Semantic Equality of Tables

By
Kavitha Nadarajah, B.Math

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(1999) McMaster University
(Computer) Hamilton, Ontario

TITLE: Semantic Equality of Tables

AUTHOR: Kavitha Nadarajah, B.Math(University of Waterloo)

SUPERVISOR: Dr. Martin von Mohrenschildt

NUMBER OF PAGES: viii, 35

Abstract

This thesis presents an algorithm to determine semantic equality of two given tables used in software documentations.

Tabular notation is seen to be a useful method for formal specification of software systems [8]. Verification of the software specification is an important part of software life cycle when software is used in safety critical environments. Industries such as Ontario Hydro invest time and money to ensure that their design of safety critical software corresponds to specification [13].

In this thesis we present an algorithm to verify whether two given tabular expressions specify the same relation. This algorithm is useful in verifying whether the design corresponds to specification. This algorithm handles all tables described in [1]. We also present the design of a prototype tool which implements this algorithm, which can help us automate the verification of software design.

Acknowledgments

I thank my supervisor Dr. Martin von Mohrenschildt for his support and enthusiasm. I am indebted to Martin for always finding time to answer my questions, for reading various drafts of this thesis, and for providing a comfortable work environment for his students.

I acknowledge professors David Parnas, Mark Lawford and Jeffrey Zucker for their helpful comments. I am grateful to my brother N.Asokan for reading this thesis and providing feed back via e-mail.

I thank Raveen for his love, patience, and support. I wouldn't have completed this thesis on time without his encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Goal

When software is used in safety critical or mission critical environments, such as in nuclear power plants, chemical plants, or air crafts, we have to make sure that the behaviour of the software corresponds to the specification. In such situations, the verification should be done as part of the software life cycle. Usually verification is done manually by one or more humans inspecting the specification and design for errors or discrepancies. However, verifying the design of complex software can be tedious and time consuming. In this thesis, we have developed and implemented an algorithm to assist in the verification of software.

Tabular Notation is used to specify relations and functions in software documentation by the Software Engineering Research Group (SERG) of McMaster University [8, 10] and others [4, 2]. A *table* describes the values of variables (state of variables) or *actions* for different *conditions*. Tables are multi-dimensional expressions, consisting of indexed sets of cells that contain other expressions. The structure of the table makes it easier for the user to understand complex conditions and actions. In Figure 1.1 tables T and $\overline{T}$ describe the same function, but they "look" different. This thesis describes a method to decide whether two given proper tables describe the same function. It also describes the design of a prototype software that impliments this method.

Comparing tables for semantic equality is very useful in software verification.

$T \equiv y =$

	$x < 0$	$x = 0$	$x > 0$
$m = A$	$1/x$	0	x
$m = B$	$-x$	0	x

$\overline{T} \equiv y =$

$x < 0 \wedge m = A$	$1/x$
$x < 0 \wedge m = B$	$-x$
$x \geq 0$	x

Figure 1.1: Two Semantically Equal Tables

Suppose a customer has a specification of a program and asks a programmer to implement the specification. To verify that the program is written according to the specification, the customer can give the source code to another designer and ask her to write the specification of it. If both specifications are same then the program is written according to the specification. This reviewing method was used at the Darlington Nuclear Power Plant [7]. Software that determines semantic equality of two given tabular specifications can be very useful in similar situations.

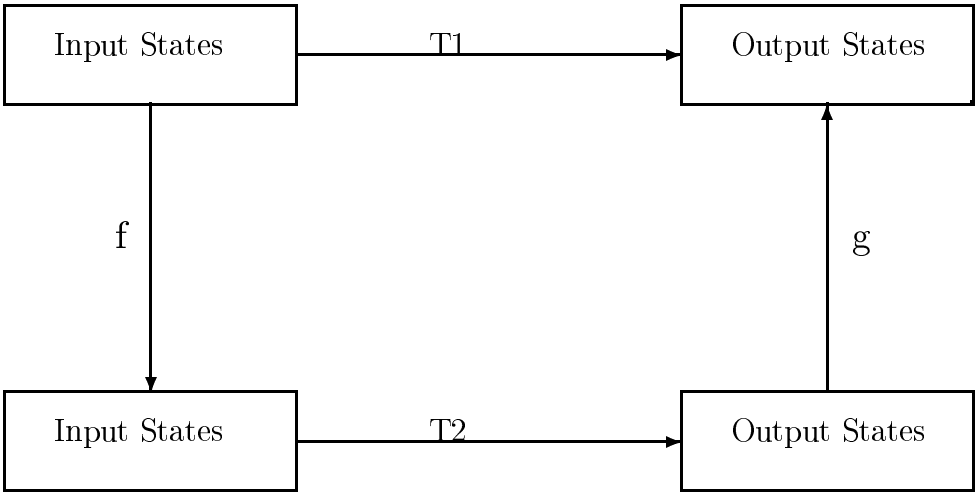

Figure 1.2: Specification vs. Design

This method can also be used to verify whether the design of a system corresponds to specification. In Figure 1.2, table T describes the specification in terms of abstract variables and table $\overline{T}$ describes the design in terms of concrete variables. An abstraction function is used to relate the abstract variables used in the specification to the concrete variables used in the design documents. The algorithm we describe in this thesis can help to verify whether the design corresponds to the specification

with respect to the abstraction functions.

Many different kinds of tables have been used in software documentations. In this thesis, we develop a *unified* algorithm for all the different kinds of proper tables described in [1]. The algorithm can handle any of these tables including nested tables. When tables are not semantically equal, this algorithm returns counter examples for diagnostic purposes.

1.2 Limitations

The algorithm we describe in this thesis works only for proper tables. Section 2.2.2 gives the definition of a proper table. The algorithm uses two dimensional tables. But the algorithm can be easily extended to handle higher dimensional tables.

The current implementation of the algorithm has some limitations. It can handle only one or two dimensional tables. It cannot handle nested tables. But the tool can be extended to handle higher dimensional tables and nested tables. The current implementation of the tool does not recognize quantifiers in conditions.

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 gives basic mathematical definitions, explains the structure and semantics of tabular expressions and introduces the concept of unified sets. Chapter 3 presents the algorithm to determine semantic equality of tables and proves that the algorithm works correctly. Chapter 4 discusses the design of Semantical Equality Checker (SEC) tool. It contains the module interface guide and informal description of access programs. Chapter 5, presents the results of using the tool and suggest future work in this area.

Chapter 2

Notation and Terminology

2.1 Elementary Definitions

In this section we give some of the mathematical concepts and basic definitions that are used in later chapters. Readers can refer to [11], [9], [12], and [14] for more information.

An *n-ary relation* is a set of n-tuples. A *binary relation* is a set of 2-tuples. The *domain* of a binary relation is the set of values that appears as the first element of the pair. The *range* of a binary relation is the set of values that appears as the second element of the pair. A *function* $\mathcal{F}$, is a binary relation with the following property: if $(x, y) \in \mathcal{F}$ and $(x, z) \in \mathcal{F}$ then $y = z$.

Boolean, Integer, and Character are examples of *types.* Functions accept arguments of certain types and map them into some type. A function with no argument is a *constant.* Each variable and constant has a type associated with it. A *term* is a variable, a constant, or a function with terms as arguments.

Now, we shall formally define *substitution* in expressions [14].

Definition 2.1 (Substitution σ) *: Let A be an expression. A substitution σ 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. Then $A\sigma$ (applying the substitution σ to the expression A) replaces every free occurrence of the variable x_i in A by the term p_i.*

2.2 Tabular Notations

This section explains the structure [8], and semantics of tabular expressions [5, 1] used in software documentation. Tables are multi-dimensional expressions, consisting of indexed sets of cells that contain expressions. Table 2.1 is an example of a normal table that describes a *count* function that counts the number of occurrences of x in an array A.

count(x,A,lower,upper) =

	$A[lower] = x$	$\neg(A[lower] = x)$	H_2
$lower < upper$	$1 + count(x, A, lower + 1, upper)$	$count(x, A, lower + 1, upper)$	
$lower = upper$	1	0	
H_1			G

Table 2.1: Example of a Normal Table

Table 2.1 contains two *headers* and a *main grid.* The header H_1 and header H_2 contain predicate expressions that partition the domain of the function. The main grid G determines the value of the function. Table 2.1 can also be described as in Figure 2.1. We note that the tabular notation is easier to understand and check for errors.

$count(x, A, lower, upper) =$

$$\begin{cases} 1 + count(x, A, lower + 1, upper) & \text{if } (lower < upper) \wedge (A[lower] = x) \\ count(x, A, lower + 1, upper) & \text{if } (lower < upper) \wedge \neg(A[lower] = x) \\ 1 & \text{if } (lower = upper \wedge (A[lower] = x) \\ 0 & \text{if } (lower = upper \wedge \neg(A[lower] = x) \end{cases}$$

Figure 2.1: Traditional Description of Table 2.1

2.2.1 Structure of Tables

A *raw table skeleton* describes the structure of a table. A raw table skeleton comprises one or more headers and a main grid.

	$H_2 = \{h_i^2 \mid i = 1,2,3\}$		
	h_1^2	h_2^2	h_3^2
h_1^1	$g_{1,1}$	$g_{1,2}$	$g_{1,3}$
h_2^1	$g_{2,1}$	$g_{2,2}$	$g_{2,3}$
h_3^1	$g_{3,1}$	$g_{3,2}$	$g_{3,3}$

$H_1 = \{h_i^1 \mid i = 1,2,3\}$ $\quad G = \{g_{i,j} \mid i = 1,2,3 \wedge j = 1,2,3\}$

Table 2.2: Raw Table Skeleton

- A header H_j is an indexed set of cells, $H_j = \{h_i^j \mid i \in I^j\}$, where $I^j = \{1,2,..k\}$ for some k is the *index set*. Intuitively, j denotes the dimension and i denotes the ith element in that dimension.
- A main grid G is an indexed set of cells $G = \{g_\alpha \mid \alpha \in I\}$ where I is the index set of G.
- A raw table skeleton is a tuple $T = (H_1, H_2, \cdots, H_n, G)$ where $H_1, H_2, \cdots, H_n$ are headers , $H_j = \{h_i^j \mid i \in I^j\}$, $j = 1, \cdots, n$ and G is the main grid indexed by $I = I^1 \times \cdots \times I^n$.
- $H_1, H_2, \cdots, H_n, G$ are called *table components.*

Only one-dimensional or two-dimensional tables are commonly used because of ease of representation. But the algorithm can be extended for higher dimensions.

A *raw element* of a table is a tuple $e_\alpha = (h_{i_1}^1, \cdots, h_{i_n}^n, g_\alpha)$ where $i_j \in I^j$, $j = 1, \cdots, n$, $\alpha = (i_1, \cdots, i_n) \in I$. A *guard cell* contains predicates that partition the domain of the relation the table describes. A *value cell* contains the values of relation the table describes. In Table 2.1 (a normal table), headers H_1 and H_2 contain guard cells and the main grid G contain the value cells.

2.2.2 Semantics of Tables

Now, we need to specify rules to interpret the contents of a table (semantics of the table). We provide two rules, the *table predicate rule* (p_T) and the *table relation rule* (r_T) to specify how to use the cell expressions. Rule p_T is a predicate expression composed of expressions in guard cells . Rule p_T partitions the domain of the expression described by the table. Rule r_T is a relation expression. It specifies the value of

the expression and is composed of the expressions in value cells. For Table 2.1, p_T is $H_1 \wedge H_2$. It means that the predicate expressions are obtained by the conjunctions of contents of each cell in header H_1 with the contents of each cell in header H_2. For Table 2.1 r_T is G. It means if a predicate expression is created for Table Table 2.1 by the conjunction of contents of cells h_i^1 and h_j^2, then the corresponding relation expression is in cell $g_{i,j}$.

The relation specified by a raw element e_α is denoted by R_α. The relation R_T that describes the whole table is given using a composition operation, $\oplus$. See [1] for more details about the composition operation. In the Table 2.1, there are four raw elements. Hence, there are four different relations. The relation that describes the entire table in Table 2.1, is
$[R_T = \{((lower < upper) \wedge (A[lower] = x), 1 + count(x, A, lower + 1, upper)), \cdots,$
$((lower = upper) \wedge \neg(A[lower] = x), 0)\}$ Hence, the composition operation that describes the entire relation R_T is a union operation.

Now we shall define proper table. Let T be a table and let $C_1, \cdots, C_n$ be the predicate expressions that partition the domain of T. The table T is proper if for every substitution on T satisfies

- at most one of $C_1, \cdots, C_n$ and
- at least one of $C_1, \cdots, C_n$

The following describes different kinds of commonly used tables and their p_T, r_T, and R_T. [1]

- Normal
  Table 2.1 is an example of a normal function table.

$$p_T : H_1 \wedge H_2$$

$$r_T : G$$

$$R_T = \cup_{\alpha \in I} R_\alpha$$

- Inverted

switch(mode,pressure)=

	open	*close*	H_2
$mode = low$	$pressure \leq 2000$	$pressure \geq 2000$	
$mode = normal$	$100 \leq pressure \leq 5000$	$pressure < 100 \vee pressure > 5000$	
$mode = high$	$pressure \leq 350$	$pressure > 350$	
H_1			G

Table 2.3: Example of an Inverted Function Table

switch(mode,pressure) =

$$\begin{cases} \text{open} & \text{if } mode = low \wedge pressure \leq 2000 \\ \text{close} & \text{if } mode = low \wedge pressure \geq 2000 \\ \text{open} & \text{if } mode = normal \wedge 100 \leq pressure \leq 5000 \\ \text{close} & \text{if } mode = normal \wedge (pressure < 100 \vee pressure > 5000) \\ \text{open} & \text{if } mode = high \wedge pressure \leq 350 \\ \text{close} & \text{if } mode = high \wedge pressure > 350 \end{cases}$$

Figure 2.2: Traditional Description of Table 2.3

Table 2.3 shows an inverted function table. Rule p_T and Rule r_T are

$$p_T : H_1 \wedge G$$

$$r_T : H_2$$

The relation R_T described by Table 2.3 is

$$R_T = R_{1,1} \cup R_{1,2} \cup R_{2,1} \cup R_{2,2} \cup R_{3,1} \cup R_{3,2}$$

In general, $R_T = \cup_{\alpha \in I} R_\alpha$

- Vector

$y_1 = -x \wedge (isnegative = true)$ if $x < 0$
$y_1 = x \wedge (isnegative = false)$ if $x \geq 0$

Figure 2.3: Traditional Description of Table 2.4

	$x < 0$	$x \geq 0$	H_2
y_1	$-x$	x	
$isnegative$	$true$	$false$	
H_1			G

Table 2.4: Example of a Vector Table

Table 2.4 shows a vector table. Rule p_T and rule r_T for a vector table are

$$p_T : H_2$$

$$r_T : H_1 = G$$

We can show that the relation described by 2.4 is

$$R_T = (R_{1,1} \cup R_{1,2}) \cap (R_{2,1} \cup R_{2,2})$$

In general, we can show that $R_T = \cap_{i=1}^{n}(\cup_{j=1}^{m} R_{i,j})$ for two-dimensional vector tables. Where n is the number of elements in Header H_1 and m is the number of elements in Header H_2.

- Decision

  Table 2.5 shows a decision table. Rule p_T and rule r_T are

$value(open, switch) =$

	a	b	c	d	H_2
$open$	$true$	$true$	$false$	$false$	
$switch$	on	off	on	off	
H_1					G

Table 2.5: Example of a Decision Table

$$p_T : H_1 = G$$

$$r_T : H_2$$

The table Relation for Table 2.5 is

$$R_T = (R_{1,1} \cap R_{2,1}) \cup (R_{1,2} \cap R_{2,2}) \cup (R_{1,3} \cap R_{2,3}) \cup (R_{1,4} \cap R_{2,4})$$

$value(open, switch) =$

$$\begin{cases} \text{a} & \text{if } (open = true) \wedge (switch = on) \\ \text{b} & \text{if } (open = true) \wedge (switch = off) \\ \text{c} & \text{if } (open = false) \wedge (switch = on) \\ \text{d} & \text{if } (open = false) \wedge (switch = off) \end{cases}$$

Figure 2.4: Traditional Description of Table 2.5

In general, R_T for a two-dimensional decision table is $R_T = \cup_{j=1}^{n} \cap_{i=1}^{m} R_{i,j}$ where n is the number of elements in Header H_1 and m is the number of elements in Header H_2.

2.2.3 Generalized Model of Table Semantics

We use the table semantics proposed in [1], [5] to describe a generalized semantics model for all tables. A table T is a tuple $T = (H_1, ..H_n, G, p_T, r_T, \psi, \oplus)$ where

- $(H_1, ..H_n, G)$ is *raw table skeleton*
- p_T is *table predicate rule*
- r_T is *table relation rule*
- ψ is mapping from cells to expressions
- $\oplus$ is a compose operation that combines all raw element relations to represent table's relation R_T

2.3 Unified Sets

We need to develop a *unified* algorithm that works for the different kinds of proper tables that are in use. We achieve this goal by transforming tables into *unified sets.* A unified set, $\mathcal{CA}$, of a table is a set of 2-tuple of *condition-action pairs.* $\mathcal{CA} = \{(Cond_i, Act_i) | i = 1, \cdots, n\}$ Where $Cond_i$ is a predicate expression and Act_i is a relation expression.

Chapter 3

Equality of tables

3.1 Proposed Approach

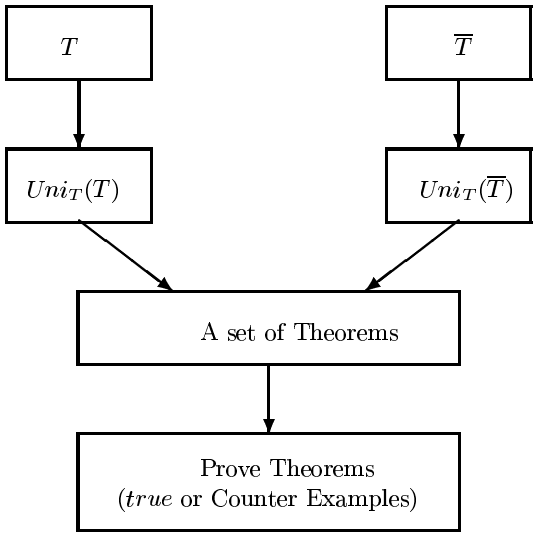

Figure 3.1: Proposed Approach

To determine semantic equality of tables, we first transform the two given proper tables into two unified sets of condition-action pairs by applying a mapping Uni_T as described in section 3.2. Hence, from input tables T and $\overline{T}$ we obtain two unified sets $Uni_T(T)$ and $Uni_T(\overline{T})$. We, then, use these unified sets to obtain a set of theorems as described in section 3.3. We then prove these theorems using a computer

algebra system. We will show that if all theorems are *true*, then the given tables are semantically equal. Section 3.5 discusses proving techniques used to prove these theorems.

First, we shall extend the definition of substitution, to *substitution in tables*.

Definition 3.1 (Substitution in Tables) *Let $T(H_1, ..H_n, G, p_T, r_T, \psi, \oplus)$ be a table and σ be a substitution. Then*

$$T\sigma = \{R_i\sigma | (P_i, R_i) \in R_T, P_i\sigma = true)\}$$

where R_T is the table's relation given using the compose operation $\oplus$.

When T is a proper table, the set $T\sigma$ contains only one element.

Definition 3.2 (Semantical equality in Tables ($\simeq$)) *: Two tables, T and $\overline{T}$ are semantically equal, $T \simeq \overline{T}$, iff $\forall\sigma\ T\sigma = \overline{T}\sigma$*

Intuitively, two tables are semantically equal, if they are equal in all possible contexts.

3.2 Generalized Table Semantics $\Longrightarrow$ Unified Set

$Uni_T : T \mapsto \mathcal{CA}$ is a mapping from a table T to a unified set $\mathcal{CA}$ such that $Uni_T(T) = \{(Cond_i, Act_i) | i = 1, \cdots, n\}$

We have discovered that the unified set is in fact R_T, that is $Uni_T(T) = R_T$. For example, for normal tables we have
$p_T : H_1 \wedge H_2$
$r_T : G$
$R_T : \cup_{\alpha \in I} R_\alpha$
We know that, $(\psi(h_j^1) \wedge \psi(h_k^2), \psi(g_{j,k})) \in R_T$, for $j = 1, \cdots, N$ and $k = 1, \cdots, M$. Then the corresponding $Cond_i$ and Act_i are,
$Cond_i = \psi(h_j^1) \wedge \psi(h_k^2)$ and $Act_i = \psi(g_{j,k})$ for $i = 1, \cdots, NM$.

Example: Let T_1 be the table in Table 2.1. Then $Uni_T(T_1)$ is

$$Uni_T(T_1) \ = \ \{((lower < upper) \wedge (A[lower] = x),\ 1 + count(x, A, lower + 1, upper)),$$

$$((lower < upper) \wedge \neg(A[lower] = x),\ count(x, A, lower + 1, upper)),$$
$$((lower = upper) \wedge (A[lower] = x),\ 1),$$
$$((lower = upper) \wedge \neg(A[lower] = x),\ 0)\}$$

Definition 3.3 (Substitution in Unified Sets) *Let $\mathcal{CA}$ be a unified set of the form*

$$\mathcal{CA} = \{(Cond_i, Act_i)|i = 1, \cdots, n\}$$

and σ be a substitution. Then

$$\mathcal{CA}\sigma = \{Act_i\sigma|Cond_i\sigma = true\}$$

Definition 3.4 (Semantical Equality of Unified Sets) *Let $\mathcal{CA}$ and $\overline{\mathcal{CA}}$ be*

$$\begin{aligned} \mathcal{CA} &= \{(Cond_i, Act_i)|i = 1, \cdots, n\} \\ \overline{\mathcal{CA}} &= \{\overline{Cond_i}, \overline{Act_i})|i = 1, \cdots, m\} \end{aligned}$$

Then, $\mathcal{CA}$ is semantically equal to $\overline{\mathcal{CA}}$, written as $\mathcal{CA} \simeq \overline{\mathcal{CA}}$, iff the set $\mathcal{CA}\sigma$ is equal to the set $\overline{\mathcal{CA}}\sigma$.

$$\forall\sigma \quad (\mathcal{CA}\sigma = \overline{\mathcal{CA}}\sigma)$$

Theorem 3.1 *Let T and $\overline{T}$ are proper tables.*

$$T \simeq \overline{T} \Leftrightarrow Uni_T(T) \simeq Uni_T(\overline{T})$$

Proof From Definition 3.2, $\forall\sigma \quad (T\sigma = \overline{T}\sigma)$. From Definition 3.1, the sets $\{R_i\sigma|P_i\sigma = true, (P_i, R_i) \in R_T\}$ and $\{\overline{R_j}\sigma|\overline{P_j}\sigma = true, (\overline{P_j}, \overline{R_j}) \in R_{\overline{T}}\}$ are equal. Hence, $\forall\sigma\ (R_i\sigma = \overline{R_j}\sigma)$ when $P_i\sigma \wedge \overline{P_j}\sigma = true$. Hence $\forall\sigma \quad (Act_i\sigma = \overline{Act_j}\sigma)$ when $Cond_i\sigma \wedge \overline{Cond_j}\sigma = true$ (from the Definition 3.4). Therefore, $Uni_T(T) \simeq Uni_T(\overline{T})$ ∎

3.3 Unified Sets $\Longrightarrow$ Theorems

From $Uni_T(T)$ and $Uni_T(\overline{T})$ we now derive a set of theorems of the form:

$$\forall\ \sigma\ (Cond_i\sigma \wedge \overline{Cond_j}\sigma \rightarrow Act_i\sigma = \overline{Act_j}\sigma)$$

where $(Act_i, Cond_i) \in Uni_T(T)$ and $(\overline{Act_j}, \overline{Cond_j}) \in Uni_T(\overline{T})$

Let $|Uni_T(T)|$, the cardinality of $Uni_T(T)$, be N and $|Uni_T(\overline{T})|$, the cardinality of $Uni_T(\overline{T})$, be M then the number of theorems will be NM.

Here $Cond_i$ and $\overline{Cond_j}$ are predicate expressions of raw elements and always evaluate to a boolean. Hence, the implication in all theorems always evaluate to $true$ or $false$.

These theorems prove equality of partially defined functions [15]. Section 3.5 discusses different strategies of proving theorems of this form.

Theorem 3.2 *Let T and $\overline{T}$ be proper tables. Let $(Cond_i, Act_i) \in Uni_T(T)$ and $(\overline{Cond_j}, \overline{Act_j}) \in Uni_T(\overline{T})$.*

$$Uni_T(T) \simeq Uni_T(\overline{T}) \Leftrightarrow \forall i\ , j, \sigma\ Cond_i\sigma \wedge \overline{Cond_j}\sigma \rightarrow Act_i\sigma = \overline{Act_j}\sigma$$

Proof $Uni_T(T) \simeq Uni_T(\overline{T})$. From Definition 3.4, $\forall\sigma\ \ (Uni_T(T)\sigma = Uni_T(\overline{T})\sigma)$ where $Uni_T(T)\sigma = \{Act_i\sigma | Cond_i\sigma = true\}$ and $Uni_T(\overline{T}) = \{\overline{Act_j}\sigma | \overline{Cond_j}\sigma = true\}$ (Definition 3.3). Since T and $\overline{T}$ are proper tables, $Uni_T(T)\sigma = \{Act_i\sigma\} \neq \{\}$ and $Uni_T(\overline{T}) = \{\overline{Act_j}\sigma\} \neq \{\}$. Hence if $\forall\sigma\ \ (Uni_T(T)\sigma = Uni_T(\overline{T}))\sigma$ then $\forall\sigma\ \ Cond_i\sigma \wedge \overline{Cond_j}\sigma \rightarrow Act_i\sigma = \overline{Act_j}\sigma$ ∎

Lemma 3.1

$$T \simeq \overline{T} \Leftrightarrow \forall i\ , j\ , \sigma\ Cond_i\sigma \wedge \overline{Cond_j}\sigma \rightarrow Act_i\sigma = \overline{Act_j}\sigma$$

This lemma follows immediately from Theorem 3.1 and from Theorem 3.2.

$y =$

$x < 0$	$x = 0$	$x > 0$
f_1	f_2	f_3

$y =$

$x \leq 0$	$x > 0$
h_1	h_2

Figure 3.2: Two Normal Tables

3.4 An Example

Consider the two tables in Figure 3.2:

$$Uni_T(T) \equiv \{(x < 0, f_1), (x = 0, f_2), (x > 0, f_3)\}$$

$$Uni_T(\overline{T}) \equiv \{(x \leq 0, h_1), (x > 0, h_2)\}$$

The set of theorems is created from $Uni_T(T)$ and $Uni_T(\overline{T})$ as described in 3.3

$$\forall\sigma\ (x < 0)\sigma \wedge (x \leq 0)\sigma \rightarrow f_1\sigma = h_1\sigma \tag{3.1}$$

$$\forall\sigma\ (x < 0)\sigma \wedge (x > 0)\sigma \rightarrow f_1\sigma = h_2\sigma \tag{3.2}$$

$$\forall\sigma\ (x = 0)\sigma \wedge (x \leq 0)\sigma \rightarrow f_2\sigma = h_1\sigma \tag{3.3}$$

$$\forall\sigma\ (x = 0)\sigma \wedge (x > 0)\sigma \rightarrow f_2\sigma = h_2\sigma \tag{3.4}$$

$$\forall\sigma\ (x > 0)\sigma \wedge (x \leq 0)\sigma \rightarrow f_3\sigma = h_1\sigma \tag{3.5}$$

$$\forall\sigma\ (x > 0)\sigma \wedge (x > 0)\sigma \rightarrow f_3\sigma = h_2\sigma \tag{3.6}$$

Left hand side of the implications in 3.2, 3.4, 3.5 are $false$ for all values of σ. Hence, these three theorems are *true* for all values of σ. The remaining theorems 3.1, 3.3, and 3.6 can be simplified to

$$(x < 0) \rightarrow f_1\sigma = h_1\sigma \tag{3.7}$$

$$(x = 0) \rightarrow f_2\sigma = h_1\sigma \tag{3.8}$$

$$(x > 0) \rightarrow f_3\sigma = h_2\sigma \tag{3.9}$$

Now we are left with proving these theorems.

3.5 Proving Theorems

To prove theorems, we apply different *proving techniques* repeatedly until: (1) all theorems are proved to be $true$, (2) a theorem is proved to be $false$, or (3) one or more theorems cannot be transformed into $true$ or $false$ values. In cases (2) and (3), the theorem that is proved to be false or the theorems that could not be proved are returned so that the user will know where the "problem spots" are.

The advantage of this method is that we can add new theorem proving techniques whenever we need them. The current implementation of the software only contains a few proving techniques. More techniques can be easily added to this system. The order of applying these techniques can be determined by heuristics or by interactive user input. The following sections discuss currently implemented proving techniques.

The theorems are proved using a computer algebra system called Maple [3] instead of theorem provers such as PVS [6]. We chose Maple to prove the theorems for several reasons. Maple contains all the functionality needed to prove these theorems. We were able to use Maple to parse tables, create theorems and check for errors.

3.5.1 Evaluating Implication

If the left hand side of the implication is $false$ or the right hand side of the implication is $true$ then, we can immediately deduce that the corresponding theorem is $true$. This technique is used to reduce the number of theorems to be proven.

3.5.2 Simplifying Theorems

Each theorem is simplified by the following process:

1. Calculate the disjunctive normal form of the left hand side of the theorem. After the simplification the left hand side will be $true$, $false$ or of the form

$$\bigvee_{j=1}^{n} \bigwedge_{i=1}^{m_j} A_{ij}$$

where A_{ij} is a literal of the form $P(x)$ or $\neg P(x)$.

2. Replace comparison operators (equal,less,greater) with *true* or $false$ whenever possible. When both operands are of type "Integer" or "Real" then we can replace the operation with *true* or $false$. For example, $2 > 3$ can be simplified to $false$ or $3 = 3$ can be simplified to true. When the operation is "=", we can simplify it to *true* or $false$ in two other cases. If both operands are the same variable or constant, then the operation can be simplified to *true* ($x = x$ is simplified to *true*). If both operands are constants and they are not the same, then it is simplified to $false$.

3.5.3 Splitting Theorems

A theorem with disjunctions is split to two or more theorems so that the theorems do not contain disjunctions. If we have a theorem of the form

$$A \vee B \rightarrow P$$

Then applying this technique creates two new theorems of the form

$$A \rightarrow P$$

and

$$B \rightarrow P$$

This technique, by itself, does not prove any theorem to *true* or $false$. But, to apply the proving techniques *Solving linear equations* and *Substituting equals*, described below, we need to make sure that all theorems contain only conjunctions. Hence, we need to apply this proving technique before we apply solving linear equations or substituting equals.

3.5.4 Solving Linear Equations

When a theorem contains only conjunctions at the right hand side of the implication, and all terms in the theorems are linear terms (terms of the form $a_1x_1 + \cdots + a_nx_n$) then we can try to *solve* for given conditions.

For example, if there is a theorem

$$(x < 0) \wedge (x > 0) \rightarrow x = y)$$

We have to solve the set $\{x < 0, x > 0\}$. This can be solved with the help of a computer algebra system such as Maple [3].

3.5.5 Substituting Equals

Whenever the right hand side of a theorem (1) contains an expression of the form $x = P$ where x is a variable and P is a term not containing x, and (2) has only conjunctions, then this expression $x = P$ can be applied to the whole theorem. For example if there is a theorem

$$(x = 2) \wedge (y = true) \rightarrow w = x + 5$$

Then by substituting $x = 2$ in the entire theorem, we get

$$(x = 2) \wedge (y = true) \rightarrow w = 7$$

Before applying this technique we split all theorems to make sure that the theorem does not have any conjunctions.

Chapter 4

Design of the Semantic Equality Checker (SEC)

The Semantical Equality Checker (SEC) tool, developed as part of this thesis, determines whether two given input tables describe the same relation(or function). SEC is implemented using Maple [3].

In this chapter, we describe the requirements, assumptions and the interface design of SEC.

4.1 Requirements

The SEC uses three input files. The first file contains type definitions and variable declarations. See appendix A for the BNF grammar of type definitions and for a sample input file. The second and third files contain tables that are to be compared for semantic equality. Appendix B contains the format of input table.

The program prints the message "Tables are semantically equal" to the standard output, if the given two tables are semantically equal. Otherwise it prints theorems that were evaluated to $false$ or the theorems that could not be proven. If there are any errors in the input file, such as type errors or syntax errors, an appropriate error message is printed.

4.2 Limitations

Currently SEC can handle only one or two dimensional tables. It cannot handle nested tables. But the tool can be extended to handle higher dimensional tables and nested tables since the algorithm works for all tables. There cannot be any quantifiers in the cells that contain conditions. When there are quantifiers or nested tables, SEC simply returns theorems that contain these quantifiers or nested tabular expressions without proving them.

4.3 Assumptions

- Input table is one of Normal,Inverted,Vector or Decision table
- Input tables have to be in th format of Appendix A and B.
- All headers of the input tables have to be boolean expressions. If any of the headers is not boolean expression, then SEC prints an error message and stops.

4.4 Module Guide

Figure 4.1 shows the module uses relation for modules of the SEC. Module A uses module B if at least one access program in A requires the correct execution of at least one access program in B.

4.4.1 Read Type (S_Read_Type)

This module reads type definitions and variable declarations from a input file. These type related informations are read from an input file called *typefile*. The format of the input type file is the secret hidden by this module.

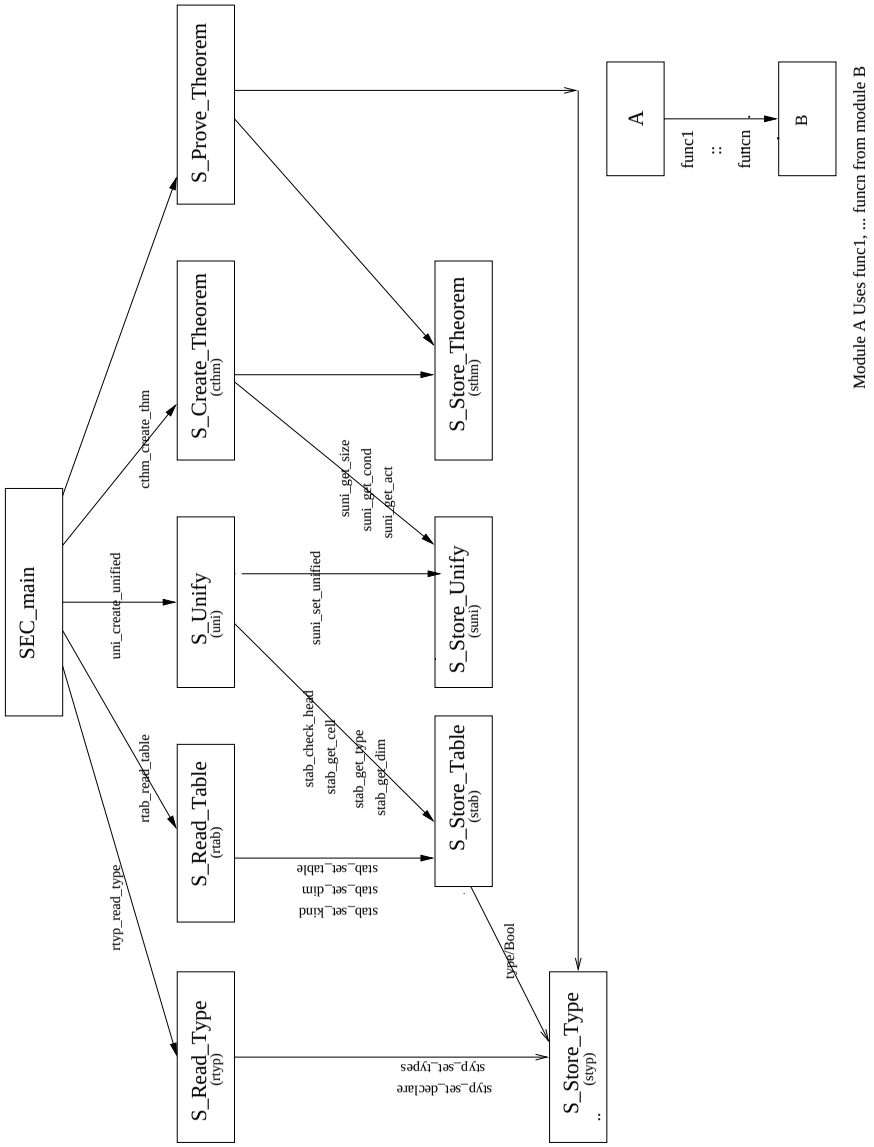

Figure 4.1: Module Design

Access Program	Description
rtyp_read_type(infile)	Reads in the type information file and calls programs in S_Store_Type to store the read information.

4.4.2 Read Table (S_Read_Table)

This module is used to read in the input tables from input files. The format of input files is the secrete.

Access Program	Description
rtab_read_table(in1,in2)	Reads the input tables in files infile1 and infile2 and calls programs in S_Store_Table to store the tables

4.4.3 Store Type Information (S_Store_Type)

This module stores the type definitions and variable declarations. It provides access programs to store type information,to check whether a given expression is of type boolean and to check a given input is constant. The data structure used to store the type information is the secret hidden by the module.

Access Program	Description
styp_set_types(tname,tdefn)	Stores type definitions
styp_set_declare(var,tname)	Stores the variable declarations
styp_get_type(var)	If the variable has a user defined type(defined in *typefile*) then returns the corresponding type. If the variable is an integer or real then returns "Integer" and "Real", respectively. Otherwise, returns the error message "Unknown type".
styp_is_const(cons)	Returns *true* if cons is member of a user defined type, an Integer or a Real.
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 of type boolean.

4.4.4 Store Tables (S_Store_Table)

This module encapsulates the semantic information of input tables. A table semantics contains the class of the table, dimension of the table, and contents of the table. Class of a table is one of *NORMAL*, *INVERTED*, *DECISION* or *VECTOR*. When headers of the tables are not boolean expressions the program prints an error message and stops execution. The secret of the module is the data structure used to store tables.

Access Program	Description
stab_set_kind(tab,class)	Stores the class of the table *tab*.
stab_set_dim(tab,dim)	Stores the dimensions of the table *tab*.
stab_set_table(tab,table)	Stores the contents of input table *tab*.
stab_check_head(tab)	Returns *true* if the headers of the table *tab* are boolean expressions, and returns false otherwise.
stab_get_cell(tab,i)	Returns the i th element of table *tab*.
stab_get_class(tab)	Returns the class of table *tab*.
stab_get_cell(tab,i,j)	Returns the contents of cell indexed by i and j in table *tab*.

4.4.5 Unify Tables (S_Unify)

This module encapsulates the transformation of input tables into the unified set. uni_create_unified creates the unified set $Uni_T(T)$ as described in Section 3.2 The secret of the modules is the algorithm to create the unified sets.

Access Program	Description
uni_create_unified	Transforms input tables into a unified set of condition-action pairs and call programs in S_Store_Unify to store the unified set.

4.4.6 Store Unified Form (S_Store_Unify)

This module is used to store the unified set(the set of condition-action pairs) of input tables. The secret of this module is the data structure used to store the unified set.

Access Program	Description
suni_set_unified(tab,uni)	Store the unified form *uni* of table *tab*
suni_get_size(tab)	Returns the size of unified set corresponding to table *tab.*
suni_get_cond(tab,i)	Returns the *i*th condition of table *tab.*
suni_get_act(tab,i)	Returns the *i*th action of table *tab.*

4.4.7 Create Theorems (S_Create_Theorem)

This module creates theorems using the unified sets of condition-action pairs as described in Section 3.3. The algorithm used to create theorems is the secret of the module.

Access Program	Description
cthm_create_thm	Create theorems using the unified sets.

4.4.8 Store Theorems (S_Store_Theorem)

This module is used to store the theorems created in previous module. The data structure used is the secret of the module.

Access Program	Description
sthm_set_theorem(thm)	Stores the theorem *thm.*
sthm_get_thm(i)	Returns the i th theorem

4.4.9 Prove Theorems (S_Prove_Theorem)

This module contains sub modules for different proving techniques. Secret of each sub module is the proving technique. Currently there are five sub-modules in this module. More modules can be added whenever a new proving technique is implemented.

Simplify Theorems (SPT_Simplify_Theorem)

Contains access program, pt_simplify_theorem to simplify theorem by 1) calculating the minimal sum of product of the left hand side of the implication, 2) replacing expressions with comparison operations(equal,greater, or less) with *true* or $false$ whenever possible as explained in Section 3.5.2.

Evaluate Implication (SPT_Eval_Implication)

Contains access program, pt_evaluate_implication to reduce each theorem to *true* or $false$ if possible as explained in Section 3.5.1. If a theorem is evaluated to false then it is printed for diagnostic purposes and the execution of the program stops because the tables are semantically not equal.

Split Theorem (SPT_Split_Theorem)

Contains access program, pt_split_theorem to split a theorem into one or more theorems when it contains disjunctions as explained in Section 3.5.3.

Solve Linear Equation (SPT_Solve)

Contains access program, pt_solve to solve using the *solve* function in maple as explained in Section 3.5.4.

Substitute Equal (SPT_Subs_Equal)

Contains access program, pt_substitute_equal to substitute as explained in Section 3.5.5.

Chapter 5

Conclusions

This chapter discusses the results of this thesis work, some limitations of the SEC, and possible future work on this area.

5.1 Results

We have developed an algorithm to compare tables to determine whether they describe the same function. The algorithm works for all tables described in [1].

We have also developed a prototype tool called SEC which implements this algorithm. SEC has some limitations. Section 4.2 discusses the limitations of SEC. SEC has been tested with many different input tables. See Appendix C for the results of some tests.

5.2 Future Work

Currently SEC accepts input tables in a specific format. We will need an interface to incorporate SEC into Table Tool System (TTS). TTS is a software system, developed by past and present members of SERG, that supports the use of tables in software documentation. SEC was developed independently from TTS because TTS currently does not store type related information needed for the algorithm. Work is underway to store type information in TTS. When this work has been completed, SEC can be incorporated into TTS.

In the current implementation, conditions cannot contain quantifiers. Quantifiers over finite interval could be handled easily. Some research might be needed to develop proving techniques to handle quantifiers over infinite intervals.

In current implementation, the tables that are compared for semantic equality should have the same variables. SEC should be modified to include LAMBDA abstractions.

More proving techniques can be added as needed to SEC. This is done by adding sub modules to the module *S_Prove_Theorem*.

5.3 Conclusion

This thesis work presents an algorithm for verifying software specifications.

We have developed an algorithm to determine semantic equality and proved that the algorithm works. SEC, the tool to check semantic equality of tables has been tested with many different kinds of tables. See Appendix C for the results of some tests.

Appendix A

Format of Type Definitions

A.1 EBNF Grammar for Type Definitions

Syntax	$\longrightarrow$	{TypeDef} {VarDef}
TypeDef	$\longrightarrow$	"TYPE:" TypeName {WhiteSpace} SetDef
SetDef	$\longrightarrow$	"Set" "{" SetEle {","SetEle} "}"
SetEle	$\longrightarrow$	String
TypeName	$\longrightarrow$	String
VarDef	$\longrightarrow$	VarName ":" Type
Type	$\longrightarrow$	TypeName | "Integer" | "Char" | "Real" | "Bool"

A.2 Example of a Type Definition File

```
TYPE ModeT : Set {'high','low','normal'}
TYPE Mode2 : Set {A,B}
valve : ModeT
pressure : Real
mode : ModeT
a : Bool
c : Integer
lower : Real
upper : Real
x : Integer
```

Appendix B

Format of Table Definitions

B.1 Format

```
DIM:=[<N1>,<N2>];
T_TYPE:=<kind>;
cell[i,j]:= <expr>;
...
...
```

`NUM1` and `NUM2` are positive integers.
`kind` is one of `NORMAL`, `INVERTED`, `VECTOR`, or `DECISION`.
`i` and `j` are positive integers such that $0 \leq i < N1$ and $0 \leq j < N2$
`expr` is any expression defined using variables in *typefile*.

B.2 Example of Two Input Table Files

```
DIM:=[3,1];
T_TYPE:=NORMAL;
cell[1,0]:= Greater(x,0);
```

```
cell[2,0]:= Equal(x,0);
cell[3,0]:= Less(x,0);
cell[0,1]:= true;
cell[1,1]:= x ;
cell[2,1]:=  -x;
cell[3,1]:=  -x;
DIM:=[4,1];
```

```
DIM:=[2,1];
T_TYPE:=NORMAL;
cell[1,0]:= GreatEqual(x,0);
cell[2,0]:= Less(x,0);
cell[0,1]:= true;
cell[1,1]:=  x;
cell[2,1]:= -x ;
```

Appendix C

Sample Output of Semantic Equality Checker

```
knadaraj@noether:tool> maple
    |\^/|     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.
> read(my);
[bequal, bsimp, canon, convert/MOD2, convert/frominert, convert/toinert,

    distrib, dual, environ, randbool, satisfy, tautology]

[THS(And(And(Greater(x, 0), true), And(GreatEqual(x, 0), true)), Equal(x, x)),

    THS(And(And(Greater(x, 0), true), And(Less(x, 0), true)), Equal(x, -x)),

    THS(And(And(Equal(x, 0), true), And(GreatEqual(x, 0), true)), Equal(-x, x))

    , THS(And(And(Equal(x, 0), true), And(Less(x, 0), true)), Equal(-x, -x)),

    THS(And(And(Less(x, 0), true), And(GreatEqual(x, 0), true)), Equal(-x, x)),

    THS(And(And(Less(x, 0), true), And(Less(x, 0), true)), Equal(-x, -x))]

>
> pt_simplify_theorem();
[THS(Greater(x, 0), true), THS(Greater(x, 0) &and Less(x, 0), Equal(x, -x)),

    THS(Equal(x, 0), Equal(-x, x)), THS(Equal(x, 0) &and Less(x, 0), true), THS(

    (Greater(x, 0) &and Less(x, 0)) &or (Equal(x, 0) &and Less(x, 0)),

    Equal(-x, x)), THS(Less(x, 0), true)]

> pt_evaluate_implication();
               THS(Greater(x, 0) &and Less(x, 0), Equal(x, -x))

                       THS(Equal(x, 0), Equal(-x, x))

THS((Greater(x, 0) &and Less(x, 0)) &or (Equal(x, 0) &and Less(x, 0)),

    Equal(-x, x))
```

```
> pt_split_theorem();
[THS(Greater(x, 0) &and Less(x, 0), Equal(x, -x)),

    THS(Equal(x, 0), Equal(-x, x)),

    THS(Greater(x, 0) &and Less(x, 0), Equal(-x, x)),

    THS(Equal(x, 0) &and Less(x, 0), Equal(-x, x))]

> pt_solve();
bytes used=1077340, alloc=851812, time=0.19
[THS(false, Equal(x, -x)), THS(&and(Equal(x, 0)), Equal(-x, x)),

    THS(false, Equal(-x, x)), THS(false, Equal(-x, x))]

> pt_evaluate_implication();
                     THS(&and(Equal(x, 0)), Equal(-x, x))

                                    false

> pt_substitute_equal();
                    [THS(&and(Equal(0, 0)), Equal(0, 0))]
> pt_simplify_theorem();
                                [THS(true, true)]
> pt_evaluate_implication();
                                     true
```

Bibliography

[1] R. F. Abraham, "Evaluating generalized tabular expressions in software documentation," M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, Feb. 1997. Also published as CRL Report 346.

[2] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore, "Software requirements for the A-7E aircraft," Tech. Rep. NRL/FR/5546-92-9194, Naval Research Lab., Washington DC, 1992.

[3] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Mongan, and S. M. Watt, *Maple V Language Reference Manual*. Springer-Verlag, 1992.

[4] D. N. Hoover and Z. Chen, "Tablewise, a decision table tool," in *Proc. Conf. Computer Assurance (COMPASS)*, (Gaithersburg, MD), pp. 97–108, June 1995.

[5] R. Janicki, "Towards a formal semantics of parnas tables," in *Proc. Int'l Conf. Software Eng. (ICSE)*, pp. 231–240, Apr. 1995.

[6] S. Owre, J. Rushby, and N. Shankar, "Analyzing tabular and state-transition specifications in PVS," Tech. Rep. SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Revised May 1996.

[7] D. L. Parnas, G. J. K. Asmis, and J. Madey, "Assessment of safety-critical software in nuclear power plants," *Nuclear Safety*, vol. 32, no. 2, pp. 189–198, April-June 1991.

[8] D. L. Parnas, "Tabular representation of relations," CRL Report 260, Communications Research Laboratory, Nov. 1992.

[9] D. L. Parnas, "Predicate logic for software engineering," *IEEE Trans. Software Engineering*, vol. 19, no. 9, pp. 856–862, Sept. 1993.

[10] D. L. Parnas and J. Madey, "Functional documentation for computer systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, Oct. 1995.

[11] J. E. Rubin, *Mathematical Logic: Applications and Theory.* Saunders College Publishing, 1990.

[12] V. Sperschneider and G. Antoniou, *Logic: A Foundation for Computer Science.* Addison-Wesley, 1991.

[13] M. Viola, "Ontario hydro's experience with nre methods for engineering safty-critical software,"

[14] M. von Mohrenschildt, "Algebra of normal function tables," CRL Report 350, Communications Research Laboratory, May 1997.

[15] M. von Mohrenschildt, "A normal form for rings of piecewise functions," *Journal of Symbolic Computation*, vol. 26, pp. 607–619, 1998.