Preliminary Requirements Checking Tool

By
Ou Wei, M.Eng. B.Sc.

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

Master of Science
McMaster University

© Copyright by Ou Wei, April 25, 2001

MASTER OF SCIENCE(2001) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: Preliminary Requirements Checking Tool
AUTHOR: Ou Wei
M.Eng. (Institute of Software, Academia Sinica, China)
B.Sc. (Lanzhou University, China)
SUPERVISOR: Dr. David L. Parnas
NUMBER OF PAGES: ix, 75

Abstract

Requirements play an important role in software systems developments. The impact of errors in requirements is costly, especially for safety and critical systems. Two kinds of properties are necessary in a formal requirements specification, application-independent properties and application properties. Application-independent properties are simple properties derived from the underlying formal requirements model and specification notation. Although detecting the failure to satisfy the application-independent properties is usually simple, the large size of requirements documents means that reviewers must spend considerable time and effort checking them. Computer-supported preliminary checking tools are necessary for industrial application of formal requirements methods and improving the quality of requirements documents.

In this thesis, a Preliminary Requirements Checking Tool (PRCT) is developed for this purpose. It checks the application-independent properties for SCR style requirements. The properties checked by PRCT are derived from the Four Variable Requirements Model [29] and Generalized Tabular Notation [27, 1]. The development of PRCT is based on the previous work on the Table Tool System (TTS) [31]. This tool will help to automatically check for errors like wrong syntax, undefined variables and circular definitions in requirements specification and will serve as a preprocessor for more advanced tools that will check the critical application properties of requirements.

Acknowledgements

I would like to express my sincere thanks and deep appreciation to my supervisor, Dr. David L. Parnas, for his guidance, insight, and enthusiasm throughout my thesis work. I have learned a lot from him in both academic and non-academic areas. Without his consistent encouragement and support, it would have been impossible for me to finish this work.

I am grateful to Dr. Ryszard Janicki and Dr. Ridha Khedri, for reviewing my thesis, and for their valuable suggestions and comments. I would like to thank all the members of the Software Engineering Research Group, especially Jianwei Zhou and Min Jing, for their helpful discussions.

Special thanks to my wife, Yun, and my family, for their love, encouragement and support.

Finally, I would like to acknowledge the financial support received from Natural Science and Engineering Research Council(NSERC) and Bell Canada.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a brief introduction to the thesis, including background, motivation and thesis scope.

1.1 Background

Requirements documents play an important role in software systems development; their importance has been repeatedly recognized during the past 25 years [22]. In [6], Bell and Thayer observed that inadequate, inconsistent, incomplete or ambiguous requirements are very common and have a critical effect on the quality of the resulting software. The impact of errors in requirements on cost and safety has been demonstrated empirically. In a study of 387 software errors found during integration and system testing in NASA's Voyager and Galileo spacecraft, Lutz reported that safety-related software errors arose most often from inadequate or misunderstood requirements [23].

It follows that improving the quality of requirements documents is crucial. Recent studies have shown that applying mathematical methods during the requirements phase is a promising approach to increase the clarity and precision of requirements specifications, and to find important and subtle errors [24, 13]. With such methods, we apply traditional engineering principles and mathematical techniques to produce a requirements specification based on some formalism. Mathematical methods

can reduce errors in requirements and then improve the level of assurance by increasing precision and unambiguity and making some instances of inconsistency and incompleteness obvious [15].

While they can increase the confidence in requirements, mathematical methods are not a miracle cure. They do not assure the correctness and do not guarantee the completeness and correctness of the specifications themselves [13]. In practice, because of carelessness, misunderstanding in communication within teams, and most important, the complexity of the requirements process, errors could be still introduced while writing requirements documentation. In the certification of the Darlinton plant, Parnas noticed that the formal documents submitted for reviews often fail to satisfy certain mathematical conditions [28].

1.2 Motivation

Although errors will still exist in formal requirements documents, thanks to the rigor provided by mathematical methods, such as the underlying requirements model and formal specification notation, we can conduct more thorough, rigorous and systematic analysis to detect errors in requirements specifications. The analysis can go beyond what can be found by simple word processing functions such as "spell checking" for requirements specified in prose format with natural language and reveal errors that the traditional informal techniques have missed.

Generally, to ensure the correctness, we need to check two kinds of properties of a formal requirements specification, *application-independent properties* and *application properties*. Application-independent properties are simple properties, like syntactical correctness, completeness, and consistency. These properties are derived from the underlying formal requirements model and formal specification notation. Checking these properties is independent of a particular application. Application properties include safety properties, security properties, and real time properties. They are the kinds of system goals which should hold in the system being specified. Formulating and checking these properties is dependent on properties of a particular application [14, 12].

Although detecting the failure to satisfy the application-independent properties is usually simple, the size of requirements documents that must be checked in practice leads the reviewers to spend considerable time and effort to verify them. In the certification of the Darlinton plant, Parnas observed that the reviewers spent too much time and energy checking for simple application-independent properties and he then suggested that the preliminary checking be done by a computer [28]. In [10], based on a study of 12 formal methods, it is indicated that simpler tools like syntax checkers, rudimentary editors etc. are badly needed for industrial application of formal methods.

1.3 Thesis scope

In this thesis, based on above observations, a Preliminary Requirement Checking Tool (PRCT) is developed for Software Cost Reduction (SCR) style requirements. This work is based on our experiences of processing generalized tabular notations and the available tools in "Table Tool System" (TTS) [31]. We will provide a syntax to represent the simplified SCR behavior requirements, and describe the requirements specification using a framework file and a TTS context file. This thesis also defines the application-independent properties checked by PRCT, which are derived from the underlying requirements model (the Four Variable Model) [29], and formal specification notation (Generalized Tabular Notation) [27, 1]. This checking tool requires no knowledge of the actual requirements and can be done directly and mechanically using only the information provided by the requirements specification files.

PRCT will relieve reviewers of many tedious routine checking tasks, letting them focus on more difficult, safety relevant issues and concentrate on whether the requirements document describes what is really needed. In addition, the requirements that pass the checker can also serve as input to more advanced and complex tools that check critical application properties.

1.4 Thesis outline

Chapter 2 describes the SCR requirements method. Chapter 3 introduces tabular notations used in this thesis and the support system - Table Tool System. Chapter 4 summarizes the related work about the process for SCR style requirements. Chapter 5 describes the organization of SCR requirements specification, and the application-independent properties checked by PRCT. Chapter 6 discusses the design of PRCT, including informal requirements and module decomposition. Chapter 7 discusses the limitation of this thesis and proposes some future work.

Chapter 2

SCR Requirements Specification

This chapter gives an introduction to the SCR requirements specification method. The "Software Cost Reduction" (SCR) method was first developed in a project of Naval Research Lab of the United States to specify the requirements for the operational flight program of the A-7E aircraft unambiguously and concisely [17]. This project resulted in the SCR requirements and design standards, as well as guidelines for software development using SCR. This method has been applied successfully to a variety of practical systems, such as telephone networks and nuclear power plant, and shown to satisfy industrial needs of requirements documentation [18, 7]. The following sections introduce the underlying requirements model - Four Variable Model and the structure of SCR requirements specification.

2.1 Four Variable Model

The initial use of SCR approach lacked an underlying formal semantics. In [29], the Four Variable Model is proposed by Parnas and Madey who provide a formal framework for SCR method.

The Four Variable Model illustrated in Figure 2.1 [11] describes a system requirements with a set of mathematical relations on four set of variables: monitored and controlled variables, and input and output data items. A *monitored variable* is an environmental quantity that affects the system behavior, and a *controlled variable*

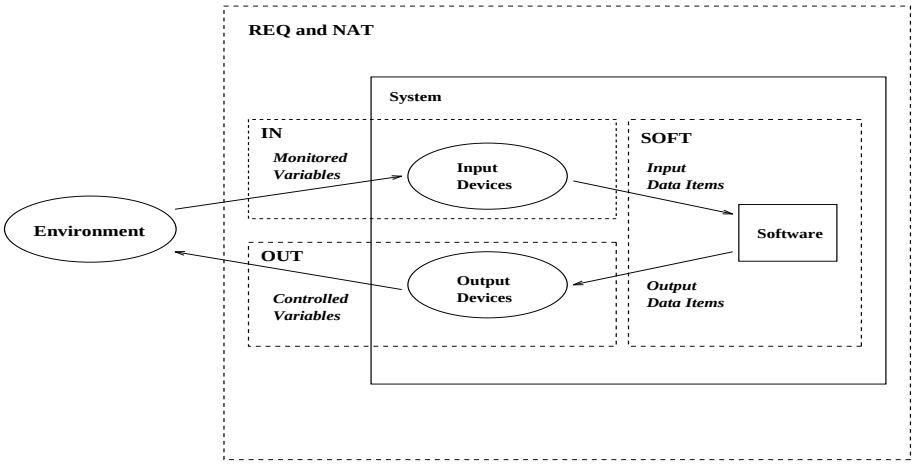

Figure 2.1: Four Variable Model

is an environment quantity that the system controls. The monitored and controlled variables constitute the environmental quantities that are visible to the users and independent of the chosen solutions. They are the best quantities to describe the requirements for the system [30]. These quantities include such things as physical properties, values displayed on output devices, and settings for input switches and controlled devices. The requirements specification of the system behavior is given as a black box with monitored and controlled variables as its inputs and outputs respectively. The specification is described by two relations NAT and REQ.

- NAT defines the set of possible values of variables, describing the natural constraints on the system behavior. These constraints are imposed by physical laws and the system environment.
- REQ defines the additional constraints on the desired system by stating as relations that the system must maintain between the monitored and the controlled variables.

In the Four Variable Model, input devices measure the monitored variables and output devices set the controlled variables. *Input* and *output data terms* represent the values that the devices read and write. The relations between monitored variables and input date items, controlled variable and output data items are captured by IN and OUT relations respectively.

- IN defines the mapping from monitored variables to the input data items.
- OUT defines the mapping from the output data items to controlled variables.
- SOFT defines the mapping between input data items and output data items.

The Four Variable Model defines the system requirements by describing the required relation between the monitored and controlled variables - REQ, and defines the software requirements by describing the required relation between input and output data items - SOFT. Using monitored and controlled variables to define the required system behavior, instead of using input and output data items, keeps the specification in the problem domain and make the specification simpler.

2.2 SCR requirements specification

Generally, a complete SCR requirements specification consists of three parts: behavior requirements, environmental assumptions and system goals [11].

2.2.1 Behavior requirements

SCR requirements model a system as an event-driven state-transition machine. The system's environment is abstracted as a set of monitored and controlled variables. Changes of monitored variables may cause the system to change its state or alter the values of its controlled variables. The behavior requirements describe the system's activities by describing the system's modes of operations, specify the events that cause the system to change operational mode and define the changes of controlled variable values. In addition to monitored and controlled variables, there are four constructs used to represent system behavior in SCR: *modes*, *terms*, *conditions* and *events*.

- **mode** A *mode* is a set of system states. A *mode class* is a partitioning of the set of system states and each partition will be called a mode in that class. This is an internal view of modes. Another external view treats a mode as a set of system histories including the values of all monitored variables over a period of time starting from system creation. A mode class is a partitioning of the set of histories and each partition will be called a mode in that class [30]. If the

system is correctly implemented, the mode transition function will be equivalent for both definitions.

- **term** A *term* is an auxiliary function defined on monitored variables, modes or other terms. That is, a term is a function built up on the other constructs in a SCR specification. It can be thought of as a kind of intermediate computational results. The purpose of defining a term is to help make the specification concise.
- **condition** A *condition* is a predicate defined on monitored variable, term or modes. Conditions are actually terms which only evaluate to True or False.
- **event** An *event* is a change to a condition's value. Conditions can change their values at most once during a single time unit under the assumption that time is discrete. Events are usually be detected at the point in time at which they occur.

An SCR requirements specification may contain one or more mode classes. The system is in exactly one mode of each mode class at any time. Each mode class specifies one aspect of the system's behavior. Transitions between modes are triggered by events. The value of each controlled variable is specified by a function defined in terms of the system modes, monitored variables and terms.

To facilitate the specification of behavior requirements, tabular notations are introduced in SCR documents to define the values of controlled variables and mode transitions. Tabular notations are used because tabular specifications of requirements are easier to understand and to maintain, and large quantities of requirements information can also be described concisely using tables. In SCR documents, three kinds of tables are used: *condition tables*, *event tables*, and *mode transition tables*. Each SCR table describes a mathematical function [15].

A condition table defines a controlled variable or term as a function of a system's modes and conditions. The typical format of a condition table is shown in Table 2.1. In Table 2.1, the first column shows a particular mode m_j of the system. The k_{th} column under caption "Conditions" in the table specifies r_i's value to be v_k when the condition $c_{j,k}$ is held.

Table 2.1: Typical format of condition table

Modes	Conditions				
m_1	$c_{1,1}$	...	$c_{1,k}$	...	$c_{1,p}$
...	...	...	...	...	...
m_j	$c_{j,1}$	...	$c_{j,k}$	...	$c_{j,p}$
...	...	...	...	...	...
m_n	$c_{n,1}$	...	$c_{n,k}$	...	$c_{n,p}$
r_i	v_1	...	v_k	...	v_p

An event table describes a controlled variable or term as a function of a system's modes and events. Table 2.2 gives the typical format of an event table. In Table 2.2, if an event $e_{j,k}$ occurs when the system is in a specific mode m_j, r_i's value change is specified by v_k.

Table 2.2: Typical format of event table

Modes	Events				
m_1	$e_{1,1}$	...	$e_{1,k}$	...	$e_{1,p}$
...	...	...	...	...	...
m_j	$e_{j,1}$	...	$e_{j,k}$	...	$e_{j,p}$
...	...	...	...	...	...
m_n	$e_{n,1}$	...	$e_{n,k}$	...	$e_{n,p}$
r_i	v_1	...	v_k	...	v_p

A mode transition table is used to specify a mode class's transition relation, describing a system's next mode as a function of the current mode and an event. A typical format of a mode transition table is given in Table 2.3, where each row describes an event $e_{i,j}$ that activates a transition from the mode m_i on the left to the mode $m_{i,j}$ on the right.

Table 2.3: Typical format of mode transition table

Current Modes	Event	New Modes
m_1	$e_{1,1}$	$m_{1,1}$
	...	...
	$e_{1,i}$	$m_{1,i}$
	...	...
	e_{1,k_1}	m_{1,k_1}
...	...	...
m_n	$e_{n,1}$	$m_{n,1}$
	...	...
	$e_{n,i}$	$m_{n,i}$
	...	...
	e_{n,k_n}	m_{n,k_n}

2.2.2 Environmental Assumptions and System Goals

A complete SCR requirements document also includes two other parts: *environmental assumptions* and *system goals*.

An environmental assumption is the NAT relation in the Four Variable Mode, which describes the natural constraints on the monitored and controlled variables. These constraints are imposed by the physical laws or system environment. Environmental assumptions are also called *environmental constraints* [30], which are invariant constraints for all the system states.

The REQ relation in the Four Variable Model describes the required behavior of a system. System goals are the properties that the system must meet. These properties are implicit information that can be extracted from the system's behavior specification. These goals could be mode invariant, global behavioral statement or temporal properties. A correct system behavior requirements specification should be

expected to enforce these properties. System goals provide a way to check REQ.

In this thesis, we will focus on the behavior aspects of SCR style requirements, and develop a tool to check the basic system application-independent properties.

Chapter 3

Tabular Notation and the Table Tool System

This chapter introduces the Generalized Tabular Notation used in this thesis and the support system - Table Tool System [1, 19].

3.1 Tabular expressions

The earliest use of tabular expressions was done without formal definitions. They are used in an ad hoc manner based on the intuitive understanding coming from our daily lives. To provide widespread use and mechanical support of tabular notations, Parnas gave the initial syntax and semantics of formal tabular notations [27]. Tabular notations are multi-dimensional tables used to represent relations and functions. They have no more expressive power than expressions written in traditional manner, but they are much more suitable for describing the conditions, relations and functions, that frequently occur in program description.

3.1.1 Syntax of tables

Tabular expressions are constructed recursively from scalar expressions, which are either terms or predicate expressions, and grids. Tables are described as a collection of sets of cells classified into main grid and headers as follows.

- A header H is an indexed set of cells, $H = \{h_i | i \in I\}$, where the index set of H is $I = 1, 2, \ldots, n$.
- A main grid G is an indexed set of cells, $G = \{g_\alpha | \alpha \in I\}$. G is indexed by headers $H_1, \ldots, H_n$, with $H_j = \{h_i^j | i \in I^j\}$ where $j = 1, \ldots, n$ and $I = I^1 \times \ldots \times I^n$. The set I is the index of G.
- A *raw element* is a tuple $e_\alpha =< h_{i_1}^1, \ldots, h_{i_n}^n, g_\alpha >$ where $i_j \in I^j, j = 1, \ldots, n, \alpha = (i_1, \ldots, i_n) \in I$.

A table that is interpreted as a function is called a function table. A table that is interpreted as a predicate is called a predicate table. A function table is also a term. A predicate table is also considered to be a predicate expression. A table can be expressed in other tables.

3.1.2 Semantics of tables

The formal semantics of tables was initially proposed by Parnas in [27]. Janicki and Khedri later proposed a model of table semantics in [20, 21]. Based on these, Abraham in [1] presents a formal model used to define generalized tabular expressions. According to her definition, a tabular expression T is a tuple: $T =< H_1, \ldots, H_n, G, \rightarrow, \Psi, \oplus, p_T, r_T >$ where

- Table components, $\{H_1, \ldots, H_n, G\}$.
- A cell connection graph, $\rightarrow$, is a relation represented by an acyclic directed graph with the main grid and all of the headers as nodes. Every arc must start or end at the main grid. It characterizes information flow among the cells and is used for interpretation of tabular expressions. There are four cell connection graph classes, "*Normal*, *Inverted*, *Vector* and *Decision*".
- A table predicate rule, p_T, is the predicate expression containing guard grids. This rule determines the domain of the tabular expression.
- A table relation rule, r_T, is the predicate expression containing value grids. This rule determines the value of the tabular expression.

- a mapping, Ψ, is responsible for assigning a predicate expression, or a part of it, to be the guard cells, or assigning a relation expression, or a part of it, to be the value cells.
- A compose operation, $\oplus$. The tabular expression T is interpreted as a relation, R_T, that can be defined from the relation specified by the table's raw elements, $\{R_\alpha | \alpha \in I\}$, which can be described as $R_T = \bigoplus_{\alpha \in I} R_\alpha$

For example,

Table 3.1: *switch* function

switch(temperature, pressure) H2

	$pressure \leq 3000$	$pressure > 3000$
$temperature < 200$	open	closed
$temperature \geq 200 \wedge temperature < 300$	open	open
$temperature \geq 300$	closed	open

H1 G

For Table 3.1, the table class, table component, guard grid, value grid, predicate rule and realtion rule are given as follows:

- Table class is *Normal*
- H_1, H_2 and G are the table components
- Guard grids are H_1 and H_2
- Value grid is G
- p_T is $H_1 \wedge H_2$
- r_T is G

For SCR tables, Arbraham has shown that the SCR condition and event tables are just special cases of inverted table, and although the format of SCR mode transition table does not fit the semantic model defined above, a SCR mode transition table can be represented using a normal or inverted function table [1]. In this thesis, we only use the generalized tabular notations to specify a requirements document. The following section introduces a support system for generalized tabular notations - Table Tool System (TTS) [31].

3.2 Table Tool System structure

Tabular expressions provide mathematical precision for software documentation. However, manipulation of tables can be tedious and error-prone because of the lack of suitable tools. The Table Tool System (TTS) is a major effort by the Software Engineering Research Group at McMaster University. One of the objectives of the TTS is to support the production of software documentation through an integrated set of tools which manipulate multi-dimensional tabular expressions. The capabilities of TTS currently include expression creation, formatting and transformation. The TTS also aims to automate checking software testing and maintenance. The following parts give an introduction of TTS. Detailed description on the design of TTS is provided in the TTS Developer's Guide [31].

The TTS is decomposed into three tiers: *TTS Kernel*, *TTS Infrastructure* and *TTS Application*.

3.2.1 TTS kernel

The kernel module hides data structures for representing expressions and algorithms for manipulating them. The following definitions are used in this module.

Expn is a data object that represents a mathematical expression. An expression can be represented with a prefix tree structure, where the nodes of the tree are symbols. Symbols are identified by their ids. Expressions are grouped into three categories: atom(constants or variable), applications(functions or predicates with one or more arguments), and tables.

Index is a data object that refers to a particular element of a table by indicating the grid and position of the cell where the element is located.

Path is a data object that identifies the location of a sub-expression of an expression, which consists a sequence of objects, each of which is either an integer or an index. A path can be interpreted as a route map to find a particular sub-expression staring from the root node of the expression tree.

Id is a data object used to identify the symbols of an expression. Each node in a expression tree is a symbol which is assigned an Id as its identifier.

SymTbl is a data object representing a symbol table, which is used to describe a collection of symbols. The information about a symbol is organized into a set of information classes. Some classes are given as follows.

- *Name* class is a string used to represent a symbol.
- *Tag* class is an integer indicating the intended interpretation of an expression, which could be one of the following: ConsTag(Constant), PConsTag(Predicate Constant), VarTag(Variable), FATag(Function Application), PETag(Predicate Expression), LETag(Logical Expression), QLE(Quantifier Logical Expression), FTableTag(Function Table) or PdTableTag(Predicate Table).
- *Rules* class is a string containing the predicate and relation rules of a table expression.
- *CCG* class is an integer used to indicate the cell connection graph of a table, which could be one of the four types: Normal, Inverted, Vector or Decision.

Default Symbols are the most commonly used symbols such as "+","-","∧", etc. The default symbol table (default.sym) contains those symbols with their corresponding information.

3.2.2 TTS infrastructure

The modules in the TTS infrastructure hide data structures and algorithms that provide general purpose functionality for the construction of TTS application. The

infrastructure includes the following modules: *Tools*, *Utilities* and *Tool Integration Framework*.

- *Tools* are modules that operate on TTS objects to provide one or more primitive services to the user. A key concept in this module is *context*. A context is an ordered set of named expressions together with a symbol table. Since all of the expressions use the same symbol table, ids will have consistent interpretation within the context, which simplifies manipulation and interpretation of expressions. A context can be saved to or loaded from a file by calling the access programs of Context Manager. *CHandle* is the data type used to represent a context. In this module, Table Construction Tool [31] can be used to construct the expressions used in requirements files that are checked by the Preliminary Requirements Checking Tool.

- *Utilities* are modules implementing algorithms that maybe used by more that one tool or application. The implementation of the Preliminary Requirement Checking Tool uses some utilities modules, such as Generalized Table Semantics module, Info Utilities module, and Id List module etc. An utility module is also developed in this thesis, which containing access programs to make tabular expressions process more convenient.

- *Tool Integration Framework* makes it easy to integrate new tools and applications to TTS, which provides a menu item for each tool and application, and a mechanism for passing TTS objects between tools and applications.

3.2.3 TTS application

The top layer in TTS provides programs that combine infrastructure modules to manipulate or interpret groups of tabular expressions as documents, which specify or describe some aspect of a computer system. Each hides data structures and algorithms that represent and manipulate relational documentation. For example, the Preliminary Requirements Checking Tool developed by this thesis is an application of TTS used to screen software requirements documents for basic errors.

Chapter 4

Related Work

To meet industry needs and provide automated support for the requirments process, there has been some significant work in developing tools to check and verify SCR requirements. A overview of this related work is given as follows.

4.1 Tools for specifying requirements

1. NRL researchers developed a tool set called SCR* which is an integrated suite of tools supporting the SCR requirements method [14]. It includes a specification editor which provides a graphical interface for creating, modifying and displaying a given requirement specification.

2. In Marsha Chechik's Ph.D. thesis [11], she developed an Analyzer used to check the consistency between requirements and designs. The front end of the Analyzer also provides a graphical interface for specifying the requirements. The front-end can produce an ASCII representation from these requirements.

3. An integrated tool set called SC(R)3 (SCR Requirements Reuse) is also developed by Marsha Chechik [12]. It includes an editor tool allowing user to specify their requirements trough a visual interface. A simplified SCR method is used in this work, where the monitored and controlled variables must have Boolean values.

4.2 Checking application-independent properties

1. In SCR*, the tool used to check application-independent properties is called consistency checker [14, 15]; it checks the properties derived from their defined automaton model and the three kinds of tables. The properties checked by this tool include checking that

   - each table is well-formed based on the representation grammar used to describe tables;
   - each variable has a defined type and the type definitions are satisfied;
   - the value of controlled variables and mode classes are defined;
   - disjointness and coverage properties are satisfied for each table;
   - initial values are defined for all mode classes and variables;
   - each mode in a mode class is reachable;
   - no circular dependencies exist;

2. SCR* also provides a tool called the Dependency Graph Browser [16], which represents the dependencies among the variables in a given SCR specification as a directed graph. By examining this graph, a user can detect errors such as undefined variables and circular definitions. The user can also use the browser to display and extract subsets of the dependency graph.

4.3 Simulation

In SCR*, to validate a specification, a user can run a simulator and analyze the results to ensure that the specification captures the intended behavior [15]. Additionally, the user can define invariant properties believed to be true of the required behavior, and using the simulator to execute a series of scenarios to determine if any violate the invariant. The user can provide the input to the simulator by either entering a sequence of input events or loading a previously stored scenario.

4.4 Model checking

A model checker accepts a system specification and a property specification, and determines whether or not the property holds in the specification. Many work has been done in this area for SCR style requirements.

1. Recently, an explicit state model checker Spin has been integrated into SCR* [16]. After using SCR* to develop a formal requirements specification, a developer can obtain an automatic translation of the specification into Promela, the language of Spin, and then invoke Spin within the tool set to check the properties of the specification.

2. Atlee, Gannon and Chechik have developed tools that enable automated analysis of SCR tabular requirements using the CTL (Computational Tree Logic) model checker [2, 3, 4, 5]. Users manually translate assertions about SCR global properties into CTL temporal logic formulas, and the model checker evaluates the formulas. In [32], to research the feasibility of model checking practical software requirements, Sreemani and Atlee improved the technique by using a symbolic model-checker SMV to analyze the A-7E aircraft requirements.

3. In [11, 12] Chechik developed the analysis tools that compare the design and implementations of a system with their requirements. The tool generates a set of logical formulas from the requirmens and a finite-state machine FSM abstraction of the design. A model checking algorithm is used to determine if the FSM is a model of the formulas. Implementations are annotated to describe changes in the values of their requirement's variables, and dataflow analysis techniques are used to determine the set of events which cause particular state changes.

4. To deal with infinite-state programs and find efficient representations for various variables types, Bultan, Gerber and League proposed a technique that combines multiple type-specific symbolic representations into one model checker. They use the representations of binary decision diagrams (BDDs) for Boolean and enumerate types and arithmetic constraint for integers [9]. The SCR requirements specification of a nuclear reactor's cooling system [8] is analyzed using this method.

Chapter 5

Requirements Representation and Properties Checked

This chapter describes the way that SCR requirements are represented in this thesis and the application-independent properties checked by PRCT.

5.1 SCR requirements representation

This section presents the file organization for SCR requirements specification and the framework file syntax.

5.1.1 File organization for SCR Requirements

The SCR specification used in PRCT contains both ASCII representation and tabular notations. A SCR requirements document consists of two files: the *Framework File* and the *TTS Context File*, which are shown in Figure 5.1.

The framework file contains the static information from the system behavior requirements specification. It includes the following parts: Monitored Variables, Controlled Variables, Mode Classes, Controlled Value Functions, and Dictionary of User-defined Functions and Constants. The framework file can be edited using any text editor.

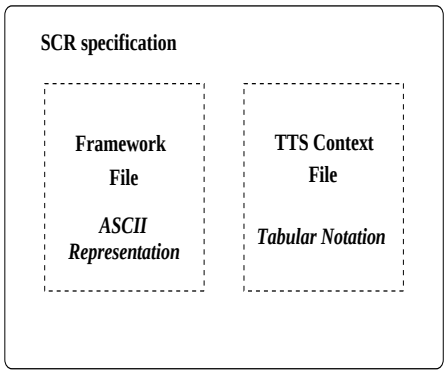

Figure 5.1: File Organization for SCR Requirements

The TTS context file contains all the tabular expressions used to describe modes transition activities, value functions for controlled variables and user defined functions. Expressions are edited using the "Table Construction Tool" (TCT), which determines the accepted expressions syntax [31] [1]. PRCT manipulates and interprets the expressions using the "Table Holder" modules, which store an abstract representations of expressions.

5.1.2 Framework file syntax

Below we define the syntax of the framework file in SCR requirements specification. This grammar is a modified version of the requirement specification language used in [11].

The environmental assumptions and system goals are not presented here, since this thesis will be focused on the behavior aspects. In the original version, monitored and controlled variables are restricted to Boolean data types. We eliminated the restriction in this syntax. In addition, we added a new section, the Dictionary, which includes user-defined functions and constants. This makes the specification more flexible and concise. Another feature of our syntax is that we need not define the syntax for tables, since we directly use the tabular expressions produced by TTS, in

[1] There is another input tool being developed for TTS. It will produce tabular expressions that are compatible with TCT. Tables produced by other tools can also be used.

which there are well-defined representation syntax for general tabular notations and most information necessary for checking tables. Consequently, the syntax defined here is much simpler.

The grammar is described using BNF notation. All non-terminals are in italics and terminals are in plain or boldface type.

This syntax defines the following parts:

- Environmental Variables
  - Monitored variables
  - Controlled Variables
- Mode Classes
  - Initial Mode
  - Modes Declaration
  - Mode Transition Relation
- Controlled Value Functions
- Dictionary
  - User-defined Functions
  - Constants

The first part is the list of declarations for environmental variables, including monitored variables and controlled variables. An initial value for each controlled variable is defined .

/* Monitored Variables */

monvar:
MONITORED *mon_vars_decl_list*

mon_vars_decl_list:
mon_vars_decl
|*mon_vars_decl_list mon_vars_decl*
mon_vars_decl:
TYPE_SPECIFIER mon_vars;
mon_vars:
MON_VAR
|*mon_vars*, *MON_VAR*

/* Controlled Variables */

convar:
CONTROLLED *con_vars_decl_list*
con_vars_decl_list:
con_vars_decl
|*con_vars_decl_list con_vars_decl*
con_vars_decl:
TYPE_SPECIFIER con_vars;
con_vars:
con_var_def
|*con_vars*, *con_var_def*
con_var_def:
CON_VAR = *VALUE*

Following the list of environmental variables is a list of one or more mode class declarations, including Initial Mode, Modes Declaration, Mode Transition Functions. Each mode class declaration must have an initial mode. The Mode Transition Function is represented using a TTS tabular expression, which is defined in the TTS context file.

/* Mode Classes */

modeclasses_decl:

modeclass_decl
|*modeclasses_decl modeclass_decl*
modeclass_decl:
modelclassname_decl initial_mode_decl modes_decl tranition_function
modelclassname_decl:
MODECLASS *MODECLASS_ID*;
initial_mode_decl:
INITIAL_MODE *MODE_ID*;
modes_decl:
MODE *mode_vars*;
mode_vars:
MODE_ID
|*mode_vars*, *MODE_ID*
tranition_function:
TRANSITION_FUNCTION *TTS_TABLE_ID*;

The Mode Classes declaration is followed by the Controlled Value Functions section. In SCR requirements specification, every controlled variable should be associated with one and only one value function, which is used to specify the changes to the variable. The value functions are represented using TTS tabular expressions, which are defined in the TTS context file.

/* Controlled Value Function */

con_value_function:
CONTROLLED_VALUE_FUNCTION *con_value_function_decl*;
con_value_function_decl:
con_value_def
|*con_value_function_decl*, *con_value_def*
con_value_def:
CON_VAR : *TTS_TABLE_ID*

The last part is the Dictionary, which includes User-defined Functions and Con-

stants. Any non-default functions or constants symbols used in tabular expressions must be defined in this section.

/* Dictionary */

dict:
DICTIONARY *functions_decl constants_decl*
functions_decl:
/ empty */*
|**FUNCTION** *function_decl_list*;
function_decl_list:
function_decl
|*function_decl_list*, *function_decl*
function_decl:
TYPE_SPECIFIER FUNCTION_ID(argumenttype_decl) : TTS_TABLE_ID
argumenttype_decl:
TYPE_SPECIFIER
|*argumenttype_decl*, *TYPE_SPECIFIER*
constants_decl:
/ empty */*
|**CONSTANT** *constant_decl_list*
constant_decl_list:
constant_decl
|*constant_decl_list constant_decl*
constant_decl:
TYPE_SPECIFIER constants;
constants:
constant_def
|*constants*, *constant_def*
constant_def:
CONSTANT_ID = VALUE

5.2 Properties checked

The following is the list of properties that will be checked by PRCT. They are application-independent properties derived from the underlying four variable requirements model and the generalized tabular specification notation discussed in chapter 2 and chapter 3. These checks require no knowledge of the actual requirements and can be done directly and mechanically on the information provided by the requirements specification.

- **Proper Syntax**: the framework file is written in the syntax defined above.
- **Initial Value**: initial values are defined for all controlled variables.
- **Initial Mode**: an initial mode is defined for each mode class.
- **No Useless Variables**: PRCT will check the whole requirements to ensure that no useless variables are defined. That is, each monitored variable, user defined function and constant defined in the framework file must be used in the tabular expressions defined in the TTS context file. Otherwise, it will be identified as a useless variable.
- **No Undefined Variables**: PRCT will check the whole requirement to ensure that no undefined variables are used. That is, each symbol used to represent monitored variable, controlled variable, mode, user defined function or constant in tabular expressions of TTS context file must be defined properly in the framework file.
- **Complete Definition**: in the Controlled Value Functions section, each controlled variable must be associated with one and only one value function, and this function should be defined properly in the TTS context file. On the other hand, each controlled variable which is associated with a value function in Controlled Value Functions section should be defined in the Controlled Variables section.
- **Non-terminal Initial Mode**: the initial mode defined in a mode class should not be a terminal mode. A terminal mode is the mode from which no mode

can be reached. For example, in Figure 5.2, mode $m1$ is a terminal mode. The intial mode must be either $m2$ or $m3$.

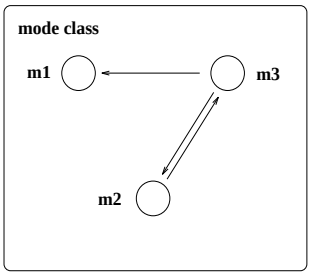

Figure 5.2: Example of Terminal Mode

- **No Isolated Mode**: isolated modes should not exist in a mode class, that is, every mode can be reached from the initial mode in the mode class. In Figure 5.3, mode $m3$ is a isolated mode. We check the properties of "Non-terminal mode" and "No Isolated mode" based on a static analysis of mode transition functions, and assume that all the trigger envents can happen eventually. These checks are a kind of weak reachability check.

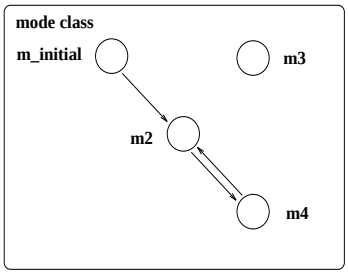

Figure 5.3: Example of Isolated Mode

- **Lack of Circularity**: no circular definition exists among all the controlled variables in the requirements, that is, the *referenced-by relation* (defined below) should be loop free.

Definition 5.2.1 *Referenced-by Relation*

Suppose there are n controlled variables. Controlled variable v_j is said to

be referenced by controlled variable v_i if v_j is used in the value function definition of v_i.

The "Lack of Circularity" property check ensures that the *referenced-by relation* is a partial order relation.

Chapter 6

Design of the Preliminary Requirements Checking Tool (PRCT)

The application tool developed for this thesis is called the Preliminary Requirement Checking Tool (PRCT). This chapter discusses the design of this tool. The first section of this chapter describes the informal requirements, including the user interface and anticipated changes to this tool. The second section describes the module decomposition, including the module guide, the module uses relation, the informal module interface guide and the program uses relation. The third section presents the algorithms used in this tool.

6.1 Requirements (Informal)

The Preliminary Requirement Checking Tool is developed to check the basic application-independent properties derived from the underlying requirement model and specification notation for SCR style requirements. The requirements of the tool are illustrated by Figure 6.1. It accepts two files, *Requirements Framework File* and *TTS Context File* as input, and produces a new output file.

The two files, framework file and TTS context file, constitute the complete system

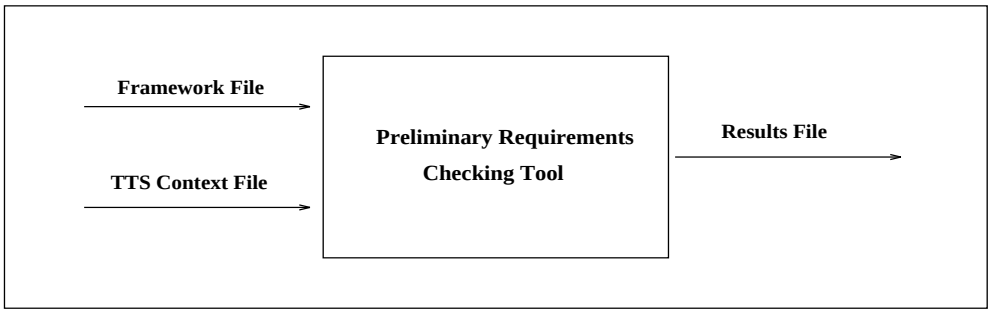

Figure 6.1: Requirement of Preliminary Requirements Checking Tool

behavior requirement specification.

PRCT results are written into the output file, which will contain the detailed information about the related errors found in requirements files.

6.1.1 User interface

PRCT provides a command line user interface based on the following syntax:

prct *frameworkFile ttscontextFile outputFile*

Where

frameworkFile	the framework file name with *.frm* as extension.
ttscontextFile	the TTS context file name with *.tts* as extension.
outputFile	the PRCT results output file name with *.res* as extension.

Since some checks must precede some others, the whole checking process may take several iterations. For example, the syntax check of framework files must precede all the other checks, and only when there is no syntax error found in framework files, can the checking continue. If there are syntax errors found, PRCT will prompt the user to check the errors in *outputFile* and fix the errors, then the user must run the tool again to continue checking other properties.

6.1.2 Anticipated changes

It is expected that the following items are likely to change within the useful life of PRCT.

- SCR requirements representation syntax. Some other parts, like environmental assumptions and system goals, may be added to the requirements specification in the future.
- Results output format. The output file may be changed to provide clearer description of check results.
- Properties checked. More properties, such as type-checking, completeness and consistency would be added in future work.
- Integrated to larger system. PRCT may be integrated into more complicated and complete requirements process system and be called as library functions.
- User interface. More friendly graphical user interface would be developed in the future.

6.2 Module decomposition

This section describes the module structure, module secrets, and module interface of the tool.

PRCT is decomposed into a set of modules to make design easy to understand and modify. The modules of the tool all have a "hidden" implementation . Each of the modules encapsulates specific algorithms and/or data structures. By using the principle of information hiding [25], users of any module need not know anything about the internal data structures and other implementation details of that module. That is, any module can be treated as a black box. The interface to a module consists a set of *access programs*. Other modules can only access or change the data kept by a module by calling the access programs provided by the module. As a result, the design is clearer and easier to understand because of *separation of concerns*. The anticipated changes can be made in one module without affecting the

behavior of the functions in other modules.

6.2.1 Module guide

A module guide is an informal document that states the secrets associated with each module. It describes the structure of the software system by indicating the design decision hidden in each module and the role played by each module in this tool [29].

PRCT is decomposed into seven top level modules, including Master Control Module, Requirements Information Module, Properties Checking Module, Framework File Processor Module, TTS Utilities Module, Result Reporting Module, Status Reporting Module. Some of them are further decomposed into sub-modules.

Master Control Module

The secret of this module is the whole process of checking requirements properties. It acts as the main control module to invoke Requirements Information Module and Properties Checking Module to accomplish the checking tasks. The sequence of invoking access programs of these modules is encapsulated here. The desire to improve the user interface of this tool will led to the programs included in this module to be changed.

Requirements Information Module

The secret of this module is the data structure used to store the information of requirements files needed for checking properties and the algorithm to obtain those information from requirements. It serves as an information provider in this tool, which accepts the requirements specification as input, then extracts and provides the requirements information to other modules. It includes programs that will be changed if there are changes in the organization or representation of requirements document. According to the operations on the two input files - framework file and TTS context file, this module is further divided two sub-modules.

- **Framework File Information Module** The data structure for storing the static requirements information in a framework file is hidden in this module. It invokes Framework File Processor Module to extract this information from the framework file.

- **TTS Context File Information Module** The secret of this module is the data structure used to represent the context and symbol table information of a TTS context file. It provides access programs to setting and retrieving this information.

Properties Checking Module

The secret of this module is the data structure and algorithm used to checking the application-independent properties of requirements files. It is the kernel of PRCT. Based on the different properties we must check, this module comprises the following sub-modules. Each of the sub-modules includes the programs to check certain properties, which will need to be changed if the definition of the properties changes.

- **Variables Checking Module** This module encapsulates the data structure and algorithm used to check the properties "No Useless Variables" and "No Undefined Variables".

- **Mode Class Checking Module** The secret of this module is the data structure and algorithm used to check "Non-terminal Initial Mode" and "No Isolated Mode" properties.

- **Controlled Value Functions Checking Module** This module hides the data structure and algorithm for checking the "Complete Definition" property.

- **Circularity Checking Module** The data structure and algorithm used to checking the property "Lack of Circularity" are hidden in this module.

All the sub-modules are organized in a parallel manner. Each module can accomplish some checking tasks independent of the others. This design makes it easier to change the internal structure of the Properties Checking Module. If there are more new properties need to be checked, we just need develop and add some new sub-modules to the original set.

Framework File Processor Module

This module encapsulates the algorithm used for lexical analysis and syntax analysis of a framework file. It is also responsible for checking the properties of "Proper Syntax", "Initial Value", and "Initial Mode" when doing the analysis work. Three sub-modules are contained in this module.

- **Parser Module** This module contains the data structure and the parser program used to do syntax analysis of the framework file.
- **Lexer Module** The secret of this module is the data structure and algorithm for conducting lexical analysis.
- **Token Lookup Module** This module hides the data structure for representing reserved words and different types of variables used in the framework file and the algorithm to detect reserved words and the types of variables.

The programs included in the Framework File Processor Module are expected to be changed if there are changes on the syntax of the framework file. These changes will be motivated by adding new sections into the framework file, such as environmental assumptions and system goals.

TTS Utilities Module

The secret of this module is the algorithms implementing common tabular expressions processing routines such as determining table types, getting symbols names and obtaining symbols list from a table grid, etc. This module is developed to bridge the gap between the other modules and the TTS libraries. Although TTS has provided basic functions to process tabular expressions, it is not convenient to call these programs directly when developing an application for special purpose. This module consists of the access programs that are often called to process tabular expressions by other modules. If needed, new programs can be added to this module to make processing tables easier.

Result Reporting Module

This module includes the programs called by other modules to output check results. The secret of this module is the data structure for representing check results and recording error numbers, and the algorithm to translate check results tokens into detailed description information. The format for outputting check results is hidden in this module. The programs in the module will need to be changed by the desire to improve user interface of this tool.

Status Reporting Module

The Status Reporting Module hides the data structure used to represent the status of invocation of access programs of PRCT modules and the algorithm for translating a status token into its textual description. It is used by other module of PRCT to enable tool monitoring and status reporting. The list of all the error tokens used in the tool will be interpreted later along with their description.

6.2.2 Module-use hierarchy

Module Uses Relation [26] is used to illustrate the system design of PRCT.

Definition 6.2.1 *Module Uses Relation*

Module A is said to use Module B if some programs in Module A rely on the correct behavior of some programs in Module B to accomplish their tasks.

The first level *Module Uses Relation* of PRCT is shown in Figure 6.2.

6.2.3 Module interface specification (Informal)

A module interface specification treats the module as a black-box, identifying the access programs of the module and describing the externally visible effects of using them [29]. In the following sections, each access program in the modules of PRCT is described informally by indicating the program name, types of arguments, type of return value, and a brief description in the access programs syntax tables.

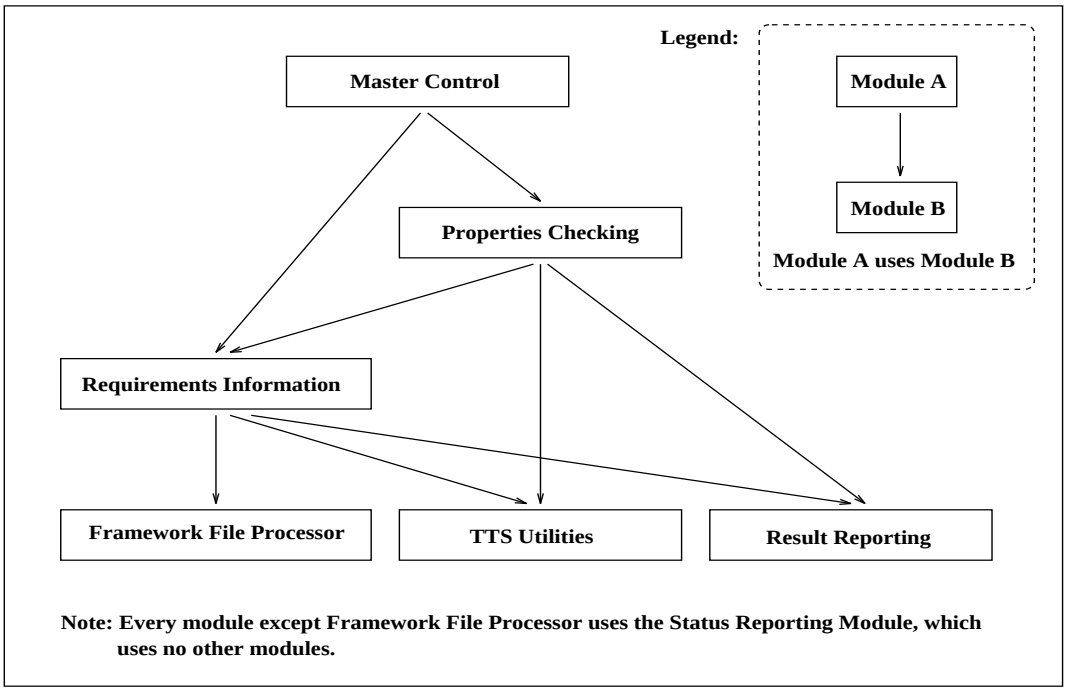

Figure 6.2: First Level Decomposition Module Uses Relation

Master Control Module (PRCT_mctrl.c)

The Master Control Module accepts the framework file and TTS context file as input, calls Requirement Information Module and Properties Checking Module to check the files syntax, get the requirements information, conduct properties checking, and write check results into the output file. The sequence in which properties are checked will be explained in a later section. The access programs are described in Table 6.1.

Table 6.1: Access Programs of Master Control Module

Program Name	Arguments	Return	Description
PrctMctrlInit		void	Initializes this module
Continued on next page			

Table 6.1: (Continued)

Program Name	Arguments	Return	Description
PrctMctrlMain	FILE **frameworkfile* FILE **ttscontextfile* FILE **reportfile*	void	Sets *reportfile* as the results output file, gets the requirements information from *frameworkfile* and *ttscontexfile*, processes the properties checking by calling other modules

Framework File Information Module (PRCT_fwinfo.c)

This module provides programs to extract the static requirements information defined in a framework file and saves the information into related data structures. It also provides the programs to retrieve this information. Table 6.2 describes the access programs in Framework File Information Module.

Table 6.2: Access Programs of Framework File Information Module

Program Name	Arguments	Return	Description
FwinfoInit		void	Initializes this module
FwinfoExtract	File **frameworkfile*	void	Extracts the static requirements information defined in *frameworkfile* and saves this information in this module
FwinfoGetNumMov		int	Returns the number of monitored variables defined in the framework file
FwinfoGetMovList		mc_var *	Returns monitored variables list

Continued on next page

Table 6.2: (Continued)

Program Name	Arguments	Return	Description
FwinfoGetNumCov		int	Returns the number of controlled variables defined in the framework file
FwinfoGetCovList		mc_var *	Returns controlled variables list
FwinfoGetNumMoc		int	Returns the number of mode classes defined in the framework file
FwinfoGetMocList		modeclass *	Returns mode classes list
FwinfoGetNumValfun		int	Returns the number of controlled value functions defined in the framework file
FwinfoGetValfunList		valfun *	Returns controlled value functions list
FwinfoGetNumUserfun		int	Returns the number of user-defined-functions in the framework file
FwinfoGetUserfunList		userfun *	Returns user-defined-functions list
FwinfoGetNumCon		int	Returns the number of constants defined in the framework file
FwinfoGetConList		constant *	Returns constants list

TTS Context File Information Module (PRCT_cfinfo.c)

The module performs the operations on context and symbol table information contained in a TTS context file. It provides access programs called by other modules to retrieve information from TTS context file and make it available for use by other programs. Table 6.3 is the description of these access programs.

Table 6.3: Access Programs of TTS Context File Information Module

Program Name	Arguments	Return	Description
CfinfoInit		void	Initializes this module
CfinfoSetContextSymtbl	FILE **ttscontextfile*	void	Reads the TTS context file specified by *ttscontextfile* and saves the context handle and symbol table information in this module
CfinfoGetCurContext		CHandle	Returns the context handle of the TTS context file stored in this module
CfinfoGetCurSymtbl		SymTbl	Returns the symbol table of TTS context file stored in this module
CfinfoSetCurContext	CHandle *cur_ttscontext*	void	Sets *cur_ttscontext* as the context handle intended to be used by the module which calls this program
CfinfoSetCurSymtbl	SymTbl *cur_ttssymtbl*	void	Sets *cur_ttssymtbl* as the symbol table intended to be used by the module which calls the program

Variables Checking Module (PRCT_var.c)

The module is responsible for checking "No Useless Variables" and "No Undefined Variables" properties. Access programs in this module are described in Table 6.4.

Table 6.4: Access Programs of Variables Checking Module

Program Name	Arguments	Return	Description
VarInit		void	Initializes this module
VarCkVarListUse	reqid **reqvarlist* int *varlistlen*	void	Checks if the variables in the list *reqvarlist* are used in the tabular expressions in relevant TTS context file and saves the check results. The length of *reqvarlist* is specified by *varlistlen*.
VarGetNumUnusedVar		int	Returns the number of unused variables according to the stored check results information in this module
VarGetUnusedVarList	reqid **unusedlist*	void	Gets unused variables information and saves this in the list specified by *unusedlist*
VarOutputUnused-VarList		void	Writes check results of the "No Unused Variables" property into the output file
VarCkVarDef		void	Checks the "No Undefined Variables" property for all the tabular expressions in TTS context file and saves check results

Continued on next page

Table 6.4: (Continued)

Program Name	Arguments	Return	Description
VarGetNumUndefVar		int	Returns the number of undefined variables according to the stored check results information in this module
VarGetUndefVarList	undef_var **undeflist*	void	Gets the stored undefined variables information and saves them into the list specified by *undeflist*
VarOutputUndefVar		void	Writes check results of the "No Undefined Variables" property into the output file

Mode Classes Checking Module (PRCT_mdclass)

This module checks the "Non-terminal Initial Mode" and "No Isolated Mode" properties. It also checks if the relevant mode transition function is created properly. Table 6.5 describes the access programs in the module.

Table 6.5: Access Programs of Mode Classes Checking Module

Program Name	Arguments	Return	Description
MdclassInit		void	Initializes this module
MdclassCkMode	modeclass *moc*	void	Checks the "Non-terminal Initial Mode" and "No Isolated Modes" properties for modeclass *moc* and saves check results

Continued on next page

Table 6.5: (Continued)

Program Name	Arguments	Return	Description
MdclassCkAll		void	Checks the "Non-terminal Initial Mode" and "No Isolated Modes" properties for all the mode classes defined in a framework file and saves check results in this module
MdclassGetNumErr		int	Returns the number of mode classes where errors are found
MdclassGetErrList	err_mdclass **errmdclasslist*	void	Gets the stored check results information of mode classes and saves them into the list specified by *errmdclasslist*
MdclassOuput		void	Writes check results of mode classes into the output file

Controlled Value Functions Checking Module (PRCT_cvfun.c)

This module is responsible for checking the "Complete Definition" property. It accomplishes this task by analyzing the information about controlled value functions defined in the framework file and the expressions in TTS context file. Table 6.6 is the description of the access programs in this module.

Table 6.6: Access Programs of Controlled Value Functions Checking Module

Program Name	Arguments	Return	Description
CvfunInit		void	Initializes this module
Continued on next page			

Table 6.6: (Continued)

Program Name	Arguments	Return	Description
CvfunCkValfun		void	Checks if there is one and only one value function defined for each controlled variable and if this value function is properly contained in TTS context file, and saves check results
CvfunGetNumErrValfun		int	Returns the number of controlled variables that are defined in framework file and contain errors
CvfunGetErrValfunList	err_valfun **errvalfunlist*	void	Gets the stored complete definition property check results for defined controlled variables and saves them into the list specified by *errvalfunlist*
CvfunCkUndefCov		void	For each controlled value function, checks if the relevant controlled variable has been defined in framework file and saves the check results
CvfunGetNumUndefCov		int	Returns the number of undefined controlled variables in controlled value functions section
CvfunGetUndefCovList	valfun **errcovlist*	void	Gets the check results of undefined controlled variable and saves them into the list specified by *errcovlist*

Continued on next page

Table 6.6: (Continued)

Program Name	Arguments	Return	Description
CvfunOutput			Writes the check results of Complete Definition property into the output file

Circularity Checking Module (PRCT_cdef.c)

The module reads the information about controlled value functions and finds any circular definitions existing in the controlled variables. The detailed algorithm will be discussed in a later section. Access programs in this module are described in Table 6.7.

Table 6.7: Access Programs of Circularity Checking Module

Program Name	Arguments	Return	Description
CdefInit		void	Initializes this module
CdefCreateAdjMatrix		void	Analyzes all the value functions in TTS context file and creates the adjacency matrix of controlled variables reference relation
CdefSearchCirularPath		void	Searches the circular definition using the referenced-by relations information of adjacency matrix and saves the results in this module

Continued on next page

Table 6.7: (Continued)

Program Name	Arguments	Return	Description
CdefGetNumCircularPath	reqid *cov*	int	Returns the number of circular definition paths associated with controlled variable *cov*
CdefGetPathLen	reqid *cov* int *path_no*	int	Returns the length of a circular path associated with controlled variable *cov*, the circular path is specified by path number *path_no*
CdefGetCircularPath	reqid *cov* int *path_no* reqid **circularpath*	void	Gets a circular definition path associated with *cov*, saving this path in the list *circularpath*, the circular path is specified by path number *path_no*
CdefOutput		void	Writes the check results of the "Lack of Circularity" property into the output file
CdefFree		void	Frees the space allocated in this module; this program should be called when exiting this module

Parser Module (y.tab.c)

This module is generated by YACC from the grammar rules defined for framework file. It is a parser to do syntax analysis. The access program is described in Table 6.8.

Table 6.8: Access Programs of Parser Module

Program Name	Arguments	Return	Description
yyparse		void	Accepts the framework file and conducts syntax analysis, checking the properties of Proper Syntax, Initial Value and Initial Mode

Lexer Module (lex.yy.c)

This module is generated by LEX from the pattern matching rules given in the framework file, providing a lexer to do lexcial analysis. Table 6.9 describes the Access program.

Table 6.9: Access Programs of Lexer Module

Program Name	Arguments	Return	Description
yylex		void	Scans and matches strings in the framework file using patterns, converts the string to tokens for syntax analysis

Token Lookup Module (PRCT_tokenlookup.c)

This module detects the reserved words and the types of variables used in the framework file. The access program is described in Table 6.10.

Table 6.10: Access Programs of Token Lookup Module

Program Name	Arguments	Return	Description
TokenLookup	char **inputtext*	int	Matches the string *inputtext* extracted from framework file, determines it is a reserved word or a variable. Returns token ids for reserved words and types for variables

TTS Utilities Module (PRCT_ttsutil.c)

This module provides common programs to make processing tabular expressions more convenient. It calls the programs provided by Table Holder and other modules in TTS. Table 6.11 describes access programs in this module.

Table 6.11: Access Programs of TTS Utilities Module

Program Name	Arguments	Return	Description
TtsutilInit		void	Initializes this module
TtsutilInited		bool	Returns *BOOL_TRUE* if this module is initialized, otherwise, returns *BOOL_FALSE*
TtsutilGetSymName	SymTbl *ttssymtbl* Id *ttssymid* char **ttssymname*	void	Gets the name of symbol specified by *ttssymid* in *ttssymtbl*, and saves the name in *ttssymname*
TtsutilGetExpnName	CHandle *ttscontext* Expn *ttsexpn* char **ttxexpnname*	void	Gets the name of expression specified by *ttsexpn* in *ttscontext*, and saves the name in *ttsexpnname*

Continued on next page

Table 6.11: (Continued)

Program Name	Arguments	Return	Description
TtsutilGetExpnIdList	Expn *ttsexpn*	IdList	Returns the list of all Ids in *ttsexpn*
TtsutilExpnIsTbl	SymTbl *ttssymtbl* Expn *ttsexpn*	bool	Returns *BOOL_TRUE* if the expression specified by *ttsexpn* is a table in *ttssymtbl*, otherwise, returns *BOOL_FALSE*
TtsutilGetTblType	SymTbl *ttssymtbl* Expn *ttsexpn*	GTS_CCG	Returns the table type of expression *ttsexpn* in *ttssymtbl*
TtsutilGetGridIdList	Expn *ttsexpn* int *gridnum*	IdList	Returns the list of all Ids in a grid specified by *gridnum* in table *ttsexpn*
TtsutilGetValueGrid-Num	SymTbl *ttssymtbl* Expn *ttsexpn*	int	Returns the value grid number associated with table *ttsexpn* in *ttssymtbl*
TtsutilGetVecHeader-GridNum	SymTbl *ttssymtbl* Expn *ttsexpn*	int	Returns the header grid number associated with vector table *ttsexpn* in *ttssymtbl*
TtsutilGetExpnShape	Expn *ttsexpn*	Shape	Returns the shape of the expression specified by *ttsexpn*
TtsutilGetGrid-SubExpn	Expn *ttssexpn* int *gridnum* int *idx_one* int *idx_two*	Expn	Returns the id for a cell expression in the $gridnum^{th}$ grid of *ttsexpn*, the cell's index is specified by *idx_one* (*idx_two* is set 0) for one dimensional grid and by (*idx_one*,*idx_two*) for two dimensional grid

Continued on next page

Table 6.11: (Continued)

Program Name	Arguments	Return	Description
TtsutilIdUsedinExpn	SymTbl *ttssymtbl* reqid *idname* Expn *ttsexpn*	bool	Returns *BOOL_TRUE* if in *ttssymtbl* the object specified by *idname* is used in *ttsexpn*
TtsutilIdUsedinGrid	SymTbl *ttssymtbl* reqid *idname* Expn *ttsexpn* int *gridnum*	bool	Returns *BOOL_TRUE* if in *ttssymtbl* the object specified by *idname* is directly used in a grid of table *ttsexpn*, the grid is specified by *gridnum*
TtsutilExpnisDefined	CHandel *ttscontext* char **ttsexpnname*	bool	Returns *BOOL_TRUE* if the expression specified by *ttsexpnname* is defined in *ttscontext*
TtsutilGetExpnId	CHandel *ttscontext* char **ttsexpnname*	Expn	Returns the id of the expression specified by *ttsexpnname* in *ttscontext*

Result Reporting Module (PRCT_resultrpt.c)

This module is responsible for checking results output. It determines the output format and records the error number. Access programs in this module are described in Table 6.12. The interpretation of each error token is presented in Table 6.13.

Table 6.12: Access Programs of Results Reporting Module

Program Name	Arguments	Return	Description
ResrptInit		void	Initializes this module
ResrptSetErrfp	FILE **errorfp*	void	Sets the file pointed by *errorfp* as the results output file
Continued on next page			

Table 6.12: (Continued)

Program Name	Arguments	Return	Description
ResrptMsg	char *msg	void	Writes text message in msg to the output file
ResrptTokenInfo	Errrpt_Token $token$ char *$info$	void	Writes the check results represented by error token $token$ and the detailed information in $info$ to the output file
yyerror	char *msg	void	Writes the error message in msg to the output file, this program is called to output syntax error message
ResrptGetNumSyntaxErr		int	Returns the number of syntax errors found in a framework file
ResrptGetNumCheckErr		int	Returns the number of errors (not including syntax errors) found in the whole requirements document

Table 6.13: Check Results Tokens

Check Results Token	Interpretation
CKRES_UnUsedVariable	Ununsed variables found in requirments specification
CKRES_UndefVariable	Undefined variables used in TTS context file
CKRES_TranTblNotCreated	Mode transition function table is not created
CKRES_TranTblInvalid	Mode transition function is not a table
CKRES_TerminalIniMode	Initial mode is a terminal mode
CKRES_IsolatedMode	Isolated modes exist in mode classes
CKRES_ValfunUnDef	Value functions are not defined for controlled variables
Continued on next page	

Table 6.13: (Continued)

Check Results Token	Interpretation
CKRES_ValfunNotCreated	Value functions are not created
CKRES_ValfunMultiDef	Value functions for controlled variables are defined more than one
CKRES_ConVarUndef	Undefined controlled variable exist
CKRES_CircularDef	Circular definition found for controlled variables

Status Reporting Module (PRCT_statusrpt.c)

The Status Reporting Module provides a means for the access programs of other modules to indicate to their callers whether the operation is successful or not. If a failure occurs, it will give detailed information for the failure reason. The access programs in the module and the interpretation of each PRCT status token is described in Table 6.14 and Table 6.15 respectively.

Table 6.14: Access Programs of Status Reporting Module

Program Name	Arguments	Return	Description
SetErrREQ	PRCT_Token t	void	Sets the status token to t
GetErrREQ		REQ_Token	Returns the current status token
GetStrErrREQ	REQ_Token t	char *	Returns a descriptive string corresponding to the status token t
SetErrREQDetailMsg	char *msg	void	Sets the detailed message in msg for current status token
GetErrREQDetailMsg		char *	Returns the detailed message for current status token

Table 6.15: PRCT Status Tokens

Status Token	Interpretation
REQ_Success	No error
REQ_ModuleUnint	Module uninitialized
REQ_OpenTTSFileFail	Can not open TTS Context file
REQ_CFileLoadFail	Can not load TTS context file
REQ_CRegGetSymTblFail	Can not get symtabl table
REQ_InvailContext	Current context is invalid
REQ_TTSModuleUnInit	TTS utilities module is not initialized
REQ_GetSymNameFail	Can not get symbol name
REQ_GetSymTagFail	Can not get symbol tag
REQ_GetTableCCGFail	Can not get table cell connection graph
REQ_GetExpnNameFail	Can not get expression name
REQ_PathCreateFail	Can not create path
REQ_PathInsertIndexFail	Can not insert index into path
REQ_PathAssignIndexFail	Can not assign index to path
REQ_IndexCreateFail	Can not create index
REQ_IndexAssignGridFail	Can not assign grid number in index
REQ_IndexAssignEltFail	Can not assign element into index
REQ_GetExpnShapeFail	Can not get expression shape
REQ_GetExpnIdFail	Can not get expression id
REQ_GetExpnIdListFail	Can not get id list associated with expression
REQ_GetSubExpnFail	Can not get sub_expression id
REQ_GetIdListLenFail	Can not get the length of id list
REQ_GetIdListSymIdFail	Can not get symbol id from a id list
REQ_ListCreateFail	Can not create list
REQ_GetNumExpnFail	Can not get the number of expressions
REQ_GetNumDimFail	Can not get table dimension number
Continued on next page	

Table 6.15: (Continued)

Status Token	Interpretation
REQ_GTSInfoSetTblFail	Can not set symbol table
REQ_GTSSemVecHeaderFail	Can not get vector header grid number
REQ_NoValueGridDefined	Value grid is not defined in table
REQ_AllocFail	Memory allocation falied
REQ_GetExpnByPosFail	Can not get expression by position
REQ_MultiDimTbl	Multi-dimension table is used
REQ_UndefExpn	Expression is undefined
REQ_InvalidVariable	Variable name is invalid
REQ_InvalidCirularPathNo	Path number to CdefGetPathLen() or CdefGetCircularPath()is invalid
REQ_UnknownErrToken	Unkown type of error

6.2.4 Program-use hierarchy

The *Program Uses Relation* [26] is used to illustrated the detailed design of the preliminary requirement checking tool in terms of the uses relation between the access programs described in last section.

Definition 6.2.2 *Program Uses Relation*

Program A is said to use Program B if the correct behavior if Program A depends on the correct behavior of Program B.

Each program of this tool is identified by a program id. Table 6.16 gives the mapping between the program ids and the program names. The *Program Uses Relation* of PRCT is shown in Table 6.17.

Table 6.16: Access Programs of PRCT

Id	Program Name	Id	Program Name
1	PrctMctrlInit	2	PrctMctrlMain
3	FwinfoInit	4	FwinfoExtract
5	FwinfoGetNumMov	6	FwinfoGetMovList
7	FwinfoGetNumCov	8	FwinfoGetCovList
9	FwinfoGetNumMoc	10	FwinfoGetMocList
11	FwinfoGetNumValfun	12	FwinfoGetValfunList
13	FwinfoGetNumUserfun	14	FwinfoGetUserfunList
15	FwinfoGetNumCon	16	FwinfoGetConList
17	CfinfoInit	18	CfinfoSetContextSymtbl
19	CfinfoGetCurContext	20	CfinfoGetCurSymtbl
21	CfinfoSetCurContext	22	CfinfoSetCurSymtbl
23	VarInit	24	VarCkVarListUse
25	VarGetNumUnusedVar	26	VarGetUnusedVarList
27	VarOutputUnusedVar	28	VarCkVarDef
29	VarGetUndefVar	30	VarGetUndefVarList
31	VarOutputUndefVar	32	MdclassInit
33	MdclassCkMode	34	MdclassCkAll
35	MdclassGetNumErr	36	MdclassGetErrList
37	MdclassOuput	38	CvfunInit
39	CvfunCkValfun	40	CvfunGetNumErrValfun
41	CvfunGetErrValfunList	42	CvfunCkUndefCov
43	CvfunGetNumUndefCov	44	CvfunGetUndefCovList
45	CvfunOutput	46	CdefInit
47	CdefCreateAdjMatrix	48	CdefSearchCirularPath
49	CdefGetNumCircularPath	50	CdefGetPathLen
51	CdefGetCircularPath	52	CdefOutput
53	CdefFree	54	yyparse
55	yylex	56	TokenLookup

Continued on next page

Table 6.16: (Continued)

Id	Program Name	Id	Program Name
57	TtsutilInit	58	TtsutilInited
59	TtsutilGetExpnShape	60	TtsutilGetGridSubExpn
61	TtsutilGetSymName	62	TtsutilGetExpnName
63	TtsutilGetExpnIdList	64	TtsutilExpnIsTbl
65	TtsutilGetTblType	66	TtsutilGetGridIdList
67	TtsutilGetValueGridNum	68	TtsutilGetVecHeaderGridNum
69	TtsutilIdUsedinExpn	70	TtsutilIdUsedinGrid
71	TtsutilExpnisDefined	72	TtsutilGetExpnId
73	ResrptInit	74	ResrptSetErrfp
75	ResrptMsg	76	ResrptTokenInfo
77	yyerror	78	ResrptGetNumSyntaxErr
79	ResrptGetNumCheckErr	80	SetErrREQ
81	GetErrREQ	82	GetStrErrREQ
83	SetErrREQDetailMsg	84	GetErrREQDetailMsg

Table 6.17: Program Uses Relation

Program	Use Programs	Program	Use Programs
1	80	2	3, 4, 5, 6, 13, 14, 15, 16, 17, 18, 23, 24, 27, 28, 31, 32, 34, 37, 38, 39, 42, 45, 46, 47, 48, 52, 53, 57, 73, 74, 78, 79, 81, 82, 84
3	80	4	54, 80, 83
5	80, 83	6	80, 83
7	80, 83	8	80, 83
9	80, 83	10	80, 83

Continued on next page

Table 6.17: (Continued)

Program	Use Programs	Program	Use Programs
11	80, 83	12	80, 83
13	80, 83	14	80, 83
15	80, 83	16	80, 83
17	58, 80, 83	18	80, 83
19	80, 83	20	80, 83
21	80, 83	22	80, 83
23	5, 6, 13, 14, 15, 16, 19, 20, 58, 80, 83	24	80, 81, 83
25	80, 83	26	80, 83
27	75, 76, 80, 83	28	61, 62, 63, 80, 81, 83
29	80, 83	30	80, 83
31	75, 76, 80, 83	32	9, 10, 19, 20, 58, 80, 83
33	80, 81, 83	34	33, 80, 81, 83
35	80, 83	36	80, 83
37	75, 76, 80, 83	38	7, 8, 11, 12, 19, 20, 58, 80, 83
39	71, 80, 81, 83	40	80, 83
41	80, 83	42	80, 83
43	80, 83	44	80, 83
45	75, 76, 80, 83	46	7, 8, 11, 12, 19, 20, 58, 80, 83
47	80, 81, 83	48	80, 81, 83
49	80, 83	50	80, 83
51	80, 83	52	75, 76, 80, 83
53	80, 83	54	57, 77
55	58	56	
57	80	58	80
59	80	60	80
61	80	62	80, 83
63	80, 83	64	80, 83

Continued on next page

Table 6.17: (Continued)

Program	Use Programs	Program	Use Programs
65	80, 83	66	80, 83
67	80, 83	68	80, 83
69	80, 83	70	59, 80, 83
71	80, 83	72	80, 83
73	80	74	80, 83
75	80, 83	76	80, 83
77		78	80, 83
79	80, 83	80	
81		82	
83		84	

6.3 Algorithm

This section describes some algorithms used in the implementation of PRCT.

6.3.1 Overview

The input to PRCT are two files : a framework file and a TTS context File. Check results are written into an output file. The access program PrctMctrlMain() in Master Control Module uses the following sequence to invoke the access programs of other modules to accomplish the check tasks. Since some checks must have positive results before others can be made, the whole checking procedure may take several iterations.

1. Check the validity of framework File, TTS context File and output File. If they are not valid, give error messages and exit.

2. Initialize the Result Reporting Module; set the output file.

3. Initialize the Framework File Information Module; check the syntax of the framework file and extract file information. If any syntax error is found, write the errors into output file, give error messages and exit.

4. Initialize the TTS File Information Module; set the TTS Context File;

5. Initialize the Variables Checking Module; check the "No Unused Variables" and "No Undefined Variables" properties; write the check results into the output file.

6. Initialize the Mode Class Checking Module; check the mode transition table, "Non-terminal Initial Mode" and "No Isolated Mode" properties; write check results into the output file.

7. Initialize the Controlled Value Function Checking Module; check the "Complete Definition" property; write check results into the output file; If any error is found in this step, give error messages and exit;

8. Initialize the Circular Definition Checking Module; create the adjacency matrix for the controlled variables referenced-by relation; search circular definition paths of controlled variables. write the check results into the output file.

6.3.2 Algorithm for searching circular definition paths

In this part, we present the algorithms used to detect the circular definition paths existing in controlled variables referenced-by relation.

A directed graph G can be built based on the controlled variables referenced-by relation defined in chapter 5. Each controlled variable is represented as a node in G. The arcs in the graph are the referenced-by relations existing in the controlled variables. For example, if controlled variable v_j is referenced by v_i, there is an arc from v_i to v_j (See Figure 6.3).

An adjacency matrix can be used to represent the controlled variables referenced-by relations in G. Suppose there are n controlled variables in G stored in a list v, and function $expn$ is the value function of controlled variable v_i. The algorithm

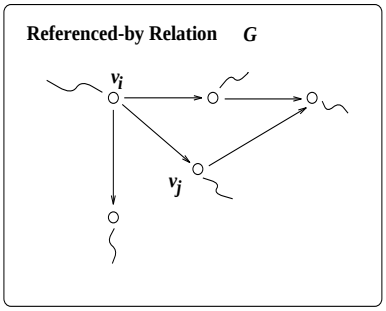

Figure 6.3: Reference Relation Graph

below describes how to set values associated with v_i in the adjacency matrix A of the graph G based the different types of *expn*. A is initialized with 0.

```
CreateAdjMatrixNode(i, expn) {

     if(expn is not a table)
     {
       for(all the other controlled variables v[j])
          if(v[j] is used in expn)
             A[i][j] = 1;
     }

     if( expn is normal table or expn is inverted table or
         expn is decision table )
     {
       for(all the other controlled variables v[j])
          if(v[j] is used in header H1 or H2 or main grid G of expn)
             A[i][j] = 1;
     }

     if(expn is vector table)
     {
       Get the vector header number of expn;
```

```
        for(all the other controlled variables v[j])
          if(v[j] is used in non-vector header or main grid G of expn)
              A[i][j] = 1;
      }
}/* end */
```

Given a node v_s in a directed graph G and the adjacency matrix A of G, the following algorithm describes how to find the circular paths that start from node v_s in G. This algorithm is based on the depth first search (DFS) algorithm for directed graphs. Starting from node v_s, when the search process reaches a node v_c from which there is an arc pointed to node v_s, since v_c is a child of the spanning tree of v_s, there is a circular path (cycle) from v_s to v_c. Based on the parent-child information recorded in the search process, function *ReverseSearch* generates the path and all the nodes on the path (See Figure 6.4).

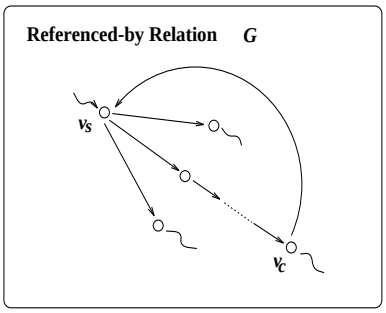

Figure 6.4: Circular Path in Reference Relation Graph

The *SearchCircularPath* function has three argument:

1. *start_node_no*: the start node number, which is the node number of v_s
2. *cur_visiting_no*: the current visiting node number, which is set to the same value as *start_node_no* when the function is called, since v_s is the first node to be visited
3. A: the adjacency matrix of Graph G

```
SearchCircularPath(start_node_no, cur_visiting_no, A)
{
  /* set current visiting node as visited node */
  v[cur_visiting_no].visited = BOOL_TRUE;

  /* look for the first adjacency node of current visiting node */
  next_no = FirstAdjNode(cur_visiting_no, A);

  while(next_no!=0) /* there exist adjacency nodes for current
                       visiting node */
  {
    if(next_no == start_node_no) /* the start node is a child
                                    of current visiting node,
                                    a cycle is found */
    {
          /* generate the circular path by looking at the
             parent-child information */
          ReverseSearch(start_node_no, cur_visiting_no, A);

          /* look for next adjacency node of current visiting node */
          next_no = NextAdjNode(cur_visiting_no,next_no, A);
    }
    else
    {
       if(!v[next_no].visited) /* the newly found adjacency node of current
                                  visiting node has not been visited */
       {
          /* Set current visiting node as the parent of the new found
             adjacency node */
          v[next_no].parent = cur_visiting_no;

          /* Search the branch tree of the new found adjacency node */
          SearchCircularPath(start_node_no, next_no, A);
```

```
          /* Get next adjacency node of current visiting node */
          next_no =  NextAdjNodeNo(cur_visiting_no,next_no, A);
       }
       else
          /* Skip the visited node, get a new adjacency node of current
             visiting node */
          next_no = NextAdjNode(cur_visiting_no,next_no, A);
    }
  }
}/* end */
```

Chapter 7

Results and Conclusion

This chapter summaries the results and the limitations of this thesis, proposes future work, and draws the conclusions.

7.1 Results

In this thesis, a Preliminary Requirements Checking Tool (PRCT) has been successfully developed for SCR tabular requirements. This tool provides support for the requirements process by checking the application-independent properties. We use both ASCII and tabular notations to represent SCR specification. In this thesis, SCR requirements documents are represented using a framework file and a TTS context file. PRCT checks the specification based on the information extracted from these files. We have defined the syntax of the framework file, and the application-independent properties derived from the underlying Four Variable Requirements Model and Generalized Tabular Specification Notation. The check results are written into an output file with detailed description, which allows users to analyze errors found in the requirements documents quickly and easily.

PRCT has been tested by using a simple requirements document for a bicycle computer, and several errors like unused variables, undefined controlled variables and isolated modes are found. More simple errors like this would be found for more complex systems where checking the huge amount of requirements information will

be tedious and time-consuming. Using this tool, preliminary requirements checking tasks will be performed automatically and efficiently.

Compared with other related works mentioned in chapter 2, this tool is distinguished in the following three major aspects:

1. Instead of just using three kinds of tables (condition table, event table, mode transition table), Generalized Tabular Notation is used in PRCT to specify SCR requirements documents. The condition table and event table are special cases of our inverted tables, and a mode transition table can be converted into normal or inverted table using generalized tabular notations. Generalized Tabular Notation is defined based on formal syntax and semantics model. Using generalized tabular notations enhances the express ability of requirements documents, and makes the specification more flexible.

2. In [4, 11, 12], a simplified SCR method was used to describe requirements, in which monitored variables and controlled variables can only have Boolean type values. We eliminate this restriction in our requirements. So the specification ability is improved and more powerful type-checking can be done based on this.

3. PRCT is based on previous work on the TTS. Table Holder and other modules in TTS provide the programs that make processing tabular notations easily and efficiently. By using TCT (which will be replaced by another table input tool in the future) to create tables, we need not define the syntax for tabular expressions, and get a simplified grammar to describe requirements documents. In addition, TTS also provide some checking tools for separate tables, like Type Checking Tool and Table Checking Tool, which are under development and modified. By integrating them into PRCT, PRCT can take advantage of the checking abilities of these tools and provide more powerful checks for SCR requirements documents.

7.2 Limitations and future work

Like any other practical tools, limitations exist in PRCT. Future work will eliminate these limitations, improve and enhance the checking abilities of this tool.

Currently, PRCT can not proceed type-checking and can not check tables' "Completeness and Consistency" properties. These abilities will be added in the future by using other tools of TTS. A *Type Checking Tool* is under development presently. By using this type checking tool, user can define types they need, and we can not only check variables and constants, but also functions, predicate expressions, and predicate rule and relation rule of tables. The abilities to check completeness and consistency can also be added into PRCT after the improvement and integration of *Table Check Tool* is finished. As discussed in chapter 5, the Properties Checking Module is designed to meet the anticipated changes of adding new sub-modules to check new properties. So the work to integrate TTS tools and expand the checking abilities of PRCT can be done easily and cleanly without worrying about redesigning the module structure of PRCT and changing other modules.

Another limitation is about the tables used in this thesis. Presently, PRCT can only process the tables that have one two-dimensional main grid indexed by two one-dimensional headers. This limitation is due to the capabilities of TTS and the table models of generalized tabular notations. This problem is expected to be solved with the development of TTS and generalized table notations models.

At present, a user can use any text editor to compose the framework file using the defined syntax. This process could be tedious and error-prone when systems become large and complicated. A *specification editor* would be developed in the future with graphical user interface. This editor serves as an auxiliary tool to facilitate the requirements specification. Users need not be familiar with the representation syntax then, and the framework file can be produced automatically. Also, a graphical user interface for PRCT would be developed later to make use of this tool more convenient.

Another area of future work is not just restricted to PRCT itself. In this thesis, we have developed a tool to support automatically check preliminary application-independent properties of requirements. However, this is just a part of the whole requirements process. In [15], Heitmeyer proposed an idealization of a real-world process for requirements specification.

1. A formal notation is used to specify the requirements.
2. An automated requirements checker is used to checking the application-independent properties of the specification, such as syntax, type correctness etc.
3. A simulator is used to execute the specification symbolically to ensure that it captures the customers' intent.
4. Mechanical support is used to analyze the specification for application properties in the later stages of the requirements phase.

The work in this thesis is focused in the first two steps. Some related work on the last two steps has been discussed in chapter 4. In the future work, we will analyze the deeper application properties and develop tools to check these properties for SCR requirements specified with generalized tabular notations. The two parts will be combined together eventually. PRCT will work as a preprocessor for the application properties checking tools. The requirements that pass PRCT will serve as input to the advanced tools to check critical properties.

7.3 Conclusions

Requirements play significant role in the whole life cycle of software development. Automatically detecting requirements errors in early phase by tools are crucial for improving the quality of requirements. In this thesis, we have successfully developed a Preliminary Requirements Checking Tool (PRCT) for this purpose, which makes it possible to automatically check application-independent properties for SCR requirements documents. PRCT can relieve reviewers from the tedious and error-prone tasks and requirements can be processed more quickly and efficiently.

In addition, the module structure of this tool is designed to meet the anticipated changes by using information hiding and separation of concerns principles, which makes the future extension and improvement to this tool easier. The development of PRCT demonstrated the importance of applying software engineering principles in software development.

Appendix A

Requirements of a Simple Bicycle Cyclometer System

This appendix gives the requirements of a simple bicycle cyclometer system, which has been used to test PRCT. The first section describes the framework file - Cyclometer.req; the second section shows the TTS context file - Cyclometer.tts, which contains all the tabular expressions used in Cyclometer.req.

A.1 Framework File - Cyclometer.req

```
/* Example Requirements Document - Bicylce Cyclometer */

/* mov_: Monitored Variable;  cov_: Controlled Variable;
 * moc_: Mode Class;          mod_: Mode;
 * fun_: Function;            con_: Constant;
 * tts_: TTS Table;
 */

/* Monitored Variables */

MONITORED float mov_CUR_SP,     mov_ELAPSED_TIME;
```

```
          bool  mov_LB_PRESSED, mov_RB_PRESSED,
                mov_KM_UNIT_ON;

/* Controlled Variables */

CONTROLLED float cov_CUR_SP = 0,          cov_AVE_SP = 0,
                 cov_TRIP_DISTANCE = 0,   cov_MAX_SP = 0,
                 cov_TOTAL_DISTANCE = 0, cov_ELAPSED_TIME= 0;
           bool  cov_PACEARROW_ON = FALSE;

/* Mode Classes */

MODECLASS   moc_BICYCLE;

INITIAL_MODE    mod_INITIAL;

MODE            mod_INITIAL,      mod_SPEED_V_AV,
                mod_SPEED_V_MAX,  mod_DISTANCE,
                mod_SET_KM_ML;

TRANSITION_FUNCTION tts_mode_tran;

/* Controlled Value Functions */

CONTROLLED_VALUE_FUNCTION

        cov_CUR_SP:             tts_cur_sp,
        cov_AVE_SP:             tts_ave_sp,
        cov_MAX_SP:             tts_max_sp,
        cov_PACEARROW_ON:       tts_pacearrow_on,
        cov_TRIP_DISTANCE:      tts_trip_distance,
        cov_TOTAL_DISTANCE:     tts_total_distance,
        cov_ELAPSED_TIME:       tts_elapsed_time;
```

```
/* Dictionary */

DICTIONARY

FUNCTION   float fun_Convert(float,bool): tts_convert,
           float fun_Max(float,float): tts_max;

CONSTANT   float con_MLtoKM = 0.667;   int con_INTERVAL = 3;
```

A.2 TTS Context File - Cyclometer.tts

tts_mode_tran defined

	@T(mov_LB_PRESSED = true) ∧ (mod_RB_PRESSED = false)	@T(mov_RB_PRESSED = true) ∧ (mod_LB_PRESSED = false)	@T(mov_LB_PRESSED = true) ∧ @T(mod_RB_PRESSED = true)
mod_SPEED_V_AV	mod_SPEED_V_MAX	mod_DISTANCE	*
mod_SPEED_V_MAX	mod_SPEED_V_AV	mod_DISTANCE	*
mod_DISTANCE	mod_SPEED_V_AV	mod_SPEED_V_MAX	*
mod_SET_KM_ML	*	*	mod_DISTANCE
mod_INITIAL	mod_DISTANCE	mod_SPEED_V_AV	*

tts_cur_sp defined

	true
cov_CUR_SP'=	fun_Convert(‘mov_CUR_SP,‘mov_KM_UNIT_ON)

tts_ave_sp defined

	mod_INITIAL	¬(mod_INITIAL)
cov_AVE_SP'=	0	fun_Convert((‘cov_TRIP_DISTANCE / ‘cov_ELAPSED_TIME),‘mov_KM_UNIT_ON)

tts_max_sp defined

	mod_INITIAL	¬(mod_INITIAL)
cov_MAX_SP'=	0	fun_Convert(fun_Max(‘cov_MAX_SP, ‘cov_CUR_SP),‘mov_KM_UNIT_ON)

tts_pacearrow_on defined

	‘cov_CUR_SP ≥ ‘cov_AVE_SP	‘cov_CUR_SP < ‘cov_AVE_SP
cov_PACEARROW_ON'=	true	false

tts_trip_distance defined

	mod_INITIAL	¬(mod_INITIAL)
cov_TRIP_DISTANCE'=	0	fun_Convert((‘cov_TRIP_DISTANCE + (‘cov_CUR_SP × ‘cov_INTERVAL)), ‘mov_KM_UNIT_ON)

tts_total_distance defined

	mod_INITIAL	¬(mod_INITIAL)
cov_TOTAL_DISTANCE'=	0	fun_Convert((‘cov_TOTAL_DISTANCE + ‘cov_TRIP_DISTANCE),‘mov_KM_UNIT_ON)

tts_elapsed_time defined

	true
cov_ELAPSED_TIME'=	‘mov_ELAPSED_TIME

fun_Convert(float arg_length,bool arg_km_unit_on) defined

	@T(arg_km_unit_on=true)	@F(arg_km_unit_on=true)
true	‘arg_length / con_MLtoKM	‘arg_length × con_MLtoKM

fun_Max(float a,float b) defined

	‘arg_a ≥ ‘arg_b	‘arg_a < ‘arg_b
true	‘arg_a	‘arg_b

Bibliography

[1] Ruth Abraham, Evaluating Generalized Tabular Expressions in Software Documentation, *CRL Report No. 346*, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ont. February 1997

[2] Joanne M. Atlee and John Gannon, State-Based Model Checking of Event-Driven Systems Requirements, *IEEE Transactions on Software Engineering*, Vol. 19, No. 1 January 1993, pp. 24-40. (A version of this paper appeared in *Proceedings of ACM Conference on Software for Critical Systems*, December 1991, pp. 16-28.)

[3] Joanne M. Atlee, *Automated Analysis of Software Requirements, Ph.D. thesis*, University of Maryland, College Park, Maryland, Dec. 1992

[4] Joanne M. Atlee and John Gannon, Analyzing Timing Requirements, In *Proceedings of the International Symposium on Software Testing and Analysis*, June 1993, pp. 117-127

[5] Joanne M. Atlee, Marsha Chechik, John Gannon, Using Model Checking to Analyze Requirements and Designs. *Advances in Computers*, Volume 43, Marv Zelkowitz, ed. Academic Press, 1996

[6] T.E. Bell and T.A.Thayer, Software Requirements: Are They Really a Problem?, In *Proc ICSE-2: 2nd International Conference on Software Engineering*, San Francisco, 1976, 61-68

[7] R.Bharadwaj and C.L. Heitmeyer, Applying the SCR Requirements Specification Method to Practical Systems: A Case Study, In *Proc. Software Engineering Workshop*, NASA Goddard Space Flight Center, 1996

[8] R.Bharadwaj and C.L. Heitmeyer, Verifying SCR requirements specifications using state exploration. In *Proceedings of First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997

[9] Tevfik Bultan, Richard Gerber and Christopher League, Composite Model Checking: Verification with Type-Specific Symbolic Representations. *ACM Transactions on Software Engineering and Methodology.* Volume 9, Number 1, Pages 3-50, January 2000. (A version of this paper appeared in *ISSTA '98*).

[10] D. Craigen et al. An international Survey of Industrial Application of Formal Methods, *Technical Report NRL-9581*, NRL. Wah., DC, 1993

[11] Marsha Chechik, *Automatic Analysis of Consistency between Requirements and Designs, Ph.D. Thesis*, University of Maryland, College Park, Maryland, 1996

[12] Marsha Chechik, SC(R)3: Toward Usability of Formal Methods, In *Proceedings of CASCON*, November 1998

[13] Steve Eastebrook, Robyn Lutz, et al. Experiences Using Lightweight Formal Methods for Requirements Modeling, *IEEE Transactions on Software Engineering, Vol. 24. No. 1*, January 1998

[14] C.L. Heitmeyer et al. SCR*: A Toolset for Specifying and Analyzing Requirements, In *Proc. 10th Annual Conf. On Computer Assurance (COMPASS'95)* June 1995

[15] C.L. Heitmeyer, R.D. Jeffords, B. G. Labaw, Automated Consistency Checking of Requirements Specifications, In *ACM Trans. On Software Eng. And Methodology*, 5,3, July 1996, 231-261

[16] C.L. Heitmeyer, et al. SCR*: A Toolset for Specifying and Analyzing Software Requirements, In *Proc. Computer-Aided Verification, 10th Ann. Conf.* Vancouver, Canada, 1998.

[17] K.L. Heninger, D.L.Parnas, J.Kallander, Software Requirements for the A-7E Aircraft, *Tech. Rep. MR 3876*, Naval Ressearch Laboratory, 1978

[18] K.L. Heninger, Specifying Software Requirements for Complex Systems: New Techniques and Their Application, *IEEE Trans. Software Eng. SE-6(1): 2-13*, Jan. 1980

[19] Junhua Hu, *Use of Aliases in Tabular Expressions, M.Eng. Thesis*, McMaster University, Hamilton, 1999

[20] R. Janicki, Towards a Fromal Semantics of Parnas Tables, In *17th International Conference on Software Engineering*, IEEE Computer Society, Seattle, WA, April 1995, pp. 231-240

[21] R. Janicki, R. Khedri, On a Fromal Semantics of Tabular Expressions, *Science of Computer Programming*, Vol. 39, 2001, pp. 189 - 213

[22] Axel van Lamsweerde, Requirements Engineering in the Year 00: A Research Perspective, To be appear in *Proc. 22nd International Conference on Software Engineering*, Limerick, June 2000, ACM Press

[23] R.R. Lutz, Targeting Safety-related errors during software requirements analysis, In *Proc. 1st ACM SIGSOFT symp. On Foundations of Software Eng.* Los Angles, CA, Dec. 1993

[24] Robyn R. Lutz, Yoko Ampo, Experiences Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software, In *Proc. Of the 19th Annual Software Engineering Workshop*, Goddar Space Flight Center, Dec. 1994, SEL-94-006.

[25] D.L.Parnas, On the Criteria to be Used in Decomposing Sustems into Modules, *Commun ACM*, vol 15, December 1972, pp. 1052-1058

[26] D.L.Parnas, On a 'Buzzword': Hierachical Structure, *Proceedings of the IFIP Congress 1974*, 1974, pp. 336-339

[27] D.L.Parnas, Tabular Representation of Relations, *Tech. Rep. CRL-260* Telecommunications Research Institute Of Ontario (TRIO), McMaster Univ. Hamilton, ON, Canada, Oct. 1992

[28] D.L.Parnas, Some Theorems We Need Prove. In *Proc. 1993 Int'l Conf. On HOL Theorem Proving and Its Applications*, Vancouver, BC, Canada, Aug. 1993, pp. 155-162

[29] D.L. Parnas and J. Madey, Functional Documentation for Computer Systems, *Science of Computer Programming*, Vol. 25, No. 1, Oct. 1995, pp. 41 - 61

[30] D.K. Peters, *Deriving Real-Time Monitors From System Requirements Documentation, Ph.D. Thesis*, McMaster University, Hamilton, 2000

[31] Software Engineering Research Group, Table Tool System Developer's Guide, *CRL Report 339*, Communications Research Laboratory, McMaster University, Hamilton Ontario, Canada, Jan. 1997

[32] Tirumale Sreemani, Joanne M. Atlee, Feasibility of Model Checking Software Requirements: A Case Study, In *Proceedings of the 11th Annual Conference on Computer Assurance*, June 1996.