SPECIALIZATION: AN APPROACH TO SIMPLIFYING TABLES IN SOFTWARE DOCUMENTATION

By

PREETI RASTOGI, M. ENG.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Engineering

McMaster University
© Copyright by Preeti Rastogi, February 1998

MASTER OF ENGINEERING (1998) McMASTER UNIVERSITY
(Electrical and Computer) Hamilton, Ontario, Canada

TITLE:	Specialization: An Approach to Simplifying Tables in Software Documentation
AUTHOR:	Preeti Rastogi
	M.Eng. (Birla Institute of Technology, Ranchi, India)
SUPERVISORS:	Dr. David Lorge Parnas and
	Dr. Martin von Mohrenschildt
NUMBER OF PAGES:	xiii, 90

Abstract

Precise mathematical documentation plays a crucial role in safety critical, mission critical, and finance critical software where the cost of failure is very high either in terms of money or human life. Good documentation is an important component of software and is absolutely necessary for maintenance. Tabular notation advocated by Parnas et al. has been found to be more readable and easy to understand when compared to conventional mathematical representations.

Tables can sometimes be complex and difficult to comprehend. In such cases “specialization” can simplify them. Some tables can be simplified without loss of generality. For others specialization may be used. *Specialization* is a technique that reduces the domain for which the expression is valid. For tabular expressions, specialization may allow the removal of rows and/or columns when the domain of predicate (condition) sub-expressions is outside the domain under consideration. User defined constraints narrow the domain under consideration. Specialization may result in several tables depending on the constraints, but each table is usually simpler than the original.

This work involves the design and development of a prototype tool to help in understanding specialization and simplification. This tool helps to test intermediate results involving partial evaluation of the mathematical expression.

Acknowledgments

I wish to express my sincere appreciation for my supervisor, Dr. David L. Parnas, for his guidance, support, inspiration and faith throughout this research work. His ideas and thoughts helped in formalizing the basis of this research, and also his constant reminder of the final product kept me on track. I would like to thank my co-supervisor, Dr. Martin von Mohrenschildt, for his encouragement, enthusiasm and invaluable suggestions that made it possible for me finish this work. I am also greatly indebted to Dr. Mohrenschildt for his insight and ideas on Computer Algebra Systems. I am also thankful to Dr. Jeffery I. Zucker for early discussions we had on the concepts this work and for his careful reading of the thesis.

I want to express my appreciation to all the members of Software Engineering Research Group, especially Dennis Peters and Ruth Abraham for their and helpful suggestions. I wish to thank Ruth also for proof reading and providing helpful comments. I would also like to thank Al Tyson for his supporting modules that were integrated in this work.

I am indebted to my husband, Mayank and son, Neelabh whose support made it possible for me to finish this work. I am also grateful for the support, and motivation of my parents and my mother-in-law.

Finally, I would also like to thank the Telecommunications Research Institute of Ontario (TRIO) , Natural Sciences and Engineering Research Council (NSERC) and Bell Canada for their financial support.

Table of Contents

List of Figures

List of Tables

List of Acronyms with Description

CAS	Computer Algebra System Used for symbolic computation
CCG	Cell Connection Graph Used in GTS to characterize the flow of information in a tabular expression
GTS	Generalized Table Semantics Used for table semantics
SAST	Specialization And Simplification Tool Tool for specialization and simplification
SERG	Software Engineering Research Group A Research Group at Communication Research Laboratory, at McMaster University, Hamilton, Ontario, Canada
TCT	Table Construction Tool A Motif based graphical user interface to input expressions
TH	Table Holder Kernel of TTS to store expression syntax in a tree form
TIF	Table Integration Framework Helps in integration of tools and applications to TTS
TPTool	Table Printing Tool A graphical interface to display expressions
TTS	Table Tool System Set of tools being build by SERG to facilitate manipulation of tabular expressions in software documentation

Chapter 1

1. Introduction

1.1 Background

Software now controls complex applications that are safety critical, mission critical, and finance critical. Aircraft, spacecraft, cars, nuclear power generating plants, telephone exchanges, and stock exchanges are just some of the applications that use software. Failures in these critical applications may result in heavy loss of human life and/or money. Such pervasive use and growing dependence on software products requires that it be reliable and accurate.

Software maintenance, where documentation plays an essential role, accounts for about 60-80 percent of the total cost of software and is frequently done by people other than developers or designers of the software. Good documentation plays a crucial role in the development of software, from requirements analysis, design, design reviews, implementation, validation and verification, and testing, through to maintenance. A software design document should clearly state the software's purpose and enhance the understanding of the software product. Documentation of software is often done imprecisely and inadequately using natural language. Moreover, the lack of supporting tools and techniques makes generation and maintenance of documentation difficult, thus affecting the quality of software.

In the Software Engineering Research Group (SERG) at McMaster University, Hamilton, Canada, precise documentation is recognized as a crucial component of the software and an integrated computer-aided software engineering (CASE) tool set is being developed to support SERG's approach to software documentation. While advocating design through documentation [9,14] suggests a rational design process and ways to even fake such a process by producing documents that are

accurate, rational and usable. The relational model of software documentation studied in SERG uses mathematical relations to represent the intended behaviour of computer systems and software applications, as described in [18]. Conventional mathematical representations denoting these relations are often complex and easily misunderstood. To overcome these problems a class of multi-dimensional expressions, called tables, has been developed. Various industrial experiences with the relational documentation technique using tables in an *ad hoc* manner are mentioned in [11]. For example: the requirements document for the Onboard Flight Program for U.S. Navy's carrier based attack aircraft, the A-7E; and documents for the inspection of safety critical programs for Darlington Nuclear Power Generating Station in Ontario, Canada, illustrate improvement in software quality resulting from reduced ambiguities while precisely documenting the intended behaviour of the software.

Tables are multi-dimensional expressions comprising sets of cells organized as one or more headers and a main grid, as described in [15]. Following are some of the benefits of tables:

- Improved understanding - Tables clearly identify various conditions required to completely specify the behaviour of software. Designers or document authors or reviewers can deal with each case separately without any confusion.
- Ease of inspection - The format of tables allows segregation of cases pertaining to an application, and thus the detection of missing cases is very easy. Tables are easy to check to see if all cases are covered and hence its inspection enhances confidence in the program that they specify.
- Reduced repetition of sub-expressions.
- Clear and accurate description of the discontinuities in the domain of the relation. Tables can also conveniently represent relations consisting of various distinct types of elements in their domain and range.

Example in FIGURE 1 and FIGURE 2 (adapted from [26]) illustrates the benefits of tabular notation over the traditional one-dimensional way of writing expressions. FIGURE 1 shows a conventional representation of a function such that f_irr_band has a value of low_irrational or rational or high_irrational based on values of s_irr_band and m_sig. FIGURE 2 shows the tabular representation of the function in FIGURE 1 and is informally read as follows:

Based on the given values of the variables, select the row from H_1 and the cell from main grid G in the same row as H_1 such that the cell expressions evaluate

to true under given values, then the value of f_irr_band is given by the corresponding cell in H_2.

$$
\text{f\_irr\_band} = \begin{cases}
\text{low\_irrational} & \text{if} \quad ((\text{s\_irr\_band} = \text{high\_irrational}) \wedge (\text{m\_sig} \le 100)) \\
\text{low\_irrational} & \text{if} \quad ((\text{s\_irr\_band} = \text{low\_irrational}) \wedge (\text{m\_sig} < (100 + \text{db}))) \\
\text{low\_irrational} & \text{if} \quad ((\text{s\_irr\_band} = \text{rational}) \wedge (\text{m\_sig} \le 100)) \\
\text{rational} & \text{if} \quad ((\text{s\_irr\_band} = \text{high\_irrational}) \wedge ((100 < \text{m\_sig}) \wedge (\text{m\_sig} \le (4900 - \text{db})))) \\
\text{rational} & \text{if} \quad ((\text{s\_irr\_band} = \text{low\_irrational}) \wedge (((100 + \text{db}) \le \text{m\_sig}) \wedge (\text{m\_sig} < 4900))) \\
\text{rational} & \text{if} \quad ((\text{s\_irr\_band} = \text{rational}) \wedge ((100 < \text{m\_sig}) \wedge (\text{m\_sig} < 4900))) \\
\text{high\_irrational} & \text{if} \quad ((\text{s\_irr\_band} = \text{high\_irrational}) \wedge ((4900 - \text{db}) \le \text{m\_sig})) \\
\text{high\_irrational} & \text{if} \quad ((\text{s\_irr\_band} = \text{low\_irrational}) \wedge (4900 \le \text{m\_sig})) \\
\text{high\_irrational} & \text{if} \quad ((\text{s\_irr\_band} = \text{rational}) \wedge (4900 \le \text{m\_sig}))
\end{cases}
$$

FIGURE 1 - A Traditional Function Representation

f_irr_band = **H_2**

p_T: $H_1 \wedge G$ **r_T**: H_2	low_irrational	rational	high_irrational
s_irr_band = high_irrational	m_sig ≤ 100	100 < m_sig ∧ m_sig ≤ 4900 - db	4900 - db ≤ m_sig
s_irr_band = low_irrational	m_sig < 100 + db	100 + db ≤ m_sig ∧ m_sig < 4900	4900 ≤ m_sig
s_irr_band = rational	m_sig ≤ 100	100 < m_sig ∧ m_sig < 4900	4900 ≤ m_sig

H_1 **G**

FIGURE 2 - Inverted Function Table

Tabular notation, may include set theoretic expressions and first-order predicate logic expressions, is practical and acceptable for industrial applications because it provides a precise method of communication. Ten classes of tables as

described in [15] were found useful during industrial experiences. A more general approach, as described in [10], allows not only ten common classes of tables but also many new classes not dealt with earlier. [1,10] also deals with semantics of tables in a more general way using a cell-connection graph to characterize information flow in tables. Using this general model of tables, [1] shows how tables used by various other groups like Ontario Hydro, Canada and Atomic Energy of Canada Limited (AECL); the Software Cost Reduction (SCR) project at the Naval Research Laboratory, Washington; the Tablewise tool by Odyssey Research Associates Inc., Ithaca; and the Requirements State Machine Language (RSML) developed in University of California at Irvine; can be defined using the rules in [1] even though their format is dissimilar.

1.2 Introduction to Simplification and Specialization

Generally, *simplification* results in an equivalent expression with fewer symbols than the original expression computed according to a set of transformation rules. Example: x + 0 can be simplified to x. *Specialization* in this thesis means reducing the domain for which the expression need to be valid.

Consider the example shown in FIGURE 3, which is the original table for FIGURE 4. FIGURE 4 is equivalent to FIGURE 3 under the constraint $y = 2 \wedge w = 6$. The constraint (w = 6) allows removal of the first column in header H_2 as this is not applicable for the given values of variable y. The constraint assigns 2 to the variable y and 6 to the variable w, everywhere in the tabular expression.

H_2

p_T: $H_1 \wedge H_2$ **r_T**: G	$w < 0$	$w \geq 0$
$x = 3$	$y = 5$	$\sqrt{xy} = \sqrt{w}$
$x < 3$	$y = x + w - 2w$	$(w > 6) \wedge (w \leq 6)$
$x > 3$	$(y = 2) \vee (y \neq 2)$	$(w = 0) \vee (y < x)$

H_1 **G**

FIGURE 3 - Original Tabular Expression

Constraint: y = 2 ∧ w = 6

H_2

p_T: $H_1 \wedge H_2$ **r_T**: G	true
x = 3	true
x < 3	false
x > 3	true

H_1 **G**

FIGURE 4 - Specialized Tabular Expression for FIGURE 3

As an example consider the cell expression y = 5 from the first row and first column of the main grid G, which can be simplified under this constraint to `false`. The final form of the specialized table is given in FIGURE 4.

Test(e, V, index, found, low, high, med) =

(‘low ≤ med ≤ ‘high) ⇒ **H_2**

p_T: H_2 **r_T**: H_1 = G	V[med] < e	V[med] = e	V[med] > e
index’|	*true*	index’ = med	*true*
found’|	found’ = ‘found	found’ = true	found’ = ‘found
low’|	low’ = med + 1	low’ = ‘low	low’ = ‘low
high’|	high’ = ‘high	high’ = ‘high	high’ = med - 1

H_1 **G**

∧ NC(e, V, med)

FIGURE 5 - Original Tabular Expression

Test(e, V, index, found, low, high, med) =

(‘low ≤ med ≤ ‘high) ⇒ **H2**

p_T: H_2_ **r_T**: H_1 = G	*true*
index’|	index’ = med
found’|	found’ = true
low’|	low’ = ‘low
high’|	high’ = ‘high

H_1 **G**

∧ NC(e, V, med)

FIGURE 6 - Specialized Table for FIGURE 5 for V[med] = e

The example in FIGURE 5 represents part of a binary search program. Refer to [17] for details. It compares the value of middle element in the array `V` with the element to be searched `e` and then sets the variables `low`, `high` and `index` variables accordingly. The variable `found’` is set to boolean value `true` if the element `e = V[med]`. Under the constraint V[med] = e, FIGURE 5 specializes to FIGURE 6.

1.3 Motivation for Simplification and Specialization of Tables

Tabular notation can be precise, readable, mathematically correct, formally defined and easily applicable. It is often a great improvement over conventional notations but tabular expressions can still be complex and difficult to understand and analyze for complicated applications involving several conditions. Transformation of a table from one class to another may also result in a complex table. As the applications grow in size and complexity, the very aspect of simplicity and intuitiveness which makes tables popular and useful can be lost.

To ameliorate this problem, a tool for restricting the domain under consideration and simplifying expressions based on user-defined constraints has been built. For tabular expressions, restriction of the domain can result in removal of rows or columns. This removal of cases that are not in the given domain and the simplification of all the sub-expressions in a table leads to a simpler table which is easy to understand. The purpose of the Specialization and Simplification Tool (SAST) developed in this work is to automate this simplification process.

SAST can assist the user in applying the "divide and conquer" approach to understanding complex expressions. This also help saves document inspectors or design reviewers from the monotonous and error prone task of tracking the inspection or analysis procedure. This tool also allows the user to apply division of labour by allowing several people to work in parallel on different cases and the reliability of each case will contribute towards the reliability of the whole software product.

1.4 Structure of SAST

SAST has three major parts that perform functions as follows:

- Input/Output of expressions
- Interface with a math engine
- Computations in the math engine

Refer to Chapter 3 and Chapter 4 for details on first two parts of SAST. The math engine is responsible for all the symbolic computations needed in SAST. A Computer Algebra System (CAS) is used as a math engine in SAST. Details of CAS are given in the next section.

1.5 Overview of Computer Algebra Systems

This section discusses symbolic computation, need for a Computer Algebra Systems and its types, and selection of a math engine for SAST. It also describes how the math engine is hidden in SAST and some issues involved in using CAS.

1.5.1 Symbolic Computation

Symbolic computation is an area of mathematics which is centuries old and treats symbols as mathematical objects. In this area of computation, integers,

rational, real and complex numbers, and systems of equations are all treated as mathematical objects and are represented as symbols. Thus it results in exact mathematical solutions using symbols, instead of numbers with machine specific or user defined precision. This avoids a cumulative effect of round-off errors and is better suited for accurate results. Computer Algebra Systems support both numerical and symbolic computations. After symbolic calculations, the numeric values can be assigned and numeric calculations can be performed.

1.5.2 Need for a Computer Algebra System

A comparison of results obtained from a CAS and a numerical software is given here. These examples have been worked on Maple, a CAS and Matlab (version 4.2c), mainly a numerical software package (without using Matlab's symbolic math toolbox). These examples illustrate the ability of CAS to handle symbols as mathematical objects and accuracy of their results.

Matlab is an interactive software for high performance numeric computation, matrix manipulation and visualization. It also has many application specific solutions called *toolbox* for signal processing, control theory, symbolic math based on Maple, neural networks etc.

A description of Maple is given in section 1.5.3.

Example 1
Simplification: x + x + x*x
CAS: $2x + x^2$
Matlab: ??? Undefined function or variable x
(Matlab will not be able to simplify)

Example 2
Rational Arithmetic: 856/56
CAS: 107/7
Matlab: 15.2857

Example 3
Solve: $4x^2 - 8x + 1 = 0$
CAS: $1 + \frac{1}{2}\sqrt{3}$ and $1 - \frac{1}{2}\sqrt{3}$
Matlab: 1.8660 and 0.1340

1.5.3 Computer Algebra Systems

Computer Algebra Systems with a combination of mathematics and computer technology serve as broadly applicable tool for expression manipulation and computation. Mathematicians, engineers, scientists and all those who need algebraic techniques for analysis can utilize these CAS. They are more useful when the size of an expression or the length of computation grows beyond the capabilities of the human brain in terms of time requirements and probability of errors. In Computer Algebra Systems, the computations are carried out according to the rules of algebra rather than the approximate floating point arithmetic of computers. They deal with the problem in a manner closer to human thinking, and allow effective reuse of calculations and results. They only substitute the value of the variable at the end of the symbolic calculation.

1.5.3.1 Types of CAS

1.5.3.1.1 Special Purpose CAS

There are two types of CAS based on their application scope: special and general. Special computer algebra systems are those that are limited to one application area. These special purpose systems are much more efficient in terms of memory and execution speed as they have optimized algorithms to deal with a specific problem. Some special purpose systems are : Schoonschip used in high energy physics, Camal for celestial mechanics, Sheep and Stensor for general relativity, Cayley and Gap for group theory, and Pari for number theory.

1.5.3.1.2 General Purpose CAS

The general purpose systems can handle a wide range of problems in many application areas. They are capable of handling both numeric and symbolic computations. These general purpose systems have built-in algorithms and thus take the burden of inherent mathematical knowledge from the users. These have simple user commands, excellent graphics and capabilities that can be extended by the user. Some of these systems are: Maple, Mathematica, Macsyma, Reduce, ScratchPad-II, Axiom and Derive.

For further discussion on CAS refer to [2, 3, 4, 5, 8, 27].

MACSYMA

Macsyma is one of the oldest computer algebra systems still in use. It was developed by Massachusetts Institute of Technology in 1968 as a part of a project lead by Carl Engleman, Bill Martin and Joel Moses. It involved a substantial amount (50-100 man-years) of programming effort. The objective of this project was to have a system capable of dealing with all algebraic problems. It was written

in LISP. Refer to [3] for more details.

REDUCE

Reduce was developed and designed by Professor A.C. Hearn who was originally at Stanford University and then at University of Utah. It is widely used because of its portability and less memory requirement. It was built on top of a LISP dialect called 'Standard LISP'. Reduce is an open software system allowing its users to modify its source code. Reduce has kept its software updated according to the latest research. Refer to [2] for further details.

SCRATCHPAD-II

Scratchpad was developed in the eighties in IBM's Thomas J. Watson Research Laboratory for its mainframe, UNIX workstation, PS/2 range. Scratchpad-II can define and handle different types of algebraic structures. In 1992, it was released under the name Axiom on IBM RS/6000 machines and later on UNIX platform also. It is also implemented in LISP. Refer to [3] for details.

DERIVE

Derive and its precursor MuMath, were condensed from Macsyma around 1980 to run on the TRS-80 Radio Shack computer. It can now run on small computers (8 bit, 64K RAM for MuMath) and Intel 80x86 processors. MuMath was developed by Stoutemyer at University of Hawaii. Derive offers a good menu-driven interface with graphical and numerical abilities. It has good user-ready facilities but limited programming capabilities.

MAPLE

Maple is a software product marketed by Waterloo Maple founded in 1988 by Keith Geddes and Gaston Gonnet. It is also an ongoing project of the Symbolic Computation Group at the University of Waterloo in Ontario, Canada. The development of Maple started in 1980 and since then it has evolved significantly. Maple is comprised of three parts as described in FIGURE 7 adapted from [8].

The Maple System	
Part	**Function**
IRIS	Parser Display of expressions Graphics Worksheet type of user interface for Motif (X Windows), MS Windows, NeXT
Kernel	Interpreter Memory management Basic and time critical procedures for computations in rational and polynomial arithmetic
Library	Library functions Application packages On-line help

FIGURE 7 - Parts of Maple

Following are some features of the Maple as described in detail in [4, 5, 8].

- Maple has a compact design as its kernel handles only basic algebraic operations and all other operations are either added by Maple automatically or explicitly by the user as needed. Its functionality can be extended by the addition of user defined procedures. Moreover, portability of Maple to different platforms and utilizing the architecture of the systems makes it a popular and faster option. The IRIS and kernel parts are written in C and the library routines are written in Maple programming language.
- Only one copy of any expression or sub-expression is saved in a session and all the sub-expressions are re-used.
- Maple has an open architecture concept with facilities that allow users to view the various algorithms used by the commands. It has a relatively good on-line help with books to cover the concepts and design decisions. Also there is a very active ‘Maple User Group‘ (MUG) moderated by the Symbolic Computation Group that is very helpful for discussions, suggestions and hints.
- Maple has a logic package that handles logical equivalence of two expressions, simplification of boolean expressions, canonical form of a boolean expression etc. which may be useful for further extensions of this tool.

- Since Maple is relatively old, some concepts like pattern matching and transformation rules are not very well developed. Though Maple has been in use for a long time, it is not entirely error free. Refer to [8] and section 1.5.6 for specific problem areas and examples.

MATHEMATICA

Another Computer Algebra System called Mathematica was also considered for use as the math engine of SAST. First developed in 1988, Mathematica is a software product from Wolfram Research, Inc., Illinois, U.S.A. All the information in this section is based on Mathematica version 2.2 for X windows system 1993 and [12]. Mathematica is also a symbolic and numerical calculator with its own programming language.

Following are some features of Mathematica including advantages and disadvantages.

- Mathematica is relatively new; many new ideas and algorithms have been implemented. It has two parts: the kernel and the front end. The kernel is the heart of Mathematica handling all the calculations and is same for all computers. The front end is the graphical user interface and it depends on the type of system.
- The functionality of Mathematica can be extended by using one of its styles of programming: functional, procedural or rule based. Mathematica has procedural programming (using conditionals and iterations) and rule based programming with pattern matching and transformation rules for changing one form of expression to another. Functional programming helps in focusing more on the concept rather than on programming by providing functions that can iterate over a structure (lists, matrix, ...).
- Mathematica handles formulae, lists, and graphical objects as expressions with a tag to specify the type of structure of the expression in conformance to its “group of data” representation. Example: {a, b, c} is stored internally as List[a, b, c] and x + y is stored as Plus[x,y].
- Mathematica has built-in programs that allow functions to be applied:

- - repeatedly to an expression
- Example 1:

```
In[1] := FoldList[Plus, 0, {a, b, c}]

Out[1] := {0, a, a + b, a + b + c}
```

- - or to expression or lists
- Example 2:

```
In[2] := Apply[f,{{a, b, c},{b, c,
d}},{0, 1}]

Out[2] := f[f[a, b, c], f[b, c, d]]
```

- - or to part of an expression
- Example 3:

```
In[3] := MapAt[f, {a, b, c},{{2},
{3}}]

Out[3] := {a, f[b], f[c], d}
```

Note: In and Out indicate input and output statements in Mathematica.

- Mathematica can also work with mixed relational operations.
  - For example: The expression 3 < 5 <= 6 is simplified to *true*.
- Mathematica does not have a well-developed logic package and needs more memory than Maple because most of the facilities in Mathematica have to be loaded as a whole system rather than just the kernel.

AXIOM

Another CAS called Scratchpad was developed by the computer algebra group headed by R. Jenks at the IBM Thomas J. Watson Research Centre in Yorktown Heights, New York in 1978. It was released in 1991 under the name Axiom and is now in continuous development at IBM T. J. Watson Research Centre, at the Numerical Algorithms Group Ltd. (who markets the product) and Axiom Developers Association.

The following describes some features of Axiom and is adapted from the

information available publicly on the World Wide Web.

- Axiom is a very consistent and robust computer algebra system with strong typing. Some type inference techniques are also built in to minimize the need for type declarations. It also allows the creation of new user defined functions.
- It has a unique object oriented hierarchy for datatypes. Axiom is implemented in Lisp.
- The new library compiler supports both object oriented and functional programming styles and produces stand-alone executable programs, portable byte library codes and C or Lisp source code.
- The library source code of Axiom is available to its users.

1.5.4 Selection of Math Engine

For this research work, Maple, Axiom and Mathematica were considered as alternatives for the math engine. Maple was finally chosen as the math engine for SAST for the following reasons:

- Maple has a logic package that is not available in Mathematica. Also, Mathematica needs more memory to store its kernel than Maple.
- Axiom has a very strong type system. However, as the type system is not yet implemented in Table Tool System (TTS) of which SAST is a part, Axiom could not be used.
- Good support for Maple was available (both internally and externally).

1.5.5 “Hiding” the Math Engine in SAST

SAST hides the computations performed by the math engine as procedures in a module. These procedures are described in detail in section 3.3.2.14 and section 4.7.

The interface between SAST and Maple has been designed in a way that allows changing to a different CAS without much difficulty. The expressions from TTS are sent to Maple in a prefixed form (for reasons described in Chapter 4). This provides a uniform interface to any computer algebra system. The interface is discussed in detail in Chapter 3 and the access programs are described in section 3.3.2.2.

1.5.6 Issues involved in using Maple

Maple has good mathematical capabilities but is not perfect because of bugs, and some need for user judgment. Some of the issues involved in using Maple are as follows:

1. Automatic simplification applies the transformation rules based on the type of mathematical functions used in the expression. This may sometimes lead to undesired results. These simplifications are not valid in general and there is a tradeoff between usability and efficiency.

   Example 1: A logical transformation that simplifies `x and x` to `x`. This transformation will take place regardless of the fact that `x` is not a boolean expression. Another similar transformation like `expn and false` → `false` will take place irrespective of the fact that `expn` is not a boolean expression.

   Example 2: The expression `a = b` would evaluate to `false` because Maple tests the equality of expressions using addresses of where these expressions are stored in memory. This test of equality by single address comparison is due to the fact that Maple maintains unique instance of an expression in memory. Following expression would evaluate to `false` even when variables `a, b, c` and `d` have not been assigned any values. In SAST, there is a Maple procedure called `Equal` (described in section 4.7) to avoid such simplifications such as the one illustrated below.

```
> a = b and c > d;

                        false
```

   Example 3: A transformation that simplifies `0 * x` to `0` will happen even if x is undefined or ∞.

   Example 4: Another transformation for automatic simplification that transforms 0^k to `0` (for all `k`) will be done even if k = 0. This is illustrated in the following example.

```
> sum(a[k] * x^k,k = 0..n);
```

$$\sum_{k=0}^{n} a[k]\,x^{k}$$

```
> eval(subs(x = 0, ”)); (” refers to previous
expression)

                    0
```

2. CAS allows both numeric and symbolic computation with user defined precision. These systems use software to handle floating point arithmetic which is significantly slower than one built in hardware. The programming languages use the built-in hardware to handle floating point calculations.

3. Computer Algebra Systems performs exact arithmetic that needs more memory space and time. These are often plagued by huge intermediate expressions in their calculations known as *intermediate expression swell*. Swelling of expressions can be avoided by using good algorithms and optimization techniques.

4. Other issues of defining the domain in which the user desires to perform their computations or defining user desired mathematical structures, are not well addressed by most of CAS. Systems like Scratchpad-II/Axiom along with Maple have started to develop packages in this area.

5. There is a steep learning curve in using these systems to avoid pitfalls in symbolic computations. The user should be familiar with the built-in functions and its data structures along with algorithms to be able to use these systems efficiently.

6. Further, these computer algebra systems still need to develop their mathematical base to handle any kind of mathematical problem. They also need to develop better user interfaces as well as interfaces with other programming languages.

1.6 Thesis Scope

A Specialization and Simplification Tool (SAST) is developed as a part of this research to automate simplification of any expression based on given constraints.

Specialization is explained in section 2.6. This work deals with applying the specialization technique to tabular expressions specifying a program, not to the program itself. This work does not consider simplification of tables by transforming one type of table to another. Transformations are described in [22, 23]. All tables are assumed to be proper and all functions are assumed to be total.

The types of tables (described in [15]) that SAST can handle are as follows:

- Normal Function
- Inverted Function
- Vector Function
- Predicate Expression
- Inverted Predicate Expression

The scope of SAST for this thesis is limited to exploiting of the existing capabilities of Maple V (Release 4 version 4.00f). Moreover, the scope of variables in this work is limited to only Reals and Booleans, with no sets, lists or arrays, or default domain of Maple (Complex). Because Maple does not support recursive definitions, no user-defined functions with recursion are supported by the SAST.

1.7 Features of SAST

This section lists the various features of SAST within the given scope. Chapters 3 4, 5, and 6 to provide more insight into the technique and the tool, its design, and its limitations and applications. Following are the features of SAST:

- SAST allows automated specialization and simplification of both conventional and tabular expressions based on user defined constraints. It can handle all commonly used types of tables (see section 1.6) as well as tables embedded in a conventional expression.
- User defined global constraints (of the form: $x_1 = a_1 \wedge ... \wedge x_r = a_r$, where, a_1, ..., a_r are constants or symbolic constants) result in specialization and simplification by assignment. Since SAST accepts values for the variables it can also perform full/partial evaluation of expressions. These constraints can be on both input and output[1]

[1] As described in [9], relations/functions can be defined on the Cartesian product of sets as **IN** $\times$ **OUT**, where **IN** denotes the sets of input values and **OUT** denotes the sets of output values.

variables.

- The input expression is not modified.
- All definitions (both user defined and default) for the expression are globally declared while initializing the Maple session. Since there is only one Maple session that is active, all definitions declared once remain valid for the entire session.
- SAST utilizes the symbolic computation power of Maple to yield accurate results close to an intuitive form. Example: 30/9 gives 10/3 rather than 3.333333 (based on the default precision).
- Simplification of tables can result in the removal of rows and/or columns based on the cell expressions.
- SAST can also perform assignment of a variable to a numeric constant or symbolic constant in an expression with quantifiers (Existential and Universal). Value is assigned only to the free variables[2] in the expression with quantifiers.
- SAST can accept quoted variables like `x and is able to distinguish a variable name with numeric characters and a numeric value of the variable. For example: Variable with name 123 and a numeric value 123.
- The design of SAST makes it is very easy to modify and to extend its functionality while maintaining the modularity of the software. Refer to Chapter 3 for SAST design and module decomposition. For example: simplification/specialization capabilities of SAST can be extended by adding more procedures in Maple syntax.

1.8 Thesis Outline

Chapter 2 discusses the predicate logic and tabular notation used in SERG, table semantics and an overview of the on going Table Tool System (TTS) project. This chapter also gives a discussion about simplification, specialization and partial evaluation, and some issues related to using specialization technique. It also gives a description of specialization and simplification of expressions.

---

[2] An occurrence of a variable x is said to be *bound* in an expression if either it is the occurrence of x in a quantifier in the expression "∀x" or it lies within the scope of a quantifier "∀x" in the expression. If the occurrence is not bound then it is free. A variable is said to be *free* if it has free occurrence.

Chapter 3 includes a module guide for SAST, uses relation and access programs for the modules with their description. This chapter also describes requirements, assumptions, some anticipated changes for the tool and an overview of various components of SAST. It also includes a user's guide and an overview of the algorithm used for SAST.

Chapter 4 highlights some design decisions of SAST including handling of TTS expressions and their conversion to prefix form, how and when the rows/columns are to be removed in a tabular expression, initialization of Maple session, interface with Maple, and functions dealt with by Maple procedures.

Chapter 5 describes various limitations of SAST which are either due to Maple or due to other support modules used in SAST. It also describes limitations resulting from design decisions for SAST.

Chapter 6 describes the results, applications of the research, conclusions, and future work.

Appendix A gives the Module Interface Specification for the TTS_Maple module which hides the interface of SAST with Maple.

Chapter 2

2. Notation and Terminology

This chapter provides an introduction to predicate logic, tabular notation and its semantics. It also briefly discusses the Table Tool System (TTS) that is used to build SAST. A description of specialization and simplification of expressions is given here. A discussion about simplification, specialization and partial evaluation is also covered in this chapter.

2.1 Predicate Logic

The logical foundation of tabular notation and this work is predicate logic, as described in [16]. This logic caters to a basic requirement of the development of precise software by unambiguously yielding ***true*** or ***false***[3] while evaluating (given assignment for all variables) the logical expressions describing observable or intended behaviour of programs. Terminology used in this thesis is as follows:

Function Application

A function application is a string consisting of a function name and arguments. For example: y + 3, f(a, b), etc. The arguments are terms. For example: In application f(a, b), *f* is a function name; and *a* and *b* are arguments.

Terms

Terms are constants, variables, or function applications. For example: 2,

[3] "***true***" is a predicate value. It is written in this manner to differentiate between "true" used as a value of a boolean program variable and as a predicate value. A predicate value ***true*** characterizes a set which includes the whole universe and ***false*** characterizes an empty set.

"true"(represents a value of a program variable), y, f(x,y) are terms. Constants can have numeric or symbolic values.

Primitive Relation

A *relation* is a set of tuples (refer to [16] for the definition of tuple). A small set of relations (e.g. <, >, =, etc.) are designated as *primitive relations*. A *binary relation* is a set of 2-tuples (pairs). In a binary relation, the set of values that appear as the first element of a pair is called the *domain* and the set of second values is called the *range*. A *function* f is a special case of a binary relation with an additional property that for an element x in the domain of function, there is a unique element y such that $(x, y) \in f$. A *predicate* is defined as a function whose range contains only ***true*** and ***false***. An important aspect of this logic is the evaluation of a primitive relation for a given assignment outside its domain to ***false***. For example: Assume > and ≤ to be primitive relations, then both $\sqrt{x} > \sqrt{y}$ and $\sqrt{x} \leq \sqrt{y}$ are ***false*** when x and/or y have negative values. Note that in this logic "f(x) = f(x)" is equivalent to "x is in the domain of f" not to ***true***. In SAST, "f(a) = f(a)" is simplified to ***true*** when the definition of f is given and value of the argument a is given.

Predicate Expressions

A predicate expression is a primitive relation or a string of the form (P), $P \wedge Q$, $P \vee Q$, $\neg P$, or $(\forall x, P)$, where P and Q are predicate expressions and x is the index variable of quantification.

2.2 Tabular Notation

Tables are constructed recursively from scalar expressions (which are either terms or predicate expressions), and grids. Definitions and the syntax of tables are discussed in [15] and [10]. Tables are described as collection of sets of cells classified into headers and main grid as follows:

- A header H is an indexed set of cells with $H = \{h_i | i \in I\}$ where $I = \{1, 2, ..., n\}$ is a set of indices.
- A grid G is an indexed set of cells, $G = \{g_\alpha \mid \alpha \in I\}$. G is indexed by headers $H_1, ..., H_n$, with $H_j = \left\{h_i^j \mid i \in I^j\right\}$ where j = 1, ..., n and $I = I^1 \times ... \times I^n$. The set I is the index of G.

A function table is also a term. A predicate table is also considered as a

predicate expression. A table is *proper* if the domains of the subfunctions (of the function represented by the table) are disjoint. In general, a table is proper if and only if, the conjunction of all the conditions is equivalent to *false*. For more details on properness of a table refer to [15, 22].

2.3 Tables Semantics

The model of table elucidated in [10] and [1] covers many more table types than those described in [15]. The foundation of this general model is the cell-connection graph (CCG), characterizing information flow among the cells and is used for interpretation of tables by SAST. A cell connection graph ($\rightarrow$) is a relation described by an acyclic directed graph which has grids associated with as its nodes. For the cell connection graph, it is required that each arc must either start or end at the main grid G. The transitive and reflexive closure[4] of $\rightarrow$ is a partial order[5]. The classes of CCG as described in [1], are "Normal, Inverted, Vector or Decision". Some definitions are as follows (refer to previous section for tabular notation):

- The elements of the set Components(T) = $\{H_1, ..., H_n, G\}$ are called table components.
- A table component $A \in$ Components(T) is *maximal* if $A \rightarrow B$ implies A = B for every $B \in$ Components(T).
- A table component $A \in$ Components(T) is *minimal* if $B \rightarrow A$ implies B = A for every $B \in$ Components(T).
- If a table component is minimal or non-maximal implies that, its cell expressions contribute in describing the domain of a relation/ function, R, specified by the table. The cells from these components are called *guard cells*.
- If a table component is maximal it implies that its cell expressions become a part of *value cells* that describe the evaluation of the function or relation, R, specified by the table.

---

[4] $(A = B) \vee (A \rightarrow B) \vee (\exists A_1, ..., A_k. A \rightarrow A_1 \rightarrow A_2 \rightarrow ... \rightarrow A_k \rightarrow B)$

[5] A *partial order* in a set A is a relation R in A which is reflexive ($(a,a) \in R$), anti-symmetric ($(a,b) \in R$ and $(b, a) \in R$ implies a = b) and transitive ($(a, b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$). For example: The relation R in natural numbers N defined by "x is a multiple of y" is a partial order in N.

- As described in [1], a *raw element* is a tuple $e_\alpha = (h^1_{i_1}, .., h^{n1}_{i_n}, g_\alpha)$ where $i_j \in I^j$, j = 1, ..., n, $\alpha = (i_1, ..., i_n) \in I$.

The ease in understanding and specifying a complex function or a relation R using tables is based on the fact that it can be decomposed into R_α where $\alpha \in I$. The cell connection graph gives the decomposition of cells into guard and value cells.

f_irr_band = **H_2**

p_T: $H_1 \wedge G$ **r_T**: H_2	low_irrational	rational	high_irrational
s_irr_band = high_irrational	m_sig ≤ 100	100 < m_sig ∧ m_sig ≤ 4900 - db	4900 - db ≤ m_sig
s_irr_band = low_irrational	m_sig < 100 + db	100 + db ≤ m_sig ∧ m_sig < 4900	4900 ≤ m_sig
s_irr_band = rational	m_sig ≤ 100	100 < m_sig ∧ m_sig < 4900	4900 ≤ m_sig

H_1 **G**

FIGURE 8 - Inverted Function Table

For example: FIGURE 9 is a raw element of FIGURE 8. The table components, CCG, guard and values cells of the raw element, $e_{1,1,}$ are as follows:

- H_1, H_2 and G are table components.
- CCG is Inverted.
- Cells h^1_1 and $g_{1,1}$ are the guard cells.
- Cell h^2_1 is the value cell.

Table rules are required to show how to built a logical expression, R_α, from the expressions in the guard and value cells. These table rules are defined as:

- A *table predicate rule*, p_T, is the predicate expression formed by the table components having guard cells. This rule helps in defining the domain of the relation/function specified by the tabular

expression.

FIGURE 9 - A Raw Element of FIGURE 8

- A *table relation rule*, r_T, is the relational expression formed by the table components having value cells. This rule helps in determining the value of the expression.

For the same inverted function table example, the table rules and R_α are as follows:

- p_T is $H_1 \wedge G$.
- r_T is H_2.
- The sub-relation, R_α, for the raw element in FIGURE 9 is $((s\_irr\_band = high\_irrational) \wedge (m\_sig \leq 100)) \Rightarrow (f\_irr\_band = low\_irrational)$.

There are nine R_αs for the relation represented by the table in the FIGURE 9. For the complete representation of the relation R by the table, all the sub-relations, R_α's have to be combined using a compose operator. The compose operator forms a part of *table composition rule*, C_T, that is built from relation/function names, indexes, and operators as described in [10]. Apart from this, a mapping, Ψ, is also needed that assigns a predicate expression, or a portion of it, to guard cells, and a relation expression, or a part of it, to the value cells. Thus, the relation R represented by a table may be written as:

$$R = C_T(R_\alpha)$$

For the example in FIGURE 8, the C_T is as follows:

- $\underline{Y_{j=1}^{3} Y_{i=1}^{3} R_{i,j}}$

Finally, the tabular expression is defined as a tuple:

$$T = (p_T, r_T, CT, CCG, H_1, ..., H_2, G; \Psi)$$

where,

- p_T is a table predicate table
- r_T is a table relation rule

- $\{H_1, ..., H_2, G\}$ are table components
- CCG is a cell connection graph, $\rightarrow$
- C_T is a table composition rule
- Ψ is a mapping

2.4 Overview of Table Tool System (TTS)

The Table Tool System is an on-going project of the Software Engineering Research Group (SERG) at McMaster University to enhance and support the use of tables in software documentation. Often, the low quality of software documentation is attributed to a lack of tools and techniques. TTS is an integrated Computer Aided Software Engineering (CASE) tool set that can manipulate tables occurring in the relational model of documentation for complex systems. TTS capabilities include entering, storing, modifying, formatting, evaluation and transforming tabular expressions as well as specification checking and test oracle code generation. TTS is decomposed into three parts as below:

2.4.1 TTS Kernel

The TTS kernel hides the representation of expressions as well as basic algorithms for their manipulation. The modules in TTS kernel provide:

- Storage of expression structure in the form of a tree. This is accomplished by the Table Holder (TH) module within the kernel.
- Information needed for interpretation and presentation of the expressions. This is provided by the Information (Info) module. Tools and applications, not the kernel, are responsible for interpretation of this information. The Info module maintains a symbol table. The symbol table has information corresponding to various symbols subdivided into classes.

2.4.2 TTS Infrastructure

The modules in TTS Infrastructure provide facilities to develop TTS applications. Some definitions involved are:

- A *service* is an invocation of a procedure by single command.
- A *context* is a file containing an ordered collection of named

expressions with an associated symbol table comprising information about default symbols and expression specific symbols, a list of different classes and a list of identifiers.

The Infrastructure forms the middle part of TTS and includes following modules:

- *Tools* operate on expressions to provide services like Context Manager (CM) to manipulate expressions in a context, Table Printing Tool (TPTool) to print and display, Table Construction Tool (TCT) to edit and construct expressions, Specialization and Simplification Tool (SAST) to simplify expressions based on user defined constraints, etc.
- The *utilities* module provides help to the tools in interpretation of table semantics, manipulation of symbol tables and identifiers, and traversal of expressions.
- The *Tool Integration Framework* (TIF), which provides the interface between tools, incorporates the ideas of "separation of concerns" and "information hiding" for seamless integration of new tools/applications to TTS. This framework is needed to facilitate the ongoing evolution of TTS. TIF enables tool developers to call different tools and pass expression, string or context between them without having any knowledge about the interface with the TTS.

2.4.3 TTS Applications

The topmost layer comprises of applications that operate at the documentation level to support the relational documentation model and to edit, analyze or interpret the documentation.

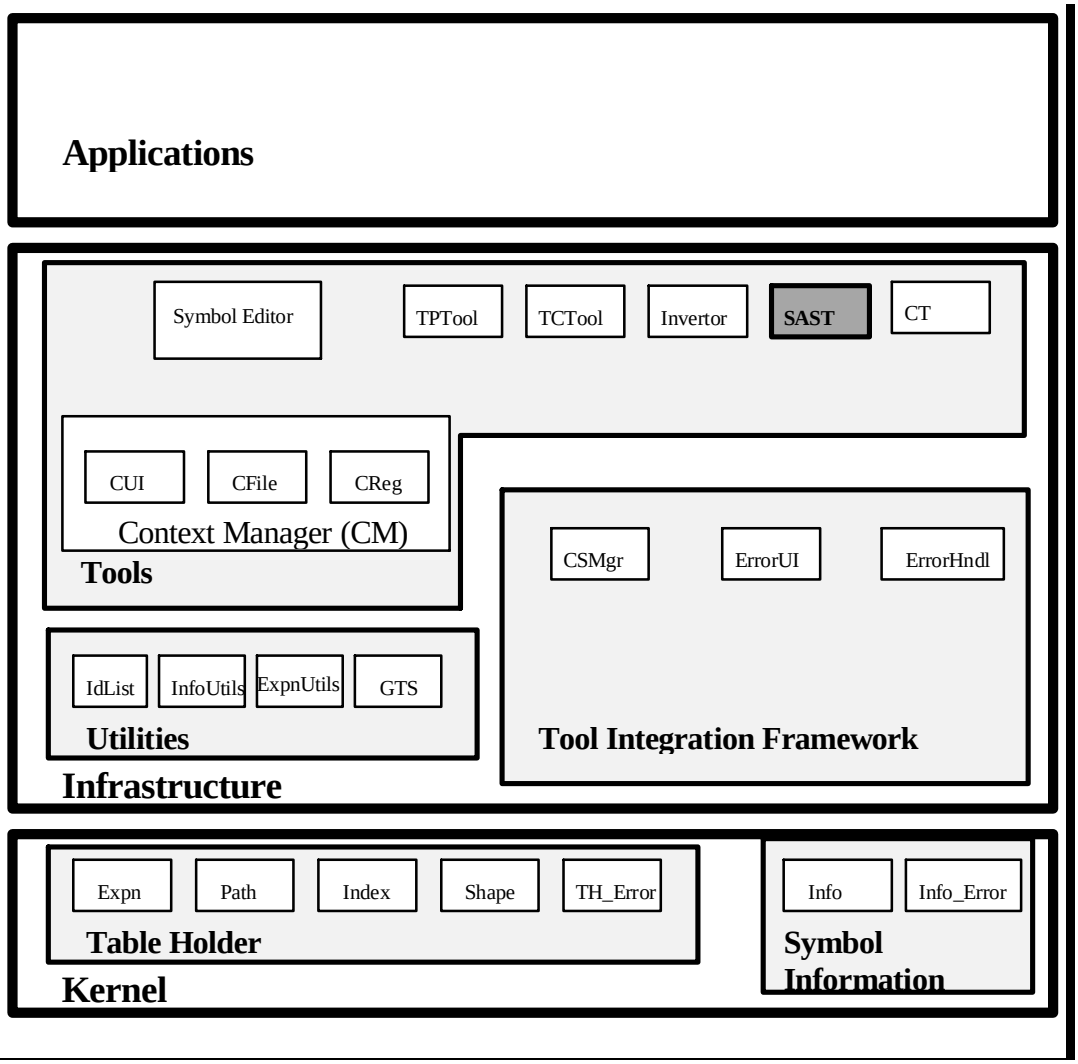

FIGURE 10 - TTS Design Scheme

FIGURE 10 illustrates some modules and their positions in the layered structure of TTS. This figure has been adapted from [24] to show only those components related to SAST. Utility tools like TCT and TPTool are used as general support tools for input and display of tables respectively. Tabular expressions from table composition tool (CT) and table conversion tools such as Table Invertor Tool can be used as an input to SAST. The TIF may be used to integrate SAST with other tools in future. Detailed design aspects of SAST and its interaction with other modules of TTS are given in chapters 3 and 4.

2.5 Simplification

Simplification in general is hard to define. The main reason of this problem is that an expression can be written in several equivalent forms. Based on context and the intended use of the expression, one form may be considered simpler than the other. Consider an example of a polynomial $8x^2-12x$.

Some equivalent forms of this polynomial are:

1. $x(8x^2-12)$
2. $4x(2x^2-3)$
3. $2x(2x-\sqrt{6})(2x+\sqrt{6})$
4. $(2x)^3-6(2x)$

If the criterion used to select the simplest form is the number of terms then $8x^2-12x$ and $(2x)^3-6(2x)$ are more suitable than others but if the criterion is that the selected form must be a polynomial in 2x, then $2x(2x-\sqrt{6})(2x+\sqrt{6})$ and $(2x)^3-6(2x)$ are better suited. Refer to [8] for more details.

Simplification can also be explained as a set of transformations that change one form of expression to another while retaining the meaning. Each CAS has a few forms that it considers simple in a given context.

For example:

1. `5` is simpler than $\sqrt{4}+3$
2. $(x+1)^3$ is simpler than x^3+3x^2+3x+1
3. $1+\frac{6}{(x-1)^5}+\frac{15}{(x-1)^4}+\frac{20}{(x-1)^3}+\frac{15}{(x-1)^2}+\frac{6}{(x-1)}$ is simpler than $\frac{(x^2-1)(x^2-x+1)(x^2+x+1)}{(x-1)^6}$ for integration

The above examples show dependence of simplification on the context. In example 1, `5` is always considered simpler than $\sqrt{4}+3$ but in example 2, some may prefer x^3+3x^2+3x+1 over $(x+1)^3$. In example 3, the second form may be preferable for computation or evaluation to the first form. Example 3 can also be

simplified to $\frac{(x^6-1)}{(x-1)^6}$.

Let ~ be an equivalence relation on a class of expressions represented by ε. An equivalent but simpler expression can be found based on a computable transformation $S:\varepsilon \rightarrow \varepsilon$ such that:

$$S(t) \sim t \text{ and } S(t) \ \underline{\pi} \ t$$

where $\underline{\pi}$ denotes some criteria for simplification. $s \ \underline{\pi} \ t$ means *s* is simpler than *t* in ε. The criterion of finding a simpler expression can be lesser number of terms, then $s \ \underline{\pi} \ t$ would mean *s* has lesser number of terms than *t*. Some other criteria for selection of a simpler expression may be for example, less memory or more readability. Refer to [8] for further discussion.

For Maple, *s* and *t* are identical if they have same internal data representation. For discussion on internal representation of data in Maple refer to [4,8]. Most CAS, in general assume that an expression with fewer number of symbols than the original expression is a simpler expression.

SAST performs logical and numeric simplifications.

For example:

1. true ∧ false simplifies to false
2. x + 3 + 2 simplifies to x + 5
3. true ∧ expn simplifies to expn

SAST can easily be enhanced to invoke different simplification engines of Maple.

For example:

1. By calling procedure *normal*, the expression $\underline{(x-2)^2 - x^2 + 4x - 4}$ can be simplified to 0.
2. By using Maple procedure *simplify/trigonometric*, $\underline{\cos(x)^2 + \sin(x)^2}$ can be simplified to 1.

2.6 Specialization

Tables help in ensuring that all the possible situations are covered in an application thus reducing the chances of encountering undesired program

behaviour. Analysis or inspection of tables for large and complex software is usually done on a per case basis. Often some cases will never occur during normal operation and as such may not be included in the given normal domain. Removal of such cases through specialization and simplification will provide a clearer view of the remaining cases and contribute towards better understanding.

2.6.1 Program Specialization

The concept of specialization is relatively old and is often used to restrict the scope of the solution to a problem. Most recently (1990s) published work focuses on specialization of programs with respect to some input using partial evaluation. Specialization is often used to tailor optimal solutions that cater to specific requirements of problems and to enhance the operational efficiency of the software. Specialization techniques have been applied to programs using transformation rules as described in [21]. The authors in [7] view specialization as a technique to enhance reuse of software modules components or software development process products such as, specifications or design manuals based on symbolic execution. In [7], specialization is also justified as a technique for re-engineering to maintain and optimize software components. Some examples of generally used components are - configuration of operating systems for specific types of computers; and pre-defined conditional compilation of programs. An application of specialization to generic operating system code using partial evaluation is shown in [20]. Some other applications of partial evaluation, that form the basis of specialization as described in [6], are: compilation and compiler generation; string and pattern matching; computer graphics; numerical computation; circuit simulation and hard real-time systems.

2.6.2 Expression Specialization

The work in this thesis is different from the other recent work [20] in that here specialization is applied to mathematical expressions (scalar and tabular) rather than to program code. Also, the tool developed as a part of this work, simplifies expressions based on the given expressions. Specialization, in this thesis, means restriction of the domain for which an expression is valid. An expression is valid if it correctly and completely specifies the behaviour of the program within the considered domain. A specialized expression (tabular or scalar) is not necessarily equivalent to the original expression outside the constrained domain.

The restriction in domain is based on user defined global constraints that result in partial/full evaluation of the given expression. If values of all the variables are given, the expression is fully evaluated.

Constraints or restrictions are global conditions that are true for a given expression in a given domain. These constraints can be any logical expression.

Conjunction of constraints is also viewed as a guard under which the specialized expression is equivalent to the original expression. In the case of no constraints, specialization of an expression results in the same or possibly a simplified expression. Specialization allows for the simplification of mathematical expressions.

Specialization and simplification of a tabular expression involve only algebraic and logical simplifications under the given constraints.

For example: $(x > 2) \wedge ((x + y) < 2)$ under the constraint $x = 5 \wedge y = 6$ specializes to $(5 > 2) \wedge (5 + 6) < 2$, which then simplifies to *false*.

Specialization and simplification for a tabular expression involve both reductions in the number of dimensions and/or rows and/or columns, and simplification of cell expressions based on the given constraints. Thus, the generality of the original table is traded off with the clarity of specialized tables. A specialized table will still be proper in the restricted domain. Thus, for tabular expressions, simplification can occur in two levels:

1. **Cell level**: involves algebraic and logical simplification of cell expressions using the given constraint information.

2. **Table level**: involves reduction of table size by removal and/or merging of rows and/or columns and in some cases may even eliminate of dimensions based on the table class.

A table subjected to constraints can be specialized to a term or to a simpler table that is easier to understand and interpret. A table is considered to be simpler than its original table if it has any/all of the following properties:

- Reduced length of the main grid in any dimension
- Reduced number of dimensions

f_irr_band = **H_2**

$\mathbf{p_T}$: $H_1 \wedge G$ $\mathbf{r_T}$: H_2	low_irrational	rational	high_irrational
true	m_sig < 115	115 ≤ m_sig ∧ m_sig < 4900	4900 ≤ m_sig

H_1 **G**

FIGURE 11- Specialized Table for FIGURE 8

An example illustrating the concept of specialization of tabular expression is shown in FIGURE 11 which is a specialized form of FIGURE 8 (in section 2.3) under the constraints:

s_irr_band = low_irrational, db = 15.

2.6.3 Description of Specialization and Simplification of Expressions

A description of specialization and simplification of tabular and scalar expressions in SAST is given in following two sections.

2.6.3.1 Tabular Expressions

For all classes of tables within the scope of this thesis, let $T = (H_1, ..., H_n, G)$ represent an n-dimensional table with n-1 one dimensional headers and an n-dimensional main grid.

The specialization of table T may be represented as $\underline{\overline{\overline{T}}}$, as follows,

$$\underline{\overline{\overline{T}} = \textit{specialize}(V, T)}$$

where V is the substitution values of the form $\underline{(x_1 = a_1 \wedge \ldots \wedge x_r = a_r)}$. x_r is any arbitrary variable and a_r is a constant or a symbolic constant.

Specialization involves substitution that simplifies expressions in each of the cells. Let *subs*(V, T) be defined as follows:

$$\textit{subs}(V, T) = \underline{\overline{T}}$$

$$\underline{\overline{T} = (\overline{H_1}, ..., \overline{H_n}, \overline{G})}$$

where *subs* is simultaneous syntactic substitution and

$$\underline{\overline{H_j} = \{\textit{subs}(V, h_i^j) | i \in I^i\}, j = 1, ..., n}$$

$$\underline{\overline{G} = \{\textit{subs}(V, g_\alpha) | \alpha \in I\}}$$

In the case of tabular expressions, along with logical and algebraic simplifications of the cell expressions, removal of rows and/or columns may occur which leads to further simplification and is represented as *simplify*($\underline{\overline{T}}$). Following are two situations in which such removal is possible:

- If $\underline{\overline{h}_i^j}$ is a guard grid and is ***false***, then the length of H_j and $len_j(G)$ is reduced.
- If CCG is INVERTED and all of the cell expressions in a given

slice (as described in [22]) of $\bar{G}$ are ***false***, then that slice of the main grid along with the corresponding header cell $\bar{h}_i^j$ in that dimension are removed.

(1) and (2) reflect the above definitions to give,

$$\bar{\bar{G}} = \{\bar{g}_\alpha \mid \alpha \in I\} \text{ where } I = \bar{\bar{I}}_1 \times \ldots \times \bar{\bar{I}}_n \quad (1)$$

$$\bar{\bar{H}}_j = \{\bar{h}_1^j, \bar{h}_2^j, \ldots, \bar{h}_{i+1}^j, \ldots \bar{h}_m^j,\} \quad (2)$$

where $\bar{\bar{I}}_j = \{1, \ldots, m\}$ where m < k and k is the original length of I_j.

A specialized table is given by using (1) and (2) as,

$$\bar{\bar{T}} = (\bar{\bar{H}}_1, \ldots, \bar{\bar{H}}_n, \bar{\bar{G}})$$

where $\bar{\bar{G}}$ and $\bar{\bar{H}}$ can also be defined as follows:

$$\bar{\bar{G}} = \textit{simplify}(\bar{G})$$
$$\bar{\bar{H}}_j = \textit{simplify}(\bar{H})$$

Thus specialization can be represented in terms of *simplify* as follows:

$$\textit{specialize}(V, T) = \textit{simplify}(\textit{subs}(V, T))$$

2.6.3.2 Scalar Expressions

For a scalar expression E, specialization is represented as follows:

$$\bar{\bar{E}} = \textit{specialize}(V, E)$$

Here also specialization involves simultaneous syntactic substitution, *subs* as:

$$\bar{E} = \textit{subs}(V, E)$$

Logical and algebraic simplifications, *simplify*, of the specialized expression $\bar{E}$ leads to a simplified expression, $\bar{\bar{E}}$ that is defined as follows:

$$\bar{\bar{E}} = \textit{simplify}(\bar{E})$$

Thus the process of specialization in SAST for a scalar expression can be summarized as:

$$\textit{specialize}(V, E) = \textit{simplify}(\textit{subs}(V, E))$$

2.6.4 On Specialization, Simplification and Partial Evaluation

Specialization means restricting the domain for which the expression is valid. The meaning of the expression must remain unchanged over the rest of the domain.

> Example 1: Under the constraint $x > 2$, the expression $(x + y) > 4$ specializes to $(x + y) > 4 \wedge (x > 2)$.

A special form of specialization is by assignment of values to variables.

> Example 2: Consider an expression $x + y + 0 \leq 7$ where x and y are positive non-zero natural numbers. Under the constraint $x = 5$, the expression specializes to $5 + y + 0 \leq 7$.

Simplification results in an expression that is equivalent to the original expression in the given domain.

> Example 3: Refer to specialized form of expression in example 2, a transformation $y + 0 \rightarrow y$ will simplify the expression to $y \leq 2$.

Partial evaluation involves evaluating an expression based on given values.

> Example 4: Consider an expression $f(x, y) = x^2 + y^2$. Under the constraint $x = 3$ the expression becomes partially evaluated to $f(3, y) = 9 + y^2$.

2.6.5 Using Specialization

Besides reducing the domain under consideration, some issues to consider when using the specialization technique are:

- There is a loss of generality as the domain under consideration is reduced.
- Various combinations of constraints might result in numerous tables. Each expression will be simpler than the original expression for the given constraints. This thesis does not deal with the management of these specialized tables.
- It might be difficult to create the original table from specialized

tables or a set of specialized tables because of the overlapping domain and loss of mutual exclusivity. The original tables are assumed proper. In SAST, a copy of the original expression is maintained and the specialized expression is a new expression saved in the context file.

Chapter 3

3. Specialization and Simplification Tool Design

This chapter discusses the design of the Specialization and Simplification Tool. Informal requirements, assumptions and their justifications, user's guide and anticipated changes to the tool are described. A module guide, access programs for the modules and an overview of the algorithm used for SAST are also covered in this chapter.

3.1 Requirements (Informal)

SAST should be TTS compatible and should accept input in a context file as described in section 3.1.2.2. It should be able to specialize and simplify the expression based on the given constraints using a Computer Algebra System. Elimination of rows and/or columns and/or dimensions, based on certain predicates in the cells of the tabular expression being *false* under given constraints, should be possible.

3.1.1 Assumptions

It is assumed that all the expressions (both constraints and the expression to be simplified) are in the context file. The last expression is the expression that is to be simplified and all other expressions, if any, are constraint expressions. Tables are assumed to be proper.

At present, SAST performs only specialization by assignment and can handle constraints only of the form x = c (Equal(x, c)), where c is a constant. The assumption of specialization by assignment is justified as after simultaneous substitution of given values for variables in the expression, the simplified expression is equivalent to the original expression in the given domain.

3.1.2 User's Guide

3.1.2.1 Operating Requirements

SAST requires the presence of the following software:

- Maple for the computational needs of SAST.
- TTS Kernel and its Infrastructure for manipulation and traversal of the expression. For further details refer [24] and section 2.4.

3.1.2.2 Input

The input of SAST is a file containing expressions and a symbol table covering the information about the symbols used in the expressions. The expressions in the input file are global constraints and the expression to be simplified. All the expressions except the last one are global user constraints and the last one is the expression that is subjected to the above constraints. The context file is created using the context manager and Table Construction Tool (TCT) of Table Tool System (TTS). A detailed description of all the tools in TTS is given in [24] and a brief description of TTS is given in section 2.4.

3.1.2.3 Execution

SAST has a very simple command line interface with the following syntax:

sast *filename*

where *filename* is the name of the context file with "tts" extension.

3.1.2.4 Viewing and Printing of SAST Results

The result of SAST is appended to the same input context file. It is the last expression in the context file with name "SAST_newExpn". Context manager and TCT of TTS are used to view the results. The expression can also be printed using Table Printing Tool (TPTool). Context Manager, TCT and TPTool are already existing tools in TTS (refer to section 2.4 for an overview of TTS).

3.1.3 Anticipated Changes

Following are some of the areas where changes are anticipated during the effective life of SAST:

- Method to obtain input expression.
- Access programs of Info, Parser, GTS and CM modules (These

TTS modules were encapsulated in SAST as they were either being developed or were being modified during the development of SAST).

- Computer Algebra System.
- SAST interface with Maple.
- Syntax of intermediate forms of expressions (A colon instead of semi-colon as delimiter in prefix form of expression, postfix form instead of prefix form etc.).

3.2 SAST Overview

A schematic diagram of SAST illustrating its three major components is shown in FIGURE 12. The TH helps SAST in the storage and retrieval of expressions, and provides all the necessary infrastructure for expression manipulation. Expressions used in SAST can be entered using Table Construction Tool (TCT) and printed using Table Printing Tool (TPTool). The user's guide given in previous section includes more details about the usage of SAST. The three major components of SAST are described as follows:

- **User-SAST Interface**: This interface is used to accept input save output expressions for SAST. It hides the assumptions about the constraints and the expression to be simplified. Refer to User_SAST Interface in section 3.3.2.1 for further details.
- **TTS-Maple Interface**: SAST includes a interface to Maple for sending expressions and receiving results from Maple. All the expressions are sent in a prefixed form (for reasons explained in section 4.2). The results obtained from Maple are parsed into TTS compatible form. This interface is established through a UNIX pipe. All the expressions and commands for Maple are sent through this pipe.
- **Maple-Simplify**: Maple is used as a math engine to perform computations in SAST as described in Chapter 1. As a part of this work, procedures are written in Maple syntax to handle simplification of the expressions. These procedures are described in section 4.7 and section 3.3.2.14.

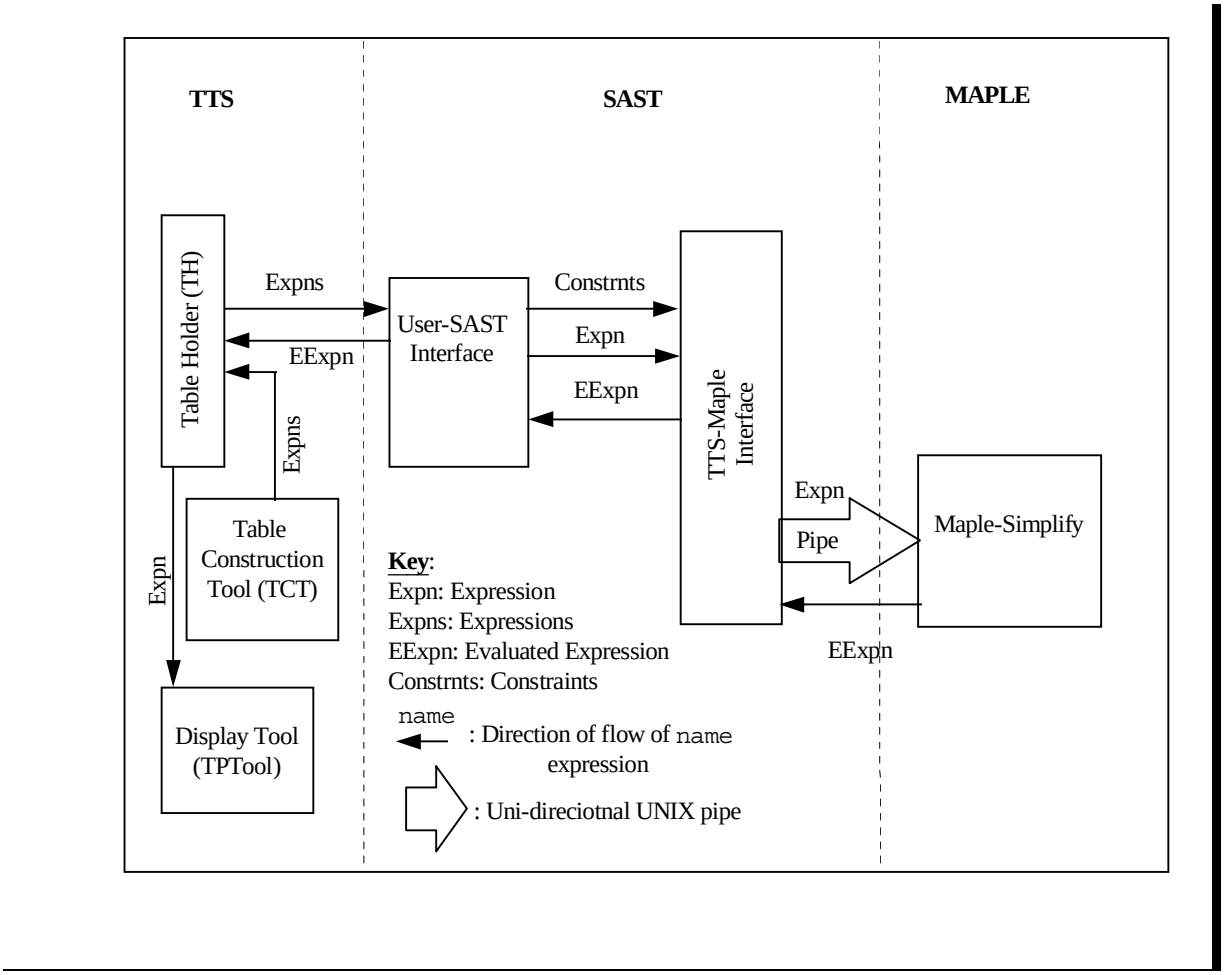

FIGURE 12 - SAST Schematic Overview

3.3 Module Decomposition

3.3.1 Module Uses Hierarchy

SAST is decomposed into modules that hide design decisions that are likely to change. This modular structure of SAST makes it easy to understand and modify.

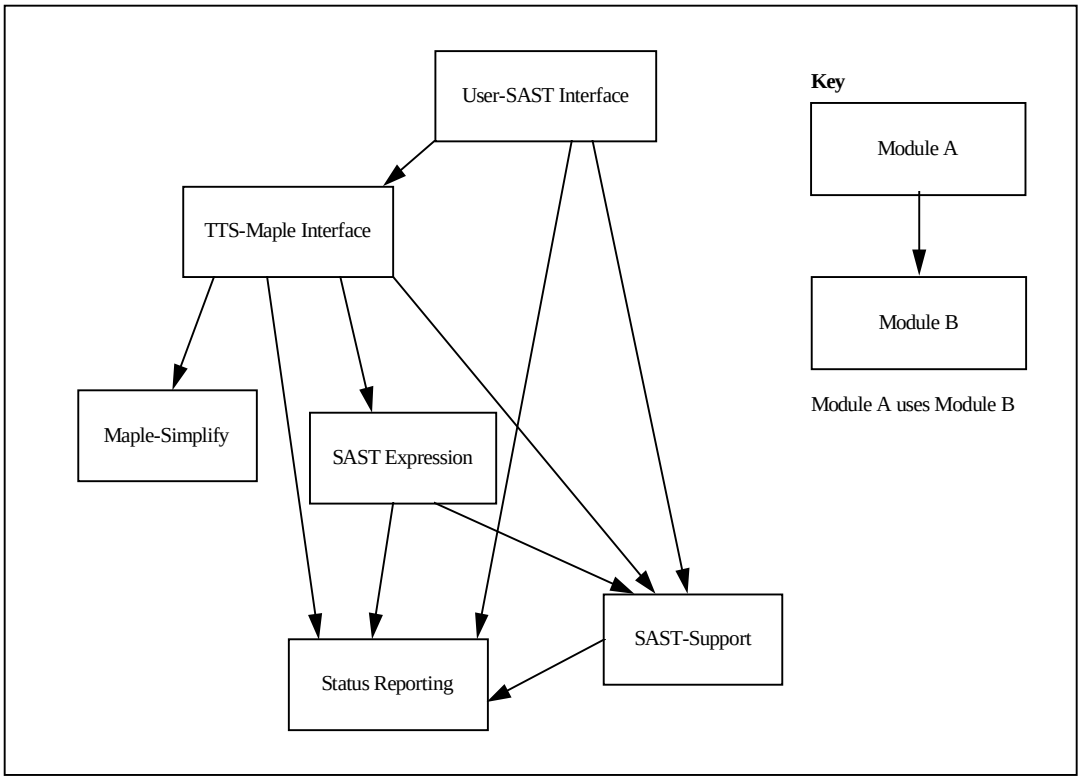

FIGURE 13 - Uses Relation of Top Level

FIGURE 13 illustrates the top-level *uses relation* (derived from the *program uses relation* as described in [13]). Module A uses Module B if at least one program in Module A rely on the correct behaviour of some programs in Module B to accomplish its task.

TABLE 1 gives the lower level uses relation for all the modules developed or modified for SAST.

Module	Level	Uses
User_SAST Interface	9	TTS_Maple Expn Interface SAST_Info SAST_Context SAST_Line Buffer Status Reporting

Module	Level	Uses
TTS_Maple Expn Interface	8	SAST_Maple Link Maple Command Constructor SAST Output Expression SAST Utility SAST_Line Buffer Status Reporting
Expression	7	Carve Maple Code Generation TTS_Maple Line Interface Maple_TTS Interface Semantics SAST_Info SAST_Line Buffer Status Reporting
Maple Code Generation	6	Table Application Variable Constant SAST Utility SAST_Info Status Reporting
TTS_Maple Line Interface	5	SAST_Maple Link SAST Output Status Reporting
Table	5	SAST_Context SAST Utility SAST_Maple Link SAST_Info Status Reporting
SAST Utility	4	SAST_Context Status Reporting
SAST_Maple Link	4	Maple Command Constructor SAST_Line Buffer SAST_Info Status Reporting
Maple_TTS Interface	3	SAST_Info Status Reporting
Application	3	SAST_Info Status Reporting
Variable	3	SAST_Info

Module	Level	Uses
		Status Reporting
Constant	3	SAST_Info Status Reporting
SAST_Context	3	SAST_Info Status Reporting
Maple Command Constructor	3	SAST_Line Buffer Status Reporting
SAST Output	3	SAST_Line Buffer Status Reporting
Semantics	2	Status Reporting
Carve	2	Status Reporting
SAST_Info	2	Status Reporting
SAST_Line Buffer	2	Status Reporting
Status Reporting	1	Standard libraries

TABLE 1 - Uses Relation for SAST

3.3.2 Module Guide

This section includes a module guide that covers a general description of all the SAST modules and their access programs. Module guide helps in understanding the overall structure of SAST and mainly includes the modules developed as a part of this work. It also describes briefly other modules that are used by SAST. For each module following information is described:

1. The secret encapsulated by the module.
2. The facilities provided by the module.

3.3.2.1 User_SAST Interface Module (SAST_main.c)

The interface between the user and SAST is encapsulated in this module. It also hides the format and assumptions about the input context file. This module acts as a control module to invoke other modules in SAST. The command line argument that is used to invoke SAST (either by user or other programs/applications) is also hidden here. It uses the TTS_Maple interface to send expressions and commands to Maple for declaration of global constraints and to simplify expressions. It also performs the necessary initialization of the TTS-Maple interface and clean up for SAST.

3.3.2.2 TTS_Maple Interface (TTStoMaple.c)

TTS_Maple interface hides the link established between the tool and Maple. This module provides access programs to initialize Maple session, send expressions, call a procedure in Maple, read and write Maple results to a file and to clean up all the temporary files. All the definitions stored in the MapleDefn class of the symbol table can be used in the expressions and therefore the Maple session is initialized with these definitions. Refer to TABLE 2 for access programs and their descriptions. Refer to Appendix A for a module interface specification of the TTStoMaple module.

Access Program	Description
`void TTStoMaple_init (SymTbl t)`	Initializes the module to a known initial state using information from symbol table `t`. Sets up the UNIX pipe to link the SAST and Maple.
`void TTStoMaple_sendExpn (Expn expn, Path p)`	Sends expression `expn` at path `p` to Maple.
`void TTStoMaple_callProcFSave (char *p, int arity, ...)`	Invokes a Maple procedure `p` with `arity` number of arguments. The result is saved in a temporary file MapleOut.
`void TTStoMaple_Save (FILE *outfptr)`	Copies the result from the temporary file, MapleOut to the file pointed by `outfptr`.
`void TTStoMaple_readFile (char *fname)`	Reads a file `fname` into Maple.
`void TTStoMaple_sendLine (Maple_line buf)`	Sends a line object `buf` to Maple and the result is saved in a temporary file named MapleOut.
`void TTStoMaple_cleanup(void)`	Removes all the intermediate files. It also closes the UNIX pipe.

TABLE 2 - Access Programs for TTStoMaple

3.3.2.3 SAST_Maple Link (SAST_MapleLink.c)

The SAST_Maple module hides the low-level interface to Maple. The interface is through temporary files. The access programs are responsible for establishing and closing the pipe based interface, creating a file necessary for initializing the Maple session, writing a character string in the Maple session, and reading an expression from a file and ensuring that it has been properly sent to Maple. This module performs the necessary work required by the TTS_Maple interface pertaining to

the link. Refer to TABLE 3 for access programs and their descriptions.

Access Programs	Description
`void SAST_Maple_LinkInit (void)`	Makes files for initialization of Maple session.
`void SAST_MapleLinkSetup (void)`	Sets up the link to Maple.
`void SAST_MapleLinkWriteChar (char *char_string)`	Writes a string `char_string` to the link.
`void SAST_MapleLink (char outputfilename[])`	Sends expression from `outputfilename` into the link and waits for a limited time to get the result.
`void SAST_LinkClose (void)`	Closes the link to Maple.

TABLE 3 - Access Programs for SAST_MapleLink

3.3.2.4 Maple Command Constructor (Maple_sentence.c)

The Maple command constructor module hides the construction of Maple commands used in SAST. It has access programs to save an expression into a file, read an expression from a file, call a procedure with `arity` number of arguments and to quit a Maple session. These commands are used by SAST for manipulation of the Maple session. Refer to TABLE 4 for access programs and their descriptions.

Access Programs	Description
`char *Maple_cmdSave (char *ename, char *OutFName)`	Constructs a Maple command to save expression `ename` in a file `OutFName`.
`char *Maple_cmdQuit (void)`	Constructs a command to quit Maple session.
`char *Maple_cmdRead (char *InFName)`	Constructs a command to read a file `InFName` into Maple.

Access Programs	Description
`Maple line Maple callProc (char *proc, int arity, ...)`	Constructs a command to call a procedure `proc` with `arity` number of arguments. The arguments are after `arity`.

TABLE 4 - Access Programs for Maple_sentence

3.3.2.5 SAST_Expression (SAST_expn.c)

The SAST_Expression module encapsulates the way SAST handles expressions. These expressions can be atoms, applications or tables. The TTS expression is changed to prefix form of expression (refer to section 4.2 for the reason to change TTS expressions to their prefix form). Based on given user-defined constraints the given expression is specialized and simplified. The TTS expression (in the form of a syntax tree) is traversed following the depth-first traversal with the node first and then the left child until the innermost sub-expression is reached. It is then backed up to the nearest ancestor with right child. This continues until the root of the syntax tree is reached. A line buffer object allows easy construction of sub-expressions. In prefixed form, specialized/simplified tables are replaced by a temporary name provided by the Table module. Refer to TABLE 5 for the access programs and its description.

Access Programs	Description
`Maple_line THtoLinear (Expn expn, Path p)`	Changes expression `expn` at path `p` in TTS form to prefix form for sending to Maple. The returned line object has the resultant expression.

TABLE 5 - Access Program for SAST_expn

3.3.2.6 Constant (Maple_const.c) module

The constant module hides the way the tool handles constant expressions. This module retrieves the information stored in the symbol table that is associated with the expression. This information is then used to construct the prefixed expression. Refer to TABLE 6 for access programs and their descriptions.

Access Programs	Description
`void Maple_constInit (void)`	Initializes this module to a known state
`char *Maple_constGetName (Id id)`	Retrieves the information about the `id` of the constant from the symbol table

TABLE 6 - Access Programs for Constant module

3.3.2.7 Variable (Maple_var.c) module

The variable module hides the method in which SAST handles variables in the expression. This module returns the information about the identifier from the symbol table to be used in the construction of the prefixed expression. Refer to TABLE 7 for access programs and their descriptions.

Access Programs	Description
`void Maple_varInit (void)`	Initializes this module to a known state
`char *Maple_varGetName (Id id)`	Retrieves information for a variable `id` from the symbol table

TABLE 7 - Access Programs for Variable module

3.3.2.8 Application (Maple_applic.c) module

The application module encapsulates the way in which the tool deals with function applications. SAST handles function application expressions as strings, each consisting of a function name and arguments. Arguments are enclosed by round brackets. SAST can handle both default and user defined function applications. The tool obtains information about the function definitions from the symbol table associated with the expression. Refer to TABLE 8 for access programs and their descriptions.

Access Programs	Description
`void Maple_applicInit(void)`	Initializes this module to a known state
`char *Maple_applicGetName (Id id)`	Retrieves the information from the symbol table related to `id`

TABLE 8 - Access Programs for Application module

3.3.2.9 Table (Maple_table.c) module

The table module hides the method in which SAST handles a table. The tool substitutes a temporary name in place of the table while the prefixed expression is sent for simplification. For example, refer to the third stage of tabular expressions in section 4.3.2. Refer to TABLE 9 for access programs and their descriptions.

Access Programs	Description
`void Maple_tableInit (void)`	Initializes this module to a known state
`char *Maple_tableGetName (Expn expn, Path path)`	Returns a table name for the tabular expression `expn` at path `path` and saves the necessary information required for later substitution of the tabular expression back in place of the name

TABLE 9 - Access Programs for Table module

3.3.2.10 Carve[6] (CarveIn.c, CarveOut.c) Module

The access programs in the carve module are utilized to remove the rows and the columns from the table based on the semantics. CarveIn and CarveOut are complementary in functionality, i.e., CarveIn results in an expression between the two columns or rows where as CarveOut results in the remaining expression, outside of the two column or rows. Refer to TABLE 10 for access programs and their descriptions.

Access Programs	Description
`Expn CarveOut (Expn expn, int dim, int pos1, int pos2)`	'Carves out' the table `expn` when given the dimension `dim` and two header points - `pos1`, `pos2`. The table returned is one

[6] Carve is a TTS facility written by Al Tyson.

Access Programs	Description
	obtained after the removal of all the cells between the two header points
`Expn CarveIn(expn expn, int dim, int pos1, int pos2)`	'Carves in' the table `expn` when given the dimension `dim` and two header points - `pos1`, `pos2`. The table returned is one in between the two header points

TABLE 10 - Access Programs for Carve module

f_irr_band = **H_2**

$\mathbf{p_T}$: $H_1 \wedge G$ $\mathbf{r_T}$: H_2	low_irrational	rational	high_irrational
s_irr_band = high_irrational	m_sig ≤ 100	100 < m_sig ∧ m_sig ≤ 4900 - db	4900 - db ≤ m_sig

H_1 **G**

FIGURE 14 - Inverted Function Table illustrating CarveOut

For the inverted function table example in FIGURE 8, a following invocation of CarveOut would result in a table shown in FIGURE 14:

CarveOut(tab_expn, 1, 2, 3)

where, tab_expn is the tabular expression in FIGURE 8.

f_irr_band = **H_2**

$\mathbf{p_T}$: $H_1 \wedge G$ $\mathbf{r_T}$: H_2	low_irrational	rational
s_irr_band = high_irrational	m_sig ≤ 100	100 < m_sig ∧ m_sig ≤ 4900 - db
s_irr_band = low_irrational	m_sig < 100 + db	100 + db ≤ m_sig ∧ m_sig < 4900
s_irr_band =	m_sig ≤ 100	100 < m_sig ∧ m_sig <

$\mathbf{p_T}$: $H_1 \wedge G$ $\mathbf{r_T}$: H_2	low_irrational	rational
rational		4900

H_1 **G**

FIGURE 15 - Inverted Function Table illustrating CarveIn

For the same example in FIGURE 8, a following invocation of CarveIn would result in a table shown in FIGURE 15:

CarveIn(tab_expn, 2, 1, 2)

where, tab_expn is the tabular expression in FIGURE 8.

3.3.2.11 Maple Code Generation (Maple_code.c) Module

The Maple code generation module encapsulates the syntax in which the information from the Table, Application, Variable and Constant modules is arranged to form an intermediate form of prefixed expression. The access program to code variables encloses the variables in quotes before sending them through the interface to Maple (Refer to Atoms in Chapter 4 for further explanation). Refer to TABLE 11 for access programs and their descriptions.

Access Programs	Description
`char *Maple_linecodeConst (Id id, char *buf)`	Returns code information for the constant `id`.
`char *Maple_linecodeVar (Id id, char *buf)`	Codes the variable `id`.
`Maple_line *Maple_linecodeApplic (Id id, int arity, Maple_line args[])`	Codes the application `id` with `arity` number of arguments in `args` (in form of Maple_line object). Returns a Maple_line object containing the code.

Access Programs	Description
`Maple_line Maple_codeTable(Expn e, Path p, Maple_line buf)`	Codes the table expression `e` at path `p`, i.e. substitutes a temporary name for the table in the Maple_line object `buf` and returns a Maple_line object for the table code.

TABLE 11 - Access Program for Maple Code Generation

3.3.2.12 SAST Output (SAST_output.c)

The SAST output module handles the syntax that is applied to the intermediate expression. The access program appends the name of the expression "SAST_expn" and the assignment symbol ":=" before the intermediate form of prefixed expression and the Maple command delimiter, semi-colon, at the end. This prepares the final form of the expression to be saved in a file to be read by Maple at a later stage. Refer to TABLE 12 for the access program and its description.

Access Programs	Description
`void SAST_outInit (char *outname, Maple_line buf)`	Saves the expression from line object `buf` in file named `outname`.

TABLE 12 - Access Program for SAST Output Module

3.3.2.13 SAST Utility (SAST_Utils.c)

The SAST utility module hides general utilities needed by SAST. The access programs of this module include facilities to copy one file to another and to put back quotes around a string. Refer to TABLE 13 for access programs and their descriptions.

Access Programs	Description
`char *codeTranslateName (char *name)`	Encloses a character string `name` in backquotes. For example: ‘var‘
`void fileCopy(FILE *fromfptr, FILE *tofptr)`	Copies the contents of file `fromfptr` to `tofptr`.

TABLE 13 - Access Programs for SAST Utility

3.3.2.14 Maple-Simplify Module (CasEnd.ms)

This module hides the computational part of the SAST that is handled by Maple. It handles the simplification of expressions based on given user constraints. The access programs help to simplify expressions and TABELT[7]; setting constraints; handling Equal, handling existential and universal quantifiers (within the scope of this work); and declaring constants.

The Maple module is written in the Maple programming language. Some of the design issues involved in the development of this module are dealt in section 4.7.

Some possible changes include extending the capabilities of SAST by incorporating more procedures to handle simplifications for expressions types not currently dealt by SAST. These are described as future work in Chapter 6.

A notation to write Maple program is necessary as Maple does not have exactly the same procedure structure as the C language. Following is the notation used to describe access programs in Maple:

Return_Value Procedure_Name (Input_parameters)

where *Return_Value* is given by either:

- `Void` - to indicate that nothing is returned by the procedure, or
- `Result` - to indicate that a result is returned by the procedure

Refer to TABLE 14 for access programs and their descriptions

Access Programs	Description
`Result CasEndProc (input)`	Handles simplification for SAST in Maple. Takes in `input` (a prefixed expression) and returns a result (an expression in Maple syntax, after simplification) based on global user defined constraints.
`Void Set_Gbl_Constrnt (expr)`	Sets the global constraints using `expr`
`Result TABELT (cond, expr)`	Handles simplification of an expression `expr` based on conditions `cond` and returns a simplified expression. This simplification does not consider the global user defined constraints.
`Void DeclareConst (expr)`	Declares `expr` as a constant.
`Void ThereExists(var, expr)`	Handles existential quantifier in an expression `expr` with `var` as the index variable of the quantification i.e. `var` is bound in `expr`.

[7] *TABELT* is an expression of the form (condition expression, value expression). For further details about its format and usage, refer to section 4.3.2.

Access Programs	Description
`Void ForAll(var, expr)`	Handles the universal quantifier in an expression `expr` with `var` as the index variable of the quantification.
`Result Equal (expr1, expr2)`	Checks if `expr1` is equal to `expr2` and returns true if they are equal, otherwise false.

TABLE 14 - Access Programs for Maple Module

3.3.2.15 SAST_Context Module (SAST_CM.c)

A context is used to store expression, the identifiers and related information. The SAST_Context module hides those functions related to the context manager that are needed by SAST. The access programs in this module deal with saving and accessing the context from a file, retrieving the associated symbol table, and accessing expressions in the context. Refer to TABLE 15 for access programs and their descriptions.

Access Programs	Description
`void SAST_CRegInit (void)`	Initializes the module to a known state.
`void SAST_CRegSetContext (CHandle context)`	Sets the `context`.
`CHandle SASTCRegGetContext (void)`	Returns the context.
`Expn SAST_CRegGetExpnByposition (CHandle ch, int pos)`	Returns the expression from the context `ch` at position `pos`.
`Expn SASTCRegGetExpnByName (CHandle ch, char expnName[])`	Returns expression from context `ch` with name `expnName`.
`int SAST_CRegGetNumExpn (CHandle ch)`	Returns number of expressions in the context `ch`.
`void SAST_CRegAddExpn(CHandle ch, Expn expn, char expnName[])`	Adds an expression `expn` with name `expnName` to the end of the context `ch`.
`void SAST_CRegRemoveExpn (CHandle ch, Expn expn)`	Removes an expression `expn` from the context `ch`.
`void SAST_CRegSetSymTbl (CHandle ch)`	Sets symbol table in the context `ch`.
`SymTbl SAST_CRegGetSymTbl`	Returns the symbol table from the context `ch`.

Access Programs	Description
`(Chandle ch)`	
`void SAST_SetContextInfo(char contextfilename[])`	Sets the context and information needed for the SAST from the file `contextfilename`.
`void SAST_CFileSave (CHandle ch, char contextfilename[])`	Saves the context `ch` in the file `contextfilename`.
`CHandle SAST_CFileLoad(char contextfilename[])`	Loads the file with name `contextfilename`.

TABLE 15 - Access programs for SAST_Context Module

3.3.2.16 SAST_Info Module (SAST_info.c)

Information about the identifiers in the expression is stored in the symbol table. The information is grouped into classes, with different tools accessing information from different classes. SAST requires information in MapleName and MapleDefn along with other default classes necessary for interpreting expressions. This module includes those parts of The information module that deal with accessing the symbol table and the information stored in it. SAST_Info module also includes a part from GTS module required by SAST to obtain table semantics information from the symbol table. Refer to TABLE 16 for access programs and their descriptions.

Access Programs	Description
`void SAST_infoInit (void)`	Initializes the module to a known state.
`void SAST_infoSetSymTable (SymTbl table)`	Sets the symbol table `table`.
`SymTbl SAST_infoGetSymTable (void)`	Returns the symbol table that is set.
`InfoPtr SAST_infoGetSymData (Id id, Name classname)`	Returns the information from the symbol table in a given `class` for a given `id`.

Access Programs	Description
`void SAST_InfoAssign (SymTbl infoTab)`	Assign all the information needed to run SAST in symbol table `infoTab`. Information is stored into MapleName and MapleDefn classes in the cases where it is missing.
`GTS_Sem SAST_GTSinfoGetTabSem (SymTbl infoTab, Id id)`	Returns table semantics from symbol table `infoTab` for the `id`.

TABLE 16 - Access Programs for SAST_Info Module

3.3.2.17 Semantics Module (SAST_GTS.c)

SAST needs semantic information in order to interpret tabular expressions as described in [1]. This semantics module hides access programs from the Table semantics and Syntax checking sub-modules of the Generalized Table Semantics (GTS) module as related to SAST. This module is needed by the tool for following purposes:

- Check the shape of the table
- Make TABELT and store the result of TABELT in a value cell based on the table rules
- Carve out the rows and columns from the table based on the Cell Connection Graph (CCG) of the table

Refer to TABLE 17 for access programs and their descriptions.

Access Programs	Description
`void SAST_GTSInit (void)`	Initializes the module to a known state.
`Expn SAST_GTSsemGetPredRule (GTS_Sem s)`	Returns the predicate rule expression from the table semantics `s`.
`Expn SAST_GTSsemGetRelRule (GTS_Sem s)`	Returns the relation rule from the table semantics `s`.
`bool SAST_GTSsemIsValueGrid (GTS_Sem s, int g)`	Checks the table semantics `s` for a grid number `g` being a value grid. BOOL_TRUE if yes, else BOOL_FALSE.
`bool SAST_GTSsemIsGuardGrid (GTS_Sem s, int g)`	Checks the table semantics `s` for a grid `g` being a guard grid. BOOL_TRUE if yes, else BOOL_FALSE.
`bool SAST_GTScheckShape(Expn expn, Path path)`	Checks if the expression `expn` at Path `path` is a well-formed table. BOOL_TRUE if yes

Access Programs	Description
	else BOOL_FALSE.
`GTS_CCG SAST_GTSsemGetCCG (GTS_Sem sem)`	Returns the CCG information from the semantics `sem`.
`GTS_Sem SAST_tableGetSem (Id id)`	Returns the semantics of a table with Id `id`.
`void SAST_GTSinfoSetTable (SymTbl infoTab)`	Sets the symbol table, `infoTab` for the GTS module.

TABLE 17 - Access Programs for Semantics Module

3.3.2.18 Maple_TTS Interface (SAST_MapletoTTS.c)

The Maple_TTS Interface module hides the parser which parses the result from Maple to a TTS compatible form. This module hides any changes to the parser module's access programs and their arguments. The access programs perform module initialization and parse a Maple expression to a form compatible to TTS. A detailed description of parsing a Maple expression is given in [25]. The parser uses the symbol table associated with the input expression to parse the resultant expression from Maple. Refer to TABLE 18 for access programs and their descriptions.

Access Programs	Description
`void SAST_MapletoTTSinit(void)`	Initializes the module to a known state.
`Expn SAST_MapletoTTS(SymTbl table, FILE *file)`	Parses the expression in infix form in file named `file` using symbols from symbol table, `table`.

TABLE 18 - Access Programs for Maple_TTS Module

3.3.2.19 Status Reporting Module (SAST_error.c)

The status reporting module for SAST encapsulates the result of invoking access programs of various SAST modules. An invocation is successful if the value of SAST_Token is SuccessSAST, otherwise an error token reflecting the error is set. Refer to TABLE 19 for access programs and their descriptions. TABLE 20 contains a list of all the error tokens used in the SAST along with their descriptions. This module includes the access programs for setting and retrieving the error token, and to get a corresponding description for an error token. These

access programs are useful to ensure correct and error free execution of SAST.

Access Programs	Description
`void SAST_errSet(SAST_Token T)`	Sets the status token to `T`.
`void char * SAST_errGetStr(SAST_Token T)`	Returns the string corresponding to the status token `T`.
`SAST_Token SAST_errGet()`	Gets the status token.

TABLE 19 - Access Programs for Status Reporting

Status Token	Interpretation
`SuccessSAST`	No error.
`THErr`	Error in Table Holder module.
`AllocFailSASTErr`	Memory allocation failure.
`InvalidHandleSASTErr`	Current object is not valid.
`TableOverflowSASTErr`	More objects of the given type cannot be created.
`InfoTabSASTErr`	Symbol table is not set.
`InfoGetSASTErr`	Error in getting the information for an id from symbol table.
`IdListErr`	Error in IdList module.
`MapleNotFoundErr`	Maple is not present on the same host machine.
`SASTLinkCloseErr`	Link to Maple could not be closed.
`SASTMapleLinkSetUpErr`	Link to Maple could not be set.
`SASTLinkWriteErr`	Character string could not be written into the link.
`LinearizationErr`	Error in changing the TTS expression into a prefixed form.
`CRegSymTblErr`	Symbol table could not be set or retrieved from the context.
`CRegGetSASTErr`	Context undefined.
`CRegExpnSASTErr`	Expression from the context is not accessible.
`CarveErr`	Error in Carve module.
`MapleCmdErr`	Error in Maple_sentence module.
`InvalidRuleExpn`	Invalid table rule expression.
`SASTGTSErr`	Error in GTS module.
`SASTGTSNoSemErr`	Semantics is not set.

Status Token	Interpretation
`MapleSaveErr`	Could not save the result from Maple into a file.
`NotInitialized`	Link is not initialized.
`NullPtrErr`	Null pointer.
`NotValidTag`	Tag is not valid.
`ParseErr`	Error in Parser module.
`CFileErr`	Error in loading or saving the context file.
`UnknownSASTErr`	Unknown type of error.

TABLE 20 - SAST Error Tokens

3.3.2.20 SAST_Line Buffer Module (Maple_line.c)

The SAST_Line buffer module is adapted from TOG's (Test Oracle Generator) TOG_line module to handle the assembling of Maple commands and prefixed expressions as line objects. Refer to [19] for details of this module. This module provides access programs to handle joining strings of arbitrary length. Refer to TABLE 21 for access programs and their descriptions.

Access Programs	Description
`void Maple_lineInit(void)`	Initializes the module to a known state.
`void Maple_line Maple_lineAppend(Maple_line l, char *text)`	Appends a character string `text` at the end of the line object `l`.
`Maple_line MAple_lineJoin(Maple_line L1, Maple_Line L2)`	Joins two line objects `L1` and `L2`.
`char *Maple_lineGetText(Maple_line line, char *separator)`	Returns the character buffer from the Maple_line object `line` delimited by `separator`.

Access Programs	Description
`Maple_line Maple_lineInsert(Maple_line line, char *text)`	Inserts a character string `text` in the beginning of the Maple_line object, `line`.
`void Maple_lineDestroy(Maple_line line)`	Destroys the Maple_line object, `line`.

TABLE 21 - Access Programs for SAST_Line Buffer Module

3.3.2.21 Table Holder Module (libtblhold.a)

The Table Holder module is responsible for storing expressions structure. Constraints and the expression to be simplified are stored and traversed using the Table Holder module which forms part of the TTS kernel. This is described in detail in [24].

3.3.2.22 Information Module (libinfo.a)

The Info module stores the information about various identifiers used in the expression. Information about the identifier's MapleName and MapleDefn is stored and retrieved using this module. A detailed description is given in [24].

3.3.2.23 Kernel Utilities (libkernelutil.a)

Kernel utilities use the TTS kernel. It includes GTS module responsible for storing and retrieval of information for interpretation of tables. Some general utilities like IdList that manipulate list of identifiers are also included in kernel utilities as described in detail in [24].

3.3.2.24 Context Manager Module (libcm.a)

The Context Manager module allows grouping of expressions in a context. The input for SAST is a context file (containing constraints and the expression to be specialized/simplified) that is created using the Context Manager as described in [24]. SAST uses access programs from Context Manager module to manipulate the context and to temporarily store the table name along with the table expression. This temporary table name is used for substituting back the table into the expression in place of the table name.

3.3.2.25 Parser Module (libparse.a)

The parser module conceals the parsing of an infix expression (Maple results)

back into TTS compatible format for storing in TH. This is described in [25]. This module contains an access program to parse an expression that is Maple compatible to a TTS compatible expression. This module also includes error status reporting for the parser.

3.4 Algorithm Overview

SAST uses the following algorithm to specialize and simplify expressions:

1. Initialize the Maple session with information from the symbol table.
2. Read constraint expressions from the context file and set them in the Maple session.
3. Read the expression on which the constraints have to be applied and send to Maple.
4. For scalar expressions, the result is obtained directly. The result from Maple is parsed back into a TTS compatible form. Following are the steps for simplification of a tabular expression:
    a) Send expressions from each of the cells to Maple and parse the Maple result into the same cell.
    b) Remove rows and/or columns based on the semantics (CCG) of the tabular expression.
    c) Send TABELT expressions from the table based on the table rules to Maple and parse the result from Maple into the value cell as defined by table rules.
    d) Replace the table expression placeholder with a temporary name and store the tabular expression corresponding to this name in the input context file.
    e) Send the expression that is to be simplified and specialized to Maple.
    f) Parse back the result into TTS form and replace the temporary name with the simplified tabular expression.
5. The expression after simplification and specialization is appended to the context file.
6. Perform cleanup.

Chapter 4

4. Design Issues of Specialization and Simplification Tool (SAST)

Chapter 4 covers some design issues encountered during the development of SAST with an example illustrating the effect of each design decision. Some issues include the format of input for SAST, intermediate forms of expressions, the method of handling different types of TTS expressions, how and when to remove rows and/or columns, the interface between SAST and Maple, and simplification procedures in Maple.

4.1 Allowed Constraints

The design of the Specialization and Simplification tool does not enforce any limitations on the type of scalar constraints. At present, SAST can handle constraints which assign numeric or symbolic values to variables. A symbolic value is defined as one that has a tag type of ConstTag in TTS. This kind of specialization by assignment leads to the simplification of expressions. For example: s_irr_band = low_irrational; db = 15. The tool design does not allow tabular constraints because they have to evaluated before being used.

4.2 Reason for converting TTS expressions to their Prefix Form

Expressions are stored in TTS as a syntax tree. Maple handles expression in infix form. For SAST, the expressions are changed from TTS compatible form to

prefixed form because it provides a flexible ASCII[8] interface with any other mathematical software. Two information classes, namely MapleName and MapleDefn, have been created to store Maple related information for the identifiers. MapleName class stores names of all commonly used identifiers. These names are used to form prefixed expressions for use in Maple. TABLE 22 shows the default symbols and their Maple related information. Symbols with "N/A" entry in the column for MapleDefn in TABLE 22 have a Maple procedure by the same name to perform desired actions in SAST. This helps in automating an action required by the tool but not provided by Maple. The default action that is available may not be the desired one for SAST. A flexible interface and ability to modify the default actions of Maple were the reasons to choose the prefix form of expression rather than the infix form.

Expressions in MapleDefn class use an arrow notation (->) for functional operators in Maple. A functional operator in Maple is a special form of a procedure. The arrow notation is:

`(<vars>) -> <result>`

where,

`<vars>` is a sequence of variable names (or a single variable)

`<result>` is the result of the procedure acting on `<vars>`

For example: `x -> x^2` is a function that gives square of its argument.

The syntax of expression in MapleDefn column of TABLE 22 and MapleDefn class can be of either Form 1 or Form 2:

Form 1:

Procedure_Name := (Variables) -> (Proc_Result);

where,

Procedure_Name is the name of the procedure

Variables is a sequence of arguments or a single argument

Proc_Result is the result of procedure *Procedure_Name* with arguments *Variables*

:= is an assignment operator

-> is a functional operator

---

[8] American Standard Code for Information Interchange

; is an expression delimiter

Form 2:

Variable := Value;

where,

Variable is the name of the variable to be assigned

Value is the constant value that is assigned to the variable

:= and ; are same as above

Symbol	MapleName	MapleDefn
+	Plus	Plus := (x, y) -> (x + y);
-	Minus	Minus := (x, y) -> (x - y);
*	Mult	Mult := (x, y) -> (x * y);
/	Div	Div := (x, y) -> (x / y);
!	Factorial	Factorial := (x) -> x!;
<	Less	Less := (x, y) -> (x < y);
>	Greater	Greater := (x, y) -> (x > y);
^	Expo	Expo := (x, y) -> (x^y);
=	Equal	N/A
∧	And	And := (x, y) -> (x and y);
∨	Or	Or := (x, y) -> (x or y);
¬	Not	Not := (x) -> not (x);
∀	ForAll	N/A
∃	ThereExists	N/A
≠	NotEqual	NotEqual := (x, y) -> (x <> y);
≤	LessEqual	LessEqual := (x, y) -> (x <= y);
≥	GreaterEqual	GreaterEqual := (x, y) -> (x >= y);
true	Pred_true	Pred_true := true;
false	Pred_false	Pred_false := false;

TABLE 22 - Identifier related Maple Information

4.3 Handling of TTS Expressions

SAST works on a copy of the expression that is to be simplified. This is particularly useful as a common application of SAST would be to apply different combinations of constraints to the same table, resulting in different specialized tables but keeping the original expression unchanged. SAST handles scalar and tabular expressions in different ways as follows:

4.3.1 Scalar Expressions

Scalar expressions are non-tabular expressions that include atoms and applications. Refer to Chapter 6 for an illustration of execution of a scalar expression by SAST.

4.3.1.1 Atoms

Atoms are numerical and symbolic constants, predicate constants, or variables. Example: 123; MAX; ***false***; x. Atoms are terms as defined in section 2.1. All the variables are enclosed in back quotes by SAST to differentiate between a string of characters which can be used as a number or as a variable. Maple uses back quotes to declare a string. For example: `s_irr_band`. The atoms can be easily identified as TTS stores them with a unique identifier and the associated tag. TABLE 23 illustrates all different types of atoms with their TTS tags, examples and their Maple representation.

4.3.1.2 Function/Relation Applications

Applications are relations/functions with a name and arguments. TTS stores an identifier, tag type, arity[9] and array of expressions that are arguments for an application. Arguments can be atoms, applications or tables. An application can either be a function application, predicate expression, logical expression or quantified logical expression. Some examples of applications with their Maple representations are shown in TABLE 23.

Expression Type	TTS Tag	Example	Maple prefix form
Constant, Symbolic Constant	ConstTag	123, MAX	123, MAX
Predicate Constant	PConstTag	***true***	Pred_true
Variable	VarTag	s_irr_band	‘s_irr_band‘

[9] Number of arguments

Expression Type	TTS Tag	Example	Maple prefix form
Function Application	FATag	s_irr_band + MAX	Plus(‘s_irr_band‘, MAX)
Predicate Expression	PETag	x = 3	Equal(‘x‘,3)
Logical Expression	LETag	x = 3 ∧ y > 5	And(Equal(‘x‘,3), Greater(‘y‘,5))
Logical Expression with Quantifiers	QLETag	(∀ x, (x < 5))	ForAll(‘x‘, Greater(‘x‘,5))
Function Table	FTableTag	See Chapter 6	See Chapter 6

TABLE 23 - Different Types of Expressions

4.3.2 Tabular Expressions

Even though Maple has a data structure for representation of tables, SAST uses a generalized approach of retaining the table format and sending only sub-expressions. Using Maple’s table structure would require development of already available table handling capabilities of TTS in Maple, resulting in redundant work. As the tabular structure is retained, following are the three stages for handling tabular expressions within SAST (refer to Chapter 6 for an example of execution of a tabular expression by SAST):

First Stage (Cell Stage):

Expressions from all the cells in headers and main grid are sent to Maple.

Second Stage (TABELT stage):

Combinations of cells are selected according to the semantics of the table, and are sent in the following format:

TABELT(GuardExpn, ValueExpn)

where *GuardExpn* is the logical combination of expressions from the cells as given by the table predicate rule and *ValueExpn* is the logical combination of cell expressions as given by the table relation rule.

For Example: Consider the raw element as shown in FIGURE 9 of Chapter 2 which will be sent to Maple utilizing the TABELT format as follows:

```
TABELT(And(Equal(s_irr_band, high_irrational),
GreaterEqual(m_sig, 100)), low_irrational)
```

Third Stage (Table stage):

Since the table format is to be retained in SAST, a temporary name is assigned to the table using function *tempnam* from standard I/O package (libc.a). A new identifier is created for storing this temporary name in the Name class and ConstTag in the Tag class of the symbol table. The tabular expression corresponding to this name is stored in the context file with the same name. This is used to substitute back the tabular expression for the name before saving the result in the context file. For example: Consider the tabular expression in FIGURE 11, the expression that will be sent to Maple in this stage will be:

```
Equal(`f_irr_band`, sast_SAST_AAAaajfaq);
```

4.4 Removal of Rows/Columns

The rows and/or columns are removed from the table based on the semantics of the table. This removal of cells from a tabular expression is due to that fact that under a given set of constraints these expressions are evaluated to *false* by Maple. The rows and/or columns are removed between the cell stage and TABELT stage. "*false"* means the condition is no longer valid under the given conditions in the given domain and hence it should not remain in the specialized table. Specialized table is the one that describes the behaviour of the program in the restricted domain.

Removal of rows and columns is done by using "carve" facilities (described in Chapter 3) that are available in TTS. Refer to the example of FIGURE 8 in section 2.3 which specializes to FIGURE 16, under the constraints: s_irr_band = low_irrational; db = 15; m_signal = 100

FIGURE 16 is an inverted function table showing an intermediate stage in

specialization and the table rules are given by p_T and r_T. Since all the cell expressions of the second and the third columns of main grid G are *false*, they can be removed because conjunction of these main grid cell expressions with cell expressions in header H_1 will yield *false* and the function will never be evaluated to rational or high_irrational. Also the first and the third cell expressions ($\underline{h_1^1, h_3^1}$) in header H_1 are *false* and can be removed, resulting in FIGURE 17.

f_irr_band = **H_2**

p_T: $H_1 \wedge G$ **r_T**: H_2	low_irrational	rational	high_irrational
false	true	false	false
true	true	false	false
false	true	false	false

H_1 **G**

FIGURE 16 - Intermediate Stage in Specialization of FIGURE 8

f_irr_band = **H_2**

p_T: $H_1 \wedge G$ **r_T**: H_2	low_irrational
true	true

H_1 **G**

FIGURE 17 - Final Stage in Specialization of FIGURE 8

The removal of rows and/or columns is based on the semantics information associated with the table and can be classified into three categories as follows:

1. Normal Table: In case of Normal tables, removal of rows and/or columns (including the corresponding rows/columns in the main grid) is possible when cell expressions in the header of a tabular expression are *false*.

2. Inverted Table: In Inverted tables, if the cell expression in the guard header is *false*, then that row/column can be removed, including the main grid cells. Also, if the cell expressions in an entire row/column of the main grid are *false* then it can be removed.

3. Vector Table: In Vector tables, if the cell expression in the guard grid is *false* then the entire row/column can be removed, including that row/column in the main grid.

4.5 Interface between SAST and Maple

As described in section 3.2, a UNIX pipe is used to establish an interface between SAST and Maple. It supports unidirectional[10] communication between the calling SAST program and the Maple session executed through a shell command.

The expression from SAST is sent to Maple through a temporary file named MapleIn. This file a contains prefixed form of the expression in a Maple compatible syntax. The result from Maple is saved in another temporary file named MapleOut which is then read by the TTS_Maple interface. The Maple_TTS interface encompasses the parser module that parses the Maple result and produces a TTS compatible equivalent form. Various transformations of the expression are shown in FIGURE 18. To ensure reliable communication, expressions are sent one by one and the tool continuously checks the presence of a temporary file named MapleOut containing the result. SAST waits for a sufficiently long time[11] to obtain the result, failing which it set an error token, MapleSaveErr, indicating that SAST is unable to get result from Maple.

4.6 Initialization of Maple Session

The Maple session is initialized as follows:

1. Definitions of all default and user defined functions: These definitions are extracted from the symbol table corresponding to the expression. The definitions are saved into a file named SASTinit which is then read

---

[10] The link is *unidirectional* if only one process connected to link can either send or receive, but not both, and each link has at least one receive process connected to it. In this thesis there is only one calling process that is the sender of messages and the invoked Maple session is the receiver of messages.

[11] Time taken by the machine on which SAST is running, to increment a counter to 65000.

by Maple. These definitions are in Maple syntax. For example:

```
Mult := (x,y) -> (x*y);
Or := (x,y) -> (x or y);
```

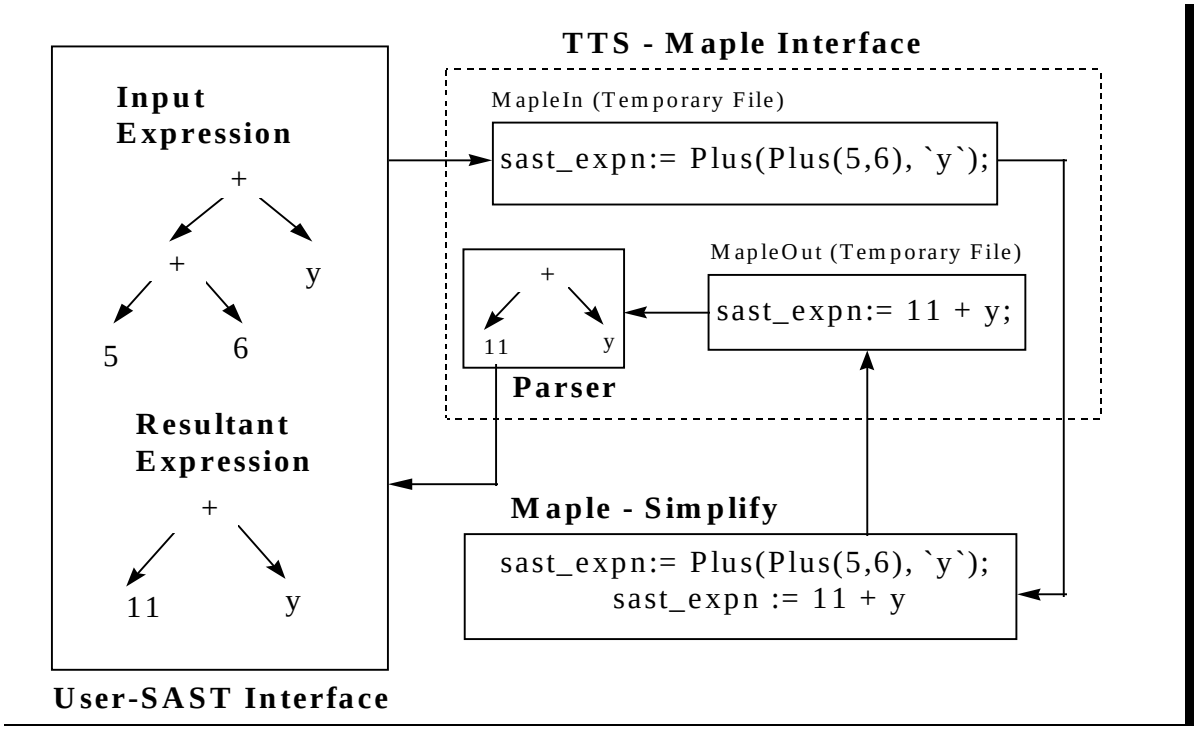

FIGURE 18 - SAST Expression Interface to Maple

2. Numeric and symbolic constants: Both numeric and symbolic constants are declared. They are identified by the type of tag (ConstTag). SAST allows substitution of a variable by either a numeric or symbolic value. In this work, substitution of a variable by another variable is not allowed since there may be situations where free variables in an expression with quantifiers can become bounded after substitution. In this work SAST deals only with substitution of free variables by constants.

3. Maple procedures: Procedures of Maple-Simplify module are written in Maple programming language. These procedures (prototypes) are to be defined at the time of initialization of the Maple session so that they can be invoked at a later stage.

4.7 Procedures in Maple

The computations in SAST include algebraic and logical simplifications of expressions that are then sent to Maple through the interface. The Maple session remains clean as no values are actually assigned to the variables and thus it remains in its initial state. Procedures mostly use the default functionality of Maple and the only modified procedure is Equal. Only substitution of free variables by a constant or symbolic constant is allowed in a quantified expression. Following is a brief description of some Maple procedures used for performing specialization/ simplification:

Equal

The procedure *Equal* illustrates the benefit of permitting names different than the default ones because it allows the incorporation of the user desired functionality. This procedure is responsible for the following evaluations:

- Equality of numerical values and combinations of numerical and symbolic constants
- Assignment of symbolic or numerical constant value to a variable.

This procedure helps in avoiding assignment of a variable by another variable. The need for this procedure is shown in Example 2 of section 1.5.6 where an expression in Maple evaluates to *false* when no values are assigned to the variables.

Symbolic constants are declared in Maple using a Maple procedure, DeclareConst and tag information from symbol table. The procedure, Equal uses this information during the evaluation.

Set_Gbl_Constrnt

Set_Gbl_Constrnt handles user defined global constraints. It gives logical formula modeling of the constraints. Since the ‘specialization by assignment’ technique is used for the current functionality of the tool, only assignment of variables to a numeric or symbolic constants is handled. All the assignments are stored in a global list which is then used by other procedures to perform simplifications by simultaneous substitution. An example is as follows:

```
Set_Gbl_Constrnt(And(Equal(x,2), Equal(y,MAX)))
Set_Gbl_Constrnt(Equal(MAX,10))
```

For above example the global list stores only [x = 2, y = MAX, MAX = 10]. MAX is already declared to be a constant during initialization of Maple session.

TABELT

TABELT simplifies a raw element of a table using its semantic rules and elements from each of the table components as described in sections 2.3. The procedure takes into account conditions in the guard expression and performs the substitution in the value expression. This procedure does the final simplification of the value expressions based on the guard expression.

CasEndProc

SAST invokes CasEndProc for performing all simplifications based on the already defined constraints. It is a control module that calls other modules in Maple to facilitate specialization and simplification of an expression.

Rename/ThereExists/ForAll

The Maple procedure Rename simplifies expressions with quantifiers ThereExists and ForAll. Rename gives a temporary name to the bound variables in the expression with quantifiers so that only the free variables in the expression get substituted. The procedures ThereExists and ForAll call the procedure Rename to assign a temporary name to the bound variable in the expression with quantifiers.

4.8 Maple Output

The result from Maple is parsed back into TTS using the SAST_MapletoTTS, an access program of the Maple_TTS interface. The MapleToTts function of SAST_MapletoTTS module is in C (language) created using Flex[12] and Yacc[13]. It uses the symbol table to get information about the symbols in its input expression (in infix form) and converts it to a TTS compatible expression. The SAST_MapleToTTS function hides the parser that searches for a character string in the Name and MapleName classes of an identifier in the symbol table. The parser can create new variables and numerical values.

In the case of tabular expressions, the intermediate results of each stage are stored as follows:

---

[12] Flex is scanner generator that accepts regular expressions and produces a table driven recognizer.

[13] Yacc is a parser generator from an input consisting of a context-free grammar specification

First Stage:

The result is stored back in the cells from where the expression was sent to Maple

Second Stage:

The result is stored in value cell of a TABELT.

Third Stage:

The table expression is substituted for the temporary table name. The result after substitution is saved in the input context file.

4.9 Clean up Procedure

Clean-up procedure involves removal of all the temporary files and closing of the UNIX pipe. It also involves removal of the temporary storage for tabular expressions corresponding to a temporary name, and its identifier in the context file.

Chapter 5

5. Limitations

Like any other practical tool or technique, this specialization and simplification tool also some has limitations. These limitations are due to the following reasons:

- Some limitations are due to Maple, that provides computational support to SAST and is responsible for its simplification capabilities.
- Other modules used in SAST but not developed as a part of this work.
- Design decisions for SAST.

5.1 Limitations due to Maple

- The result returned by Maple is used in SAST. Maple has automatic simplifications which could lead to undesired results (as described in section 1.5.6). Further, SAST cannot verify the correctness and appropriateness of the results returned by Maple.
- Since Maple was developed in 1980s, its algorithms and certain design decisions are affected by the restrictions of the computer technology available at that time.
- Like any other software, Maple may not be error free.
- As CAS works best when guided properly by the user, it becomes necessary that the user has some knowledge of mathematics to achieve, comprehend, and analyze the results.

- Maple does not handle recursive definitions and issues a warning in such situations.
- The results returned by Maple are based on its default complex domain unless specified. For example: Result of $\sqrt{-3}$ is $I3^{1/2}$ in Maple. This example illustrates that for square root of a negative number, Maple assumes a complex domain unless otherwise specified.

5.2 Limitations due to other support modules

5.2.1 Generalized Table Semantics

SAST uses Generalized Table Semantics (GTS) module as described in [1] for semantics of tables used with in SERG. This module helps in removal of rows and/or columns of tables based on their interpretation. Some of the design decisions of GTS which impose limitations on usage of SAST are:

- Rules must be scalar expressions.
- Quantifiers in predicate rule expressions are not presently supported by GTS.
- Function applications are not allowed in table predicate rules.
- Rules should not contain any variables.

These limitations allow all the common types of tables.

5.2.2 Parser Module

SAST invokes access program from the parser module to parse the Maple result. The parser can only create new variables and numeric constants. This limitation of parser satisfies the current requirements of SAST.

5.3 Limitations of SAST

In addition to the limitations described above which are external to this work, the user should also have a clear understanding of the scope, assumptions, and design limitations of this tool itself in order to appreciate the results. In addition to

some of the limitations of SAST that are due to its design there are features that are not yet implemented and are described as future work in chapter 6.

- Tables should fit SERG's tabular model with its semantics interpreted by GTS module. Tables compatible to this model should have a main grid with one dimensional headers.
- Maple and SAST have to be on the same host machine, as SAST uses a unidirectional pipe to communicate with Maple.
- Design of procedures developed in Maple for SAST is limited to Maple V Release 4 version 4.00f.
- The definitions used to initialize the Maple session have to be in Maple syntax. User definitions and symbolic constants need to be declared at the time of initialization of the Maple session for appropriate results.
- Tables which can be handled by SAST are mainly Normal and Inverted function tables, and Predicate expression and Inverted predicate expression tables. Mixed vector tables should be changed to vector predicate tables with predicate expressions in their main grid cells. All types of tables follow their usual interpretation described in [1, 15]. Relation tables are dealt in a way that for a given set of assignments, SAST can evaluate the predicate. SAST does not deal with abbreviated grids.
- As per the scope of this work only scalar auxiliary predicates and functions are handled. Further more, the current work cannot handle tabular auxiliary definitions and specified domain.
- Tool cannot handle tabular constraints.
- SAST only handles simultaneous syntactic substitution represented by a term Equal(x,c), where c is either a numeric constant or a symbolic constant. This term is used in TABELT and for declaring global constraints. This tool can only handle Equal and conjunction of Equal terms but there are no design restrictions on the type of the constraints.
- In the case of substitution for expressions with quantifiers, only free variables can get substituted. SAST does not allow substitution of a variable by another variable.
- The number of dimensions in a table is always maintained in SAST. SAST will keep the table structure and will not evaluate it to a value even when all the variables are assigned values.

Chapter 6

6. Results and Future Work

This chapter discusses the results of SAST and some future work that is envisioned for SAST. Since this tool is in its early stages, there is a substantial scope for modifications and enhancements.

6.1 Results

SAST successfully applies the technique of specialization to expressions specifying the behaviour of a program, for the first time. The specialization by assignment approach has been found useful to simplify expressions for given values of variables describing the state of the program. These values can be constants or symbolic constants. SAST can handle any kind of expression for simplification. These expressions could be scalar or tabular or a combination of both. Tabular expressions can be one dimensional or multidimensional.

Some non-tangible results of this work are: it helped in establishing the utility of CAS for tables and showed utility of these software tools in manipulation of tables. A summary of execution steps for specialization and simplification of a scalar and a tabular expression is as follows:

Example 1: Scalar Expression

Input:

Constraints:

gauge = 3000

signal = 3000

Expression:

$(\forall db, db > signal) \wedge (gauge = 3000)$

Specialization and Simplification Steps:

All the expressions are converted to prefixed form before sending them to Maple.

1. Initialization of Maple session by default definitions, and declaration of constants and Maple procedures. During the initialization process, *signal* is declared as a constant as follows:

   DeclareConst(signal);

2. The user-defined constraints (in prefixed form) are set in the Maple session using the "Set_Gbl_Constrnt" procedure in Maple as:

   Set_Gbl_Constrnt(Equal(`gauge`, 3000));
   Set_Gbl_Constrnt(Equal(signal, 3000));

3. Then the expression to be simplified is sent to Maple. The "ForAll" and "Equal" function names are procedures in Maple with the same names.

   And(ForAll(`db`, Greater(`db`,signal)), Equal(`gauge`, 3000));

4. Then a command sent to Maple instructs it to simplify the expression using CasEndProc as:

   CasEndProc(SAST_expn);
   where SAST_expn is the name of the expression to be simplified.

5. The result of the simplification is saved in a temporary file that is read by the parser. The TTS compatible expression from the parser is then saved in the same input context file with the name "SAST_newExpn." The result is:

   (∀ db, db > 3000));

Example 2: Tabular Expression.

This example shows intermediate stages during specialization and simplification of a tabular expression in SAST. Refer to Chapter 4 for more details

Input:

Constraints:
s_irr_band = low_irrational
m_signal = 100
db = 15
Expression:

f_irr_band = **H_2**

$\mathbf{p_T}$: $H_1 \wedge G$ $\mathbf{r_T}$: H_2	low_irrational	rational	high_irrational
s_irr_band = high_irrational	m_sig ≤ 100	100 < m_sig ∧ m_sig ≤ 4900 - db	4900 - db ≤ m_sig
s_irr_band = low_irrational	m_sig < 100 + db	100 + db ≤ m_sig ∧ m_sig < 4900	4900 ≤ m_sig
s_irr_band = rational	m_sig ≤ 100	100 < m_sig ∧ m_sig < 4900	4900 ≤ m_sig

H_1 **G**

FIGURE 19 - Inverted Function Table (Original)

Specialization and Simplification Steps:

1. Initialize the Maple session using default definitions, declaration of constants and Maple procedures. The symbolic constants in FIGURE 19 are *low_irrational, rational and high_irrational*. These symbolic constants are declared using the procedure "DeclareConst" written in Maple programming language.

2. Constraints are set using "Set_Gbl_Constrnt", a Maple procedure as follows:

```
Set_Gbl_Constrnt(Equal(`s_irr_band`, low_irrational ));
Set_Gbl_Constrnt(Equal(`m_signal`, 100));
Set_Gbl_Constrnt(Equal(`db`, 15));
```

3. For tables, expressions from each cell have to be simplified first using the given constraints. The result is then saved back into the same cell. The table at this stage is:

f_irr_band = **H_2**

p_T: $H_1 \wedge G$ **r_T**: H_2	low_irrational	rational	high_irrational
false	true	false	false
true	true	false	false
false	true	false	false

H_1 **G**

FIGURE 20 - Inverted Function Table

4. Since FIGURE 20 is a Inverted function table, headers H_1 and G contain conditions and H_2 contains the value of the function. A *false* in the headers indicates that this condition is not valid in the domain restricted by the given constraints so the rows and/or columns are removed. Also if all the cell expressions in entire row and/or column of main grid are *false,* then it can be removed. The resultant table after the removal of rows and columns is shown in FIGURE 21.

5. For final simplification, the expression in the value grid and the expressions in the headers are combined using table rules (predicate rule: H_1 and G; relation rule: H_2). The combinations of conditions from guard grids, H_1 and G along with the expression from the value grid H_2 form a table element that is represented by TABELT, also a procedure in Maple that simplifies table element. Pred_True represents the predicate value ***true***. The following is sent to Maple:

6. TABELT(And(Pred_True, Pred_True),low_irrational);

7.

f_irr_band = **H_2**

$\mathbf{p_T}$: $H_1 \wedge G$ $\mathbf{r_T}$: H_2	low_irrational
true	true

H_1 **G**

FIGURE 21 - Inverted Function Table

6. The resultant table, FIGURE 22 in this case is same as before as it involves no more simplification.

f_irr_band = **H_2**

$\mathbf{p_T}$: $H_1 \wedge G$ $\mathbf{r_T}$: H_2	low_irrational
true	true

H_1 **G**

FIGURE 22 - Inverted Function Table

7. In this expression the table occurs as a part of an expression. In order the maintain the structure of the tabular TTS expression, a temporary name is assigned to the placeholder of the table. The table expression and the temporary name are saved in the symbol table. In the parsed resultant expression from Maple, the table is substituted back in place of the temporary name . The following expression is sent to Maple with a temporary name in place of the table:

   Equal(`f_irr_band`, sast_SAST_AAAaajfqa);

8. The result of specialization and simplification of the input tabular expression under the given constraints is FIGURE 23:

f_irr_band = **H_2**

$\mathbf{p_T}$: $H_1 \wedge G$ $\mathbf{r_T}$: H_2	low_irrational
true	true

H_1 **G**

FIGURE 23 - Inverted Function Table

6.2 Applications

One of the major applications of Specialization And Simplification Tool (SAST) is in its usage as an visualization tool. It provides a focused view of the expression which describes or specifies a program under given constraints or pre-conditions. It achieves this for a table by simplification of sub-expressions under given constraints and further removal of rows or columns based on the semantics of the input table. This simpler table for a reduced domain helps in inspection, which is usually done case by case.

It can also be used as an intermediate simplification utility tool while transforming one type of tables to another. Usually transformation tools like Invertor[14] results in a complex table that can be simplified using SAST as a large number of cell expressions are ***false***.

As SAST uses symbolic computation it can also be utilized for software testing where values for the all variables cannot be pre-determined. It can be used effectively as an analysis tool for understanding the specification and thus the behaviour of the software. By knowing the contextual information in terms of user defined conditions, optimizations might be possible during implementation. Simpler expressions thus generated can also be used for automatic optimized code generation.

Instantiated functional equivalence of expressions is also possible when instantiations of variables are enumerated.

---

[14] Invertor transforms a Normal table to an Inverted table, described in [22].

6.3 Conclusions

Development of SAST has shown the feasibility of automating specialization and simplification of expressions. SAST is useful in enhancing and analyzing tabular specifications because it provides a simpler view based on users' constraints. It discards what is not desired or applicable and thus helps in gaining a better understanding of the program specified by the table. Therefore it can also be used as an visualization tool.

Since SAST uses Maple for its symbolic computation needs, it has an added advantage of giving results in a very intuitive way and can also handle partial evaluation of expressions. Maple gives accurate results as compared to tools or techniques that apply approximate floating point arithmetic. Because of Maple's accurate results this tool can also be used in testing where the availability of values for all variables is not always possible and where the desired value is available or can be inferred from the results obtained by the tool. It takes advantage of Maple's vast mathematical knowledge base without having to re-invent the wheel. It generates no intermediate code which localizes sources of all possible errors to Maple for a given correct input. While using these CAS one should be careful about the inherent transformations that lead to automatic simplifications. SAST was designed to provide a flexible interface between TTS and other mathematical software.

To describe user defined domains, another CAS with good type system would have been desirable for SAST. Also, development of infrastructure in TTS to support a type system would help in enhancing SAST.

Development of this tool has reaffirmed the author's belief in a modular approach for development of a large software project by gradually integrating different modules. During the development process of this tool, special care was taken to make this tool as general and flexible as possible, keeping in mind possible future work and changes. While developing this tool, ideas of separation of concerns and information hiding were also incorporated.

6.4 Future Work

At present, SAST only handles specialization by assignment using Equal or conjunction of Equals but it can be extended to handle all other combinations of logical expressions with $>$, $<$, $\geq$,$\vee$, etc., in logical expressions. This can be done by either using Maple's capabilities or writing separate procedures in Maple for the desired operation. It can be adapted to handle full logic functionality including universal (ForAll) and existential (ThereExists) quantifiers. An option of using

Maple's logic capability can also be further investigated as an extension of SAST. There also exists a possibility of combining Maple and a theorem prover to work in tandem.

The scope of SAST can also be broadened to include sets, lists, and other domains (including user defined domains). This would help in incorporating enumerated types i.e. SWITCH_POSITION = { ON, OFF }. Some further simplifications are possible if SAST's capability could be extended to handle assumptions like ON $\vee$ OFF = *true*. A situation which occurs while transforming a table type to another might have tautological expressions which are always *true* leading to simplifications like C1 $\vee$ C2 $\vee$ C3 = *true* or C1 $\vee$ C2 = $\neg$ C3. Since simplification by nature is context dependent, knowing the domain of operation could be advantageous in exploiting full prowess of Maple. SAST only utilizes information about constants and symbolic constants (i.e. terms with PConstTag or ConstTag), as simultaneous substitution of variables is limited only to equality. Further simplifications might be possible by knowing the types of the terms as well.

In SAST, the symbol table is used to store all the definitions. All these definitions are in Maple syntax so it becomes imperative for a user to have some basic knowledge of Maple syntax to define new functions. These definitions are in the default MapleDefn class of the symbol table, which stores all the information pertaining to symbols used in the given expression. Since all the default definitions are already included, only new definitions would have to be added. A TTS to Maple syntax generator for generating definitions can be developed to help in defining new functions.

SAST can only handle a few commonly used types of tables which are within the scope of this work. While removing the rows or columns, the dimensionality of the table is maintained in SAST. This restriction can be removed by allowing the new simplified table to have fewer dimensions than the original table (input table) or even a scalar form. For a tabular expression, combining rows and/or columns when cells expressions are the same also results in a simpler table.

Another area of future work will be to extend the capability of SAST to add an option for differentiating whether the computation should be numeric or symbolic.

Constraint solving can be performed to remove redundancy and inconsistency before applying to a given expression. This would avoid unnecessary computations and will improve performance of the tool.

Some of the issues about specialization which are not dealt in this thesis are:

- management of numerous specialized tables due to different constraints
- generation of optimized code based on the specialized table

A very simple command line interface is now used by SAST that can be changed to a graphical interface.

Appendix A

Module Interface Specification

Appendix A includes the module interface specification for the TTS_Maple Interface (TTStoMaple) module which is a part of the Specialization and Simplification tool. This module can be used by developers of other tools that need to interface with Maple or any other Computer Algebra System for computing.

A.1 TTS_Maple Module Interface Specification

(0) CHARACTERISTICS

Imported Types: Expn, SymTbl, Maple_line

(1) SYNTAX

The module initialization program must be the first program if a tool uses the TTS_Maple Interface module.

Access Programs

Access Program Name	Arg#1
TTStoMaple_init	SymTbl

Access Program Name	Arg#1	Arg#2	Arg#3
TTStoMaple_sendExpn	Expn	Path	
TTStoMaple_callProcFSave	char *	int	args[]
TTStoMaple_Save	FILE *		
TTStoMaple_readFile	char *		
TTStoMaple_sendLine	Maple_line		
TTStoMaple_cleanup			

Constants

Maple_InFName - Name of the input file for the Maple session

Maple_OFName - Name of the output file in which the Maple result is saved

(2) ABSTRACT REPRESENTATION

Name	Description
InFile	File named Maple_InFName
status	The error status given in SAST_error. SuccessSAST if no error token given.
OutFile	File named Maple_OFName
initialized	TRUE when module is initialized, FALSE otherwise.
Link	The UNIX pipe that is established to facilitate an interprocess communication between SAST and Maple.

(3) BEHAVIOUR (for “good” case only)

TTStoMaple_init(table):

(initialized’ = TRUE) $\wedge$ isPresent_Link(Link’) $\wedge$
Maple_Session(convertInfo(table))

TTStoMaple_sendExpn(e, p):

(InFile’ = convertExpn(e,p)) $\wedge$ (OutFile’ = Maple_Session(InFile’))

TTStoMaple_sendLine(line):

(InFile’ = convertLine(line)) ∧ (OutFile’ = Maple_Session(InFile’))

TTStoMaple_callProcFSave(fileptr, proc, arity, args[]):

(InFile’ = convertProc(proc, arity, args[])) ∧ (fileptr’ = Maple_Session(InFile’))

TTStoMaple_callProcSave(proc, arity, args[]):

(InFile’ = convertProc(proc, arity, args[])) ∧ (OutFile’ = Maple_Session(InFile’))

TTStoMaple_cleanup():

¬isPresent_Link(Link’) ∧ ¬isPresent(Maple_InFName) ∧
¬isPresent(Maple_OFName) ∧ (initialized’ = FALSE)

TTStoMaple_Save(fileptr):

fileptr = Maple_Session(InFile’)

TTStoMaple_readFile(fptr):

OutFile’ = Maple_Session(fptr)

AUXILIARY PREDICATES

isPresent_Link: char → bool

isPresent_Link(pname) = (popen(cmd, “w”) == NULL)

popen is used to establish an interprocess communication through pipe between SAST and Maple. *cmd* is the command used to invoke Maple.

isPresent_File: char → bool

isPresent_File(fname) = (fopen(fname, “r”) == NULL)

fopen is used to open a file.

AUXILIARY FUNCTIONS

convertInfo: SymTbl → FILE *

Description: Information is extracted from the input symbol table and saved in a file.

convertExpn: Expn $\times$ Path $\rightarrow$ FILE *

Description: Expression from the path is converted to a Maple compatible syntax and is saved in a file.

convertLine: Maple_line $\rightarrow$ FILE *

Description: A line object containing the expression is saved in a file.

convertProc: char * $\times$ int $\times$ args $\rightarrow$ FILE *

Description: A procedure with integer number of arguments is called and the result is saved in a file.

Maple_Session: FILE * $\rightarrow$ FILE *

Description: Maple session utilizes information from an input file and the result is saved in an output file.

References

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

2. Brackxx F., Constales D., “Computer Algebra with LISP and REDUCE: An Introduction to Computer-aided Pure Mathematics”, Kluwer Academic Publishers group, The Netherlands, 1991.

3. Buchbberger B., Collins G. E., and Loos R. in corporation with Albrecht R., “Computer Algebra Symbolic and Algebraic Computation”, second edition, edited by Springer-Verlag/Wein, 1983.

4. Char B. W., Geddes K. O., Gonnet G. H., Leong B. L., Monagan M. B., Watt S. M., “Maple V Language Reference Manual”, Springer- Verlag, 1992.

5. Char B. W., Geddes K. O., Gonnet G. H., Leong B. L., Monagan M. B., Watt S. M., “Maple V Library Reference Manual”, Springer-Verlag, 1991.

6. Consel C., Danvy O., “Tutorial Notes on Partial Evaluation”, *ACM Symposium on Principles of Programming Languages*, 1993, pp 493-501.

7. Coen-Porisini A., De Paoli F., Ghezzi C., Mandrioli D., “Software Specialization via Symbolic Execution”, *IEEE Transactions of Software Engineering*, Vol. 17, No. 9, September 1991, pp 884-889.

8. Heck A., “Introduction to Maple”, Springer-Verlag, 1993.

9. Hester S.D.L., Parnas D. L., Utter D. F., “Using Documentation as a Software Design Medium”, *Bell System Technical Journal*, Vol. 60, No. 8, October 1981, pp 1941-1977.

10. Janicki R., “Towards a Formal Semantics of Parnas Tables”, 17th International Conference on Software Engineering, *IEEE Computer Society*, Seattle, WA, April 1995, pp. 231-240.

11. Janicki R., Parnas D. L., Zucker J., “Tabular Representations in Relational Documents”, Relational Methods in Computer Science, C. Brink, W. Kahl and G. Schmidt (eds.), Springer-Verlag New York Inc., 1996.

12. Mathematica: A System for Doing Mathematics by Computer, Wolfram S., second edition, Addison-Wesley Publishing Company, 1991.

13. Parnas D. L., “On a Buzzword: Hierarchical Structure”, *Proceedings of the IFIP Congress 1974*, North Holland 1974, pp. 336-339.

14. Parnas D. L., Clements P. C., “A Rational Design Process: How and Why to Fake it”, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp 251-257

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

16. Parnas D. L., “Predicate Logic for Software Engineering”, *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, September 1993, pp. 856-862.

17. Parnas D. L., Madey J., Iglewski M., “Precise Documentation of Well-Structured Programs”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, December 1994, pp. 948-976.

18. Parnas D. L., Madey J., “Functional Documentation for Computer Systems”, *Science of Computer Programming*, Elesevier, Vol. 25, No. 1, October 1995, pp. 41-61.

19. Peters D. K., “Generating A Test Oracle From Program Documentation”, *CRL Report 302*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Canada, April 1995.

20. Po C., Autrey T., Black A., Consel C., Cowan C., Inouye J., Kethana L., Walpole J., Zhang K., “Optimistic Incremental Specialization: Streamlining a Commercial Operating System”, *Proceeding of 15th ACM Symposium on Operating Systems Principles*, Colorado, December 1995, pp 314-324.

21. Scherlis W. L., “Program Improvement by Internal Specialization”, *Proceedings of Eighth Annual ACM Symposium on Principles of*

Programming Languages, Williamsburg, Virginia, January 1981, pp. 41-49.

22. Shen H., "Implementation of Table Inversion Algorithms", *CRL Report 315*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Canada, December 1995.

23. Shen H., "Table Transformations Tools: Why and How", *CRL Report 328*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Canada, May 1996.

24. Software Engineering Research Group, "Table Tool System Developer's Guide", *CRL Report 339*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Canada, January 1997.

25. Tyson A., "Composition Tool", Masters Thesis (*work in progress*), Communication Research Laboratory, McMaster University, Hamilton, Canada, 1997. Private Communication.

26. Viola M., McDougall J., Moum G., "Tabular Representation of Mathematical Functions for the Specification and Verification of Safety Critical Software", *Technical Report TR702*, Ontario Hydro, Toronto, Canada

27. Wester M., "A review of CAS Mathematical Capabilities", ftp from an internet site math.unm.edu. Earlier versions have been published in "*Applied Mechanics in the Americas*", Vol III, edited by Godoy Luis A., Idelsohn Sergio R., Laura Patricio A. A., and Mook Dean T., American Academy of Mechanics and Association Argentina de Mecanica Computacional, Santa Fe, Argentina, 1995, pp 450-455. Another 1994 version of this paper was published in *Computer Algebra Nederland Nieuwsbrief*, Number 13, December 1994, ISSN 1380-1260, pp 41-48.