9] Erwin Engeler. *Foundations of Mathematics.* Springer-Verlag, 1992.

[10] D.N. Hoover and Z. Chen. *Tablewise, a Decision Table Tool.* COMPASS '95 (Proceedings of the Ninth Annual Conference on Computer Assurance), Gaithersburg, MD, June 1995.

[11] R. Janicki. *Towards a Formal Semantics of Parnas Tables.* 17th International Conference on Software Engineering, IEEE Computer Society, Seattle, WA, April 1995.

[12] Min Jing. *A Table Checking Tool.* CRL Report 384, McMaster University, Hamilton, Ontario, Canada, March 2000.

[13] Bela Martos. *Nonlinear Programming Theory and Methods.* North-Holland Publishing Company, 1975.

[14] Garth P. McCormick. *Nonlinear Programming.* John Wiley and Sons, 1983.

[15] Katta G. Murty. *Linear Programming.* John Wiley and Sons, Inc., 1983.

[16] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, Toronto, Ontario, 1979.

[17] Kavitha Nadarajah. *Semantic Equality of Tables.* CRL Report 378, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, August 1999.

[18] S. M. Reza Nejat. *Program Reliability Estimation Tool.* CRL Report 391, McMaster University, Hamilton, Ontario, Canada, October 2000.

[19] E.W. Dijkstra O.J. Dahl and C.A.R. Hoare. *Structured Programming.* Academic Press, New York, 1972.

[20] David Lorge Parnas. *Predicate Logic for Software Engineering.* IEEE Transactions on Software Engineering, VOL.19, No. 9, September 1993.

[21] David Lorge Parnas and Jan Madey. *Functional Documents for Computer Systems.*

[22] D.L. Parnas. *Tabular Representation of Relations.* CRL Report 260, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, October 1992.

[23] D.K. Peters. *Generating a Test Oracle From Program Documentation.* CRL Report 302, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, April 1995.

[24] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, Ltd., 1986.

[25] Martin von Mohrenschildt. *Algebra of Normal Function Tables.* CRL Report 350, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, May 1997.

[26] Denise Woit. *An Analysis of Black-Box Testing Techniques.* CRL Report 245, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, May 1992.

[27] Dr. Zucker. *Computer Science 769 Course Notes.* McMaster University, Hamilton, Ontario, Canada, September 1999.