

Implementation of Tabular Verification and Refinement

IMPLEMENTATION OF TABULAR VERIFICATION AND REFINEMENT

By

NING ZHOU, B.Eng.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

McMaster University

©Copyright by Ning Zhou, February 2007

MASTER OF SCIENCE(2006)
COMPUTING AND SOFTWARE

McMaster University
Hamilton, Ontario

TITLE: Implementation of Tabular Verification and Refinement

AUTHOR: Ning Zhou, B.Eng.

SUPERVISOR: Dr. Emil Sekerinski

NUMBER OF PAGES: xiv, 195

Abstract

It has been argued for some time that tabular representations of formal specifications can help in writing them, in understanding them, and in checking them. Recently it has been suggested that tabular representations also help in breaking down large verification and refinement conditions into a number of smaller ones.

The article [32] developed the theory, but the real *proof* in terms of an implementation is not provided. This project is about formalizing tables in a theorem prover, Simplify, defining theorems of [32] in terms of functions written in the OCaml programming language, and conducting some case studies in verifying and refining realistic problems.

A parser is designed to ease our job of inputting expressions. Pretty-print is also provided: all predicates and tables of the examples in our thesis are automatically generated.

Our first example is a control system, a luxury sedan car seat. This example gives us an overall impression on how to prove correctness from tabular specification. The second example specifies a visitor information system. The design features of this example involve modeling properties and operations on sets, relations and functions by building self-defined axioms. The third example illustrates another control system, an elevator. Theorems of algorithmic refinements, stepwise data refinements, and the combination of algorithmic abstraction and data abstraction are applied correspondingly to different operations.

Acknowledgements

First and foremost I would like to sincerely thank my supervisor, Dr. Emil Sekerinski, for his insightful guidance, great patience and generous support; for his careful review and corrections to the draft of this thesis.

I am also grateful to the professors of my graduate courses, and my classmates, for making this a fun and enjoyable learning experience.

Especially, I am indebted to my parents and my daughter Karen Song. To them, I dedicate this thesis.

Contents

| | |
|---|-------------|
| List of Figures | xi |
| List of Tables | xiii |
| 1 Introduction | 1 |
| 1.1 Formal Manipulation of Tabular Expressions | 1 |
| 1.2 Contributions | 1 |
| 1.3 Structure of the Thesis | 3 |
| 2 Related Work | 5 |
| 2.1 Application of Tabular Specification | 5 |
| 2.1.1 Navy’s A-7 Aircraft Documentation | 5 |
| 2.1.2 Functional Tables Applied in Shutdown Systems | 7 |
| 2.1.3 Documentation of Non-deterministic Programs | 8 |
| 2.1.4 An Example of the Use of Tables in System Documentation | 9 |
| 2.2 Syntax, Semantics, and Transformations of Ten Kinds of Tables | 11 |
| 2.3 Automatic Tabular Documentation Tools | 17 |
| 2.3.1 RSML for Process-Control Systems | 17 |
| 2.3.2 Model Checking | 18 |
| 2.3.3 SCR* | 19 |

| | | |
|----------|---|-----------|
| 2.3.4 | Tablewise | 21 |
| 2.3.5 | TABLE Construct in PVS | 22 |
| 2.4 | Table Tool System | 25 |
| 3 | Tabular Verification and Refinement Overview | 27 |
| 3.1 | Terminology and Notation | 27 |
| 3.1.1 | Properties of Vectors | 27 |
| 3.1.2 | Relations | 28 |
| 3.1.3 | Relational Operation | 28 |
| 3.1.4 | Precondition and Weakest Precondition | 29 |
| 3.2 | Tabular Predicates | 31 |
| 3.3 | Operations on Tabular Predicates | 33 |
| 3.4 | Tabular Relations | 35 |
| 3.5 | Tabular Verification | 36 |
| 3.6 | Refinement | 39 |
| 3.7 | Tabular Refinement | 43 |
| 4 | Design Features | 45 |
| 4.1 | Interface with Simplify | 45 |
| 4.2 | Pattern Matching in Function Definition | 47 |
| 4.3 | Unified Data Type | 49 |
| 4.4 | Variable Types of Theorems | 50 |
| 4.5 | Structure of Implementation | 52 |
| 4.5.1 | Parser | 52 |
| 4.5.2 | Notations for Sets and Relations | 54 |
| 4.5.3 | Printing | 57 |
| 4.5.4 | Theorem Proving | 90 |

| | | |
|----------|--|------------|
| 5 | Implementation of Theorems | 99 |
| 5.1 | Principle of Proof | 99 |
| 5.2 | Implementation of Tabular Transformation | 100 |
| 5.3 | Implementation of Operations on Tabular Predicates | 104 |
| 5.4 | Proof of Precondition and Weakest Precondition | 106 |
| 5.5 | Implementation of Verification with Predicates | 108 |
| 5.6 | Implementation of Refinement | 110 |
| 5.6.1 | Implementation of Algorithmic Refinement | 110 |
| 5.6.2 | Implementation of Data Refinement | 111 |
| 6 | Luxury Sedan Car Seat Case Study | 117 |
| 6.1 | Requirement | 118 |
| 6.2 | Organization | 121 |
| 6.3 | Normal Mode and Its Properties | 124 |
| 6.3.1 | Types of Motor Adjustment Buttons | 124 |
| 6.3.2 | Releasing Motor Adjustment Buttons | 124 |
| 6.3.3 | Pressing Group 1 Motor Adjustment Buttons | 125 |
| 6.3.4 | Pressing Group 2 Motor Adjustment Buttons | 129 |
| 6.4 | Memory and Memory Set Mode | 133 |
| 6.4.1 | Memory Mode and Its Properties | 133 |
| 6.4.2 | Memory Set Mode | 138 |
| 6.5 | Calibration Mode and Its Properties | 139 |
| 6.6 | Summary | 143 |
| 7 | Modeling a Visitor Information System | 145 |
| 7.1 | Example Introduction | 145 |
| 7.2 | Specification of Visitor Information System | 148 |

| | | |
|----------|--|------------|
| 7.3 | Axiom | 152 |
| 7.4 | Proving Invariants and Preconditions | 158 |
| 7.5 | Conclusion | 165 |
| 8 | Elevator Control Refinement | 167 |
| 8.1 | Controlling Elevators | 167 |
| 8.2 | Call Button Pressed Abstraction | 168 |
| 8.2.1 | Case Introduction | 168 |
| 8.2.2 | Axioms | 169 |
| 8.2.3 | Result and Further Simplification | 170 |
| 8.3 | Elevator Scheduling Refinement | 173 |
| 8.3.1 | Case Introduction | 173 |
| 8.3.2 | Stepwise Data Refinement | 175 |
| 8.4 | Summary | 186 |
| 9 | Conclusion and Future Work | 189 |

List of Figures

| | | |
|------|---|----|
| 2.1 | The Dialogue Box | 11 |
| 2.2 | Command Mode Table for Dialogue Box | 12 |
| 2.3 | State Mode Table for Dialogue Box | 12 |
| 2.4 | A Normal Function Table | 13 |
| 2.5 | An Inverted Function Table | 13 |
| 2.6 | A Vector Function Table | 13 |
| 2.7 | A Normal Relation Table | 14 |
| 2.8 | An Inverted Relation Table | 14 |
| 2.9 | A Vector Relation Table | 14 |
| 2.10 | A Mixed Vector Table | 15 |
| 2.11 | A Predicate Expression Table | 15 |
| 2.12 | A Characteristic Predicate Table | 16 |
| 2.13 | A Generalized Decision Table | 16 |
| 2.14 | An AND/OR Table | 18 |
| 2.15 | WaterPres Mode Transition Table for Press | 20 |
| 2.16 | One-Dimensional Vertical and Horizontal Tables | 23 |
| 2.17 | A Two-Dimensional Table and Its Corresponding Enumeration Table | 23 |
| 2.18 | The SRC Table Represented in PVS | 24 |

| | | |
|-----|--|-----|
| 6.1 | Controlling a Luxury Sedan Car Seat | 118 |
| 6.2 | Performance of Proof in Car Seat Control | 144 |
| 7.1 | Performance of Proof in Visitor Information System | 166 |
| 8.1 | Performance of Proof in Elevator Control | 187 |

List of Tables

| | | |
|------|--|-----|
| 2.1 | Condition Table, Magnetic heading (//MAGHDGR//) output value . | 6 |
| 2.2 | Event Table, when AUTOCAL Light switched on/off | 7 |
| 2.3 | A Mode Transition Table | 19 |
| 2.4 | A Simple Decision Table | 21 |
| 4.1 | Logical Operators and Quantifiers | 53 |
| 4.2 | A Sample Table | 54 |
| 4.3 | A Sample Vector Table | 54 |
| 6.1 | Variables in Car Seat Control | 123 |
| 6.2 | LA Motor Button Pressed <i>lapressed</i> | 125 |
| 6.3 | RH Motor Button Pressed <i>rhpressed</i> | 128 |
| 6.4 | B Motor Button Pressed <i>bpressed</i> | 129 |
| 6.5 | FH Motor Button Pressed <i>fhpressed</i> | 130 |
| 6.6 | FH Motor Button Pressed <i>fhpressed</i> | 132 |
| 6.7 | LA Motor Moving to Its Setting <i>latoset</i> | 133 |
| 6.8 | RH Motor Moving to Its Setting <i>rhtoset</i> | 134 |
| 6.9 | B Motor Moving to Its Setting <i>btoiset</i> | 134 |
| 6.10 | FH Motor Moving to Its Setting <i>fhtoset</i> | 135 |
| 6.11 | HR Motor Moving to Its Setting <i>hrtoset</i> | 135 |

| | | |
|------|--|-----|
| 6.12 | Memory Mode Movement <i>memory</i> | 136 |
| 6.13 | Memory Set Mode | 139 |
| 6.14 | LA Motor Moving to Its Home <i>latohome</i> | 139 |
| 6.15 | RH Motor Moving to Its Home <i>rhtohome</i> | 140 |
| 6.16 | B Motor Moving to Its Home <i>btohome</i> | 140 |
| 6.17 | FH Motor Moving to Its Home <i>fhtohome</i> | 141 |
| 6.18 | HR Motor Moving to Its Home <i>hrtohome</i> | 141 |
| 7.1 | Checking Meeting Attended <i>visitorInfo</i> | 150 |
| 7.2 | Conference Room <i>conferenceRooms</i> | 151 |
| 7.3 | Dining Room <i>diningRooms</i> | 151 |
| 8.5 | Encoding Relation $R(rs)(reqs)$ | 177 |

Chapter 1

Introduction

1.1 Formal Manipulation of Tabular Expressions

The use of tabular expressions in the specification of programs is motivated by the discontinuous input/output behavior of programs. Sekerinski continues this line by deriving a number of theorems to support formal manipulations on tables in [32]. Many of the theorems have an intuitive interpretation, but the side conditions which are also derived are less obvious.

The expressions which is dealing with are predicates—as they allow an abstract specification of the input/output behavior—and relations—as they model non-deterministic programs. Our project is the implementation of the theorems in [32]. The definition of precondition in [34] is also applied in our thesis.

1.2 Contributions

Our contributions in the research of tabular specification and refinement include:

- Define a unified data type *form*, based on which all first-order logic predicates,

tables, set properties and operations, and relations and functions in terms of sets of pairs are expressed.

- Design a small parser to ease the work of inputting expressions directly by programmers who develop examples.
- Implement a function to pretty print any expression of *form*, including tables, by its ASCII character to screen; implement a function for printing expressions to L^AT_EX files.
- The side conditions are checked before implementing theorems. The results are printed by S-expressions which can be recognized and validated by Simplify.
- Develop examples and case studies to apply theorems. Build axioms for types and properties in each example.
- The proofs of *invariants*, *weakest precondition*, *precondition*, *algorithmic refinement*, *data refinement* are illustrated.

Our work provides a demonstration on how tabular predicates and tabular relations help in formal manipulations that occur in the process of verifying and refining specifications. We argue that

1. the structure of tables leads to a natural way of decomposing their manipulation; and
2. tabular manipulation rules are easier to memorize and apply than their textual counterparts.

We illustrate these claims with three examples. It turns out that some of the theorems are applied frequently and others are less useful.

Our adoption of OCaml as our programming language is based on the fact that ML was designed for theorem proving [28]. Objective Caml is an implementation of the ML language, based on the Caml Light dialect extended with a complete class-based object system and a powerful module system in the style of Standard ML [5]. It adds object-oriented features into the functional programming language and produces higher performance.

We send our formulae to the automatic theorem prover Simplify for validation. Simplify is the proof engine of the Extended Checkers ESC/Java and ESC/Modula-3. The input to Simplify is a formula of untyped first-order logic with function symbols and equality [7].

1.3 Structure of the Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 introduces related work on applications of tables in software engineering and table tool systems.
- Chapter 3 gives an introduction to the design features of our project. It includes the interface of our program with Simplify, pattern matching in function definition, types, and structure of implementation.
- Chapter 4 models theorems in terms of functions. It first explains the principle of the proof, then selects some typical theorems from each section in [32] to demonstrate the proof.
- Chapter 5 studies the case of a Luxury Sedan Car Seat in its specification and verification by applying functions of Chapter 4.

- Chapter 6 illustrates an information management system where axioms about sets, functions and relations are used.
- Chapter 7 gives an elevator example to show algorithmic and data refining operations in control systems. Call button pressed operation is abstracted to a state space including two boolean variables. Elevator scheduling is specified by an abstract relation. It is then stepwise refined to our specification which executes the elevator algorithm.
- Chapter 8 draws conclusions from our work, in addition to discussing future work.

Chapter 2

Related Work

Tables support a *divide and conquer* approach to understanding a complicated question by breaking large amounts of information into several small parts. Illuminated by their application in an aircraft project to specify the controls, the use of tables for software documentation is formalized and tabular expressions are defined by logical expressions and conventional mathematical formulae.

2.1 Application of Tabular Specification

2.1.1 Navy's A-7 Aircraft Documentation

The first application of this technique was documenting the requirements of existing flight software for the Navy's A-7 aircraft [14]. When the *Naval Research Laboratory* and the *Naval Weapons Center* rebuild the operational flight program, software engineering techniques were not suitable for this project because of stringent resource limitations for the old program. New techniques allowed them to achieve completeness, precision, and clarity within a 500-page document. Function characteristics described in tables make it easy to find answers to specific questions and to

| MODE | CONDITIONS | |
|---------------------------------|---|-----------------------|
| *DIG*, *DI*, *I+Wag sl*, *Grid* | Always | X |
| *IMS fail* | (NOT /IMSOMODE/=\$Offnone\$) | /IMSMODE/=\$Offnone\$ |
| //MAGHDGR// value | angle defined by /MAGHCOS/ and /MAGHSIN/ | 0 (North) |

Table 2.1: Condition Table, Magnetic heading (//MAGHDGR//) output value

detect gaps and inconsistencies in specifications. Two kinds of tables are introduced: *condition tables*, which are used to define some aspect of an output value that is determined by an active mode, and *event table*, which show when demand functions¹ should be performed or when periodic functions² should be started or stopped.

Table 2.1 gives an example of a condition table. Each row corresponds to a group of one or more modes in which this function acts like. Each condition column at one row characterizes the time intervals within a mode, so it must be exclusive to the other columns in the same mode. All condition columns together at one row describe the entire time the program is executing within a mode, so they must be complete. To find the information appropriate for a given mode and given condition, first find the row corresponding to the mode, find the condition within the row, and follow that column to the bottom of the table. An "X" instead of a condition indicates that information at the bottom of the column is never appropriate for that mode. With the condition table, one can easily check any inconsistency of the periodic functions.

Each row in an event table corresponds to a mode or group of modes. Table entries are events that cause an action to be taken when the system is in a mode associated with the row. At the bottom of the column is the action triggered by that event.

The event table in Table 2.2 specifies that the autocalibration light controlled by output data item //AUTOCAL// be turned on when the two listed modes are entered

¹performed differently in different event request.

²performed differently in different time intervals.

| MODE | EVENTS | |
|------------|---------------------|----------------------|
| *Lautocal* | @T(In mode) | @F(In mode) |
| *Sautocal* | | |
| ACTION | //AUTOCAL//:=\$On\$ | //AUTOCAL//:=\$Off\$ |

Table 2.2: Event Table, when AUTOCAL Light switched on/off

and off when they are exited. Symbol “:=” is used to denote assignment. The event @T(In mode) occurs when all the conditions represented by the mode become true, i.e, when the mode is entered. @F(In mode) occurs when any one of the conditions represented by the mode becomes false, i.e., when the system changes to a different mode.

Formulating questions with tabular notation in this program helps to separate concerns before trying to answer them.

2.1.2 Functional Tables Applied in Shutdown Systems

Program functions describe the precise effect of a deterministic program without describing the intermediate states. The upper header of a program *function table* consists of predicates partitioning the function’s domain, while the left header contains program variable names. The entries store the final variable values in corresponding conditions. An example in [25] shows a simple program and the tabular expression of the relational specification of that program. The tables ease the expressions in that:

1. Tables reveal the intended structure of the expression.
2. Tables replace repetition parts of the subexpression with a single name.
3. Because each table entry only applies to a small part of the function’s domain, the expression in that entry can be simplified.

A large safety-critical project using program-function tables was the inspection of the shutdown systems of the Darlington Nuclear Power Generating Station in Ontario, Canada [24]. Since conventional approach of software inspection is not good enough for safety-critical software, function documentation is applied where long programs are decomposed into sequences of state changes. Each part in the decomposition implements its assigned function. The entire program can be precisely specified and documented with tabular representations of the program function. *Requirement tables* are also constructed based on formal mathematical notation by nuclear safety experts. The correctness of the software is assessed by program experts who are in charge of showing program function tables and requirement tables expressing the same information through a step-by-step transformation.

2.1.3 Documentation of Non-deterministic Programs

Non-deterministic programs cannot be fully described by program functions since a program started in a safe state may terminate in one of several distinct final states. A *relation*, meaning here a *binary relation* has following definitions [27]:

- A *binary relation* R on a given set U is a set of ordered pairs with both elements from U , i.e. $R \subseteq U \times U$. The set U is called the Universe.
- The set of pairs R could also be defined by its *characteristic predicate*, $R(p, q)$, i.e. $R = \{(p, q) : U \times U \mid R(p, q)\}$.
- The *domain* and the range of R can be expressed as follows:

$$Dom(R) = \{p \mid \exists q[R(p, q)]\}, Range(R) = \{q \mid \exists p[R(p, q)]\}.$$

A *limited domain relation* (LD-relation) on U is an ordered pair $L = (R_L, C_L)$, where:

- R_L , the *relational component* of L , is a relation on U , i.e. $R_L \subseteq U \times U$,

- C_L , the *competence set* of L , is a subset of the domain of R_L , i.e. $C_L \subseteq \text{Dom}(R_L)$.

In order to explain how to use LD-relations especially in tabular form to specify a program in [27], Parnas analyzes the problem of writing an operation which finds the maximum of two integer values stored in programming variables:

- Rewrite a characteristic predicate of a relation directly as a table.
- Check the overlap of a table header.
- Express some rows of a table by standard notation and combine both notations together.
- Use abbreviation to represent predicates.
- Narrow long conditions.
- Replace relational operator.

Two examples are taken to illustrate Display Method—*Binary search* and *Dutch national flag*. In these two examples, the specification of the procedure invocation is written in terms of the combination of simple predicates and tabular predicates. Both specification of subproblems in the declaration and statements in the declaration body are written in terms of formal parameters.

2.1.4 An Example of the Use of Tables in System Documentation

Tables can also be used for semi-formal specifications of computing systems [36]. The general method applies to deterministic systems and is based on describing the system by a function

$$n : S \times C \rightarrow S$$

where S and C are appropriately chosen sets of states and commands, respectively. A classification of the inter-connection between states and commands, and of the algebraic structure of the state-transition function n , determine the way tables can be generated.

The interface of the dialogue box shown in Figure 2.1 provides the features for entering and deleting numeric values in X fields, toggling both of the check boxes B and I, and selecting the focus of the dialogue box. The behavior of the dialogue box is modeled by the system model $(S, C; n)$. The carrier and operators of this algebra are defined by :

1. The set of states is the Cartesian product.

$$S = Focus \times N \times B \times B$$

where set $Focus = \{x, b, i\}$ holds the possible locations in the dialogue box, N is the set of natural numbers, and B is the set of Boolean values.

2. The set of commands is defined to hold all possible events the system can receive.

$$C = Digits \cup Del \cup Space \cup FocusC, \text{ where}$$

- (a) $Digits = \{0, \dots, 9\}$ of digit events.
- (b) $Del = \{del\}$ of delete events.
- (c) $Space = \{space\}$ of toggle events.
- (d) $FocusC = \{\bar{x}, \bar{b}, \bar{i}\}$ of focus selection events.

3. System specification is the function $diag : S \times C \rightarrow S$.

It defines the behavior of the dialogue box is documented by the table in Figure 2.2. This decomposition fits the command-state decomposition. Here the semi-formal definition of $diag(s, c)$ starts by first classifying the command c and

then secondly the state s . The main header enumerates the control events and then each sub-header enumerates the local state conditions employed for each command event.

Alternatively, a state-command decomposition of the *diag* function can be used. This generates the table shown in Figure 2.3. Here the semi-formal definition of $diag(s, c)$ starts by first classifying the state s and then secondly the control event c .

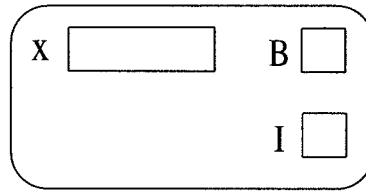


Figure 2.1: The Dialogue Box

2.2 Syntax, Semantics, and Transformations of Ten Kinds of Tables

Parnas summarizes his findings on tabular specifications and describes ten kinds of tables [23], giving their syntax and semantics based on the fact that tabular notation is useful for improving the readability of long mathematical definition, and is particularly well-suited to software documentation. The ten ways in which tables may be interpreted as predicates and functions are:

1. Normal Function Tables

A *normal function table*, T , is a table in which the elements of the main grid, G , are terms and the elements of the headers are predicate expressions.

| | | |
|-----------|--------------|--------------|
| Digits(c) | Focus[x](s) | Focus[bi](s) |
| | ins(s,c) | id(s,c) |
| Del(c) | Focus[x](s) | Focus[bi](s) |
| | del(s) | id(s,c) |
| Space(c) | Focus[bi](s) | Focus[x](s) |
| | toggle(s,c) | id(s,c) |
| FocusC(c) | All(s) | |
| | cycle(s,c) | |

Figure 2.2: Command Mode Table for Dialogue Box

| | | | | |
|--------------|-----------|----------|-------------|------------|
| Focus[x](s) | Digits(c) | Del(c) | Space(c) | FocusC(c) |
| | ins(s,c) | del(s,c) | id(s,c) | cycle(s,c) |
| Focus[bi](s) | Digits(c) | Del(c) | Space(c) | FocusC(c) |
| | id(s,c) | id(s,c) | toggle(s,c) | cycle(s,c) |

Figure 2.3: State Mode Table for Dialogue Box

2. Inverted Function Tables

An *inverted function table*, T , is a table in which the elements of the main grid, G , are predicate expressions, the elements of $H_2, \dots, H_{dim(T)}$ are predicate expressions, and the elements of H_1 are terms.

3. Vector Function Tables

| | | | | |
|---------|----------------------|---------------------------|-----------------|-------|
| | $y = 27$ | $y > 27$ | $y < 27$ | |
| | | | | H_1 |
| $x = 3$ | $27 + \sqrt{27}$ | $54 + \sqrt{27}$ | $y^2 + 3$ | G |
| $x < 3$ | $27 + \sqrt{-(x-3)}$ | $y + \sqrt{-(x-3)}$ | $y^2 + (x-3)^2$ | |
| $x > 3$ | $27 + \sqrt{x-3}$ | $2 \times y + \sqrt{x-3}$ | $y^2 + (3-x)^2$ | |
| | | | | H_2 |

Figure 2.4: A Normal Function Table

| | | | | |
|---------|----------|----------|--------------|-------|
| | $x + y$ | $x - y$ | $x \times y$ | |
| | | | | H_1 |
| $x = 3$ | $y < 3$ | $y = 3$ | $y > 3$ | G |
| $x < 3$ | $y < x$ | $y > x$ | $y = x$ | |
| $x > 3$ | $y < -x$ | $y > -x$ | $y = -x$ | |
| | | | | H_2 |

Figure 2.5: An Inverted Function Table

A *vector function table*, T , is a table in which the elements of the main grid, G , are terms, elements of $H_1, H_3, \dots, H_{\dim(T)}$ are predicate expressions, and the elements of H_2 are single variables.

| | | | | |
|-----|-------------|-------------|-------------|-------|
| | $w < 0$ | $w = 0$ | $w > 0$ | |
| | | | | H_1 |
| x | $x + w + q$ | $x + 2 - q$ | $x - w$ | G |
| y | $y + 2$ | $x + y$ | $x + y + 2$ | |
| z | $z - w$ | z | $z + w$ | |
| | | | | H_2 |

Figure 2.6: A Vector Function Table

4. Normal Relation Tables

A *normal relation table*, T , is a table in which the elements of the main grid and headers are predicate expressions. The expressions in the main grid and headers are constructed from the usual variables except that one variable, which will be written, " \textcircled{R} ", may not appear in the headers.

5. Inverted Relation Tables

| | | | | |
|---------|----------------------------|----------------------|----------------------------|-------|
| | $\sqrt{y} < 27$ | $\sqrt{y} = 0$ | $y > 0$ | H_1 |
| $x = 3$ | $x^2 + y^2 = \mathbb{R}^2$ | $x^2 = y^2$ | $true$ | G |
| $x < 3$ | $y^2 = \mathbb{R}^2$ | $x^2 = \mathbb{R}^2$ | $false$ | |
| $x > 3$ | $x^2 = \mathbb{R}^2$ | $x - \mathbb{R} > 3$ | $x^2 + y^2 = \mathbb{R}^2$ | |
| H_2 | | | | |

Figure 2.7: A Normal Relation Table

An *inverted relation table*, T , is a table in which the elements of the main grid and headers are predicate expressions. The expressions in $H_2, \dots, H_{dim(T)}$ and G are constructed using the usual variables except that one variable, which will be written " \mathbb{R} ", may not appear. The expressions in H_1 may include " \mathbb{R} ".

| | | | | |
|---------|----------------------------|------------------|----------------------------|-------|
| | $x^2 + y^2 = \mathbb{R}^2$ | $\mathbb{R} = 3$ | $x^2 + \mathbb{R}^2 = y^2$ | H_1 |
| $x = 3$ | $y > 3$ | $y = 3$ | $y < 3$ | G |
| $x < 3$ | $y < 0$ | $y \geq 0$ | $false$ | |
| $x > 3$ | $y > 100$ | $y = 100$ | $y < 100$ | |
| H_2 | | | | |

Figure 2.8: An Inverted Relation Table

6. Vector Relation Tables

A *vector relation table*, T , is a table in which the elements of the main grid, G , are predicate expressions, the elements of $H_1, H_3, \dots, H_{dim(T)}$ are predicate expressions, and the elements of H_2 are single variables.

| | | | | |
|---------|-------------------------|-----------------|---------------|-------|
| | $w < 0$ | $w = 0$ | $w > 0$ | H_1 |
| $x = 3$ | $y > 3$ | $x^2 = 4$ | $x^2 = w$ | G |
| $x < 3$ | $y < 0$ | $y = x + 2$ | $y = x + 2$ | |
| $x > 3$ | $z^2 = x^2 + y^2 + w^2$ | $z^2 = x^2 + y$ | $z = 5$ | |
| H_2 | | | | |

Figure 2.9: A Vector Relation Table

7. Mixed Vector Tables

A *mixed vector table*, T , is a table in which the elements of the main grid, G , are either predicate expressions or terms, the elements of $H_1, H_3, \dots, H_{\dim(T)}$ are predicate expressions, and the elements of H_2 are single variables.

| | | | | |
|-------|-------------------------|-------------------|---------------|-------|
| | $w < 0$ | $w = 0$ | $w > 0$ | |
| | | | | H_1 |
| $x =$ | $x + w + q$ | $x + 2 - q$ | $x - w$ | |
| $y $ | $y^2 = x + 2$ | $y = x + 2$ | $y = x + 2$ | |
| $z $ | $z^2 = x^2 + y^2 + w^2$ | $z^2 = x^2 + w^2$ | $z = 5$ | |
| | | | | G |
| H_2 | | | | |

Figure 2.10: A Mixed Vector Table

8. Predicate Expression Tables

A *predicate expression table*, T , is a table in which the elements of the main grid, G , and all headers are predicate expressions.

| | | | | |
|---------|-----------|-------------|-------------|-------|
| | $w < 0$ | $w = 0$ | $w > 0$ | |
| | | | | H_1 |
| $x = 3$ | $y = 5$ | $y + x = w$ | $x + y = z$ | |
| $x < 3$ | $y > 7$ | $y - x = 6$ | $x - y = z$ | |
| $x > 3$ | $y^2 = 4$ | $y^2 = 4$ | $z = y$ | |
| | | | | G |
| H_2 | | | | |

Figure 2.11: A Predicate Expression Table

9. Characteristic Predicate Tables

A *characteristic predicate table*, T , is a *predicate expression table* where the *decorations* (the symbols “ $'$ ” and “ $''$ ”) are considered part of the variable name. It can be viewed as the set of ordered pairs that constitutes the relation. Characteristic predicate tables are especially useful in tabular verification and refinement [32] to derive weakest precondition and data refinement theorems.

| | | | | |
|-------|-----------------------------------|---|----------------|-------|
| | $'w < 0$ | $'w = 0$ | $'w > 0$ | H_1 |
| H_2 | $(x' = w') \wedge$ $(w' = 'x)$ | $(x' = w') \wedge$ $(w' = 'y) \wedge$ $(w' = 'y)$ | $w' = x' = 'y$ | |
| | $y' = 'x$ | $y' = 'y$ | $'w = w'$ | |
| | $y'^2 = 4$ | $x' + w' = 'y$ | $y' = 'x$ | G |

Figure 2.12: A Characteristic Predicate Table

10. Generalized Decision Tables

A *generalized decision tables*, T , is a table in which the elements of the main grid, G , are predicate expressions that may contain a distinguished variable, which we shall denoted by " $\#$ ", and the elements of H_1 and H_2 are terms that do not include " $\#$ ". $H_3, \dots, H_{\dim(T)}$ are not used in this interpretation of tables.

| | | | |
|-------|--------------|--------------|-----------|
| | $x + y$ | $x - y$ | H_1 |
| H_2 | $x \times y$ | $\# < 20$ | $\# < 20$ |
| | $x \div y$ | $\# \geq 20$ | $\# = 20$ |
| | x^2 | $true$ | $\# > 20$ |
| | | | G |

Figure 2.13: A Generalized Decision Table

Problems arise when considering the best kind of tables to represent a function. In order to transform a table to a simpler form of the same kind, Zucker describes the change of a table's dimension and the transformation between a normal and an inverted function table in [37]. The concept of proper table is clarified in an algebraic way. Terms, conditions of a table and the table itself are defined over many-sorted signatures.

2.3 Automatic Tabular Documentation Tools

2.3.1 RSML for Process-Control Systems

The article [16] defines general analysis criteria that can be applied to black-box requirements specified for process-control systems. Particular attention is focused on the properties of robustness and lack of ambiguity. Semantic analysis techniques associated with a specific model, *Requirements State Machine* (RSM), and a specification language, *Requirements State Machine Language* (RSML), are developed to ensure that these criteria are satisfied for a given specification.

RSML's application on an industrial aircraft collision avoidance system, Traffic Alert and Collision Avoidance System II (TCAS II), are first introduced in [19]. RSML has the features in common with Statecharts: *superstates*, *AND decomposition*, *arrays*, and *connectives*. The syntactic and semantic additions to Statecharts are: *directed communications*, *external events*, *interface definitions*, *component state machines*, *transition definitions*, *macros and functions*, *transition buses*, *cross referencing and identifier types*, *identity transitions*, *timing*, and *step semantics*.

The guarding conditions on the transitions are described by a tabular representation of *disjunctive normal form* (DNF) called AND/OR tables instead of predicate calculus which makes it easier to parse an expression. The far-left column of the AND/OR table contains the logical phrases, each other column is a conjunction of the logical phrases, all columns together are disjunction of each conjunct term which make it much easier to parse the expression. A dot denotes "don't care" for omissions. The table in Figure 2.14 is equivalent to predicate:

$$((\text{Expression-1} \wedge \neg \text{Expression-3}) \vee (\neg \text{Expression-2} \wedge \text{Expression-3}))$$

AND/OR tables are provided in analysis procedures to find completeness, consistency and safety errors in specifications. Although in the TCAS II requirements

| | | | |
|---|--------------|----|---|
| | | OR | |
| A | Expression-1 | T | • |
| N | Expression-2 | • | F |
| D | Expression-3 | F | T |

Figure 2.14: An AND/OR Table

specification, automated analysis tools use *Binary Decision Diagrams* (BDDs) for the manipulation of the guarding conditions [11], the readability and reviewability of the AND/OR tabular representation make the error be discovered quickly by the application experts.

2.3.2 Model Checking

The *Software Cost Reduction* (SCR) requirements method comes from A-7 aircraft operational flight documentation. Faulk [8] provides formal definitions for the A-7 model, van Schouwen [35] extends the original SCR method for the safety-critical components of the Darlington nuclear power plant. The characteristic of this model are compositional, event-driven, mode-machines.

Atlee demonstrate how model checking can be used to verify safety properties for event-driven systems in [3]. A model checking system, MCB, is used for formalizing the semiformal event-driven SCR requirements. MCB accepts a system's behavior requirements as a finite state machine and the safety assertions as temporal logic formulae. First, the transformation algorithm formalizes the software requirements. Then, it can be verified or disproved that the tabular specifications and the relationships between conditions were entered correctly. The shortcoming is that the model checker only analyzes properties of *mode transition tables*. The mode transition tables

| Old Mode | Conditions | | | | New Mode |
|----------|---------------|---------------|---------|---------------|-----------|
| m_1 | $EOC_{1,1,1}$ | $EOC_{1,1,2}$ | \dots | $EOC_{1,1,p}$ | $m_{1,1}$ |
| | $EOC_{1,2,1}$ | $EOC_{1,2,2}$ | \dots | $EOC_{1,2,p}$ | $m_{1,2}$ |
| | \dots | \dots | \dots | \dots | \dots |
| | $EOC_{1,k,1}$ | $EOC_{1,k,2}$ | \dots | $EOC_{1,k,p}$ | $m_{1,k}$ |
| \dots | \dots | \dots | \dots | \dots | \dots |
| m_n | $EOC_{n,1,1}$ | $EOC_{n,1,2}$ | \dots | $EOC_{n,1,p}$ | $m_{n,1}$ |
| | $EOC_{n,2,1}$ | $EOC_{n,2,2}$ | \dots | $EOC_{n,2,p}$ | $m_{n,2}$ |
| | \dots | \dots | \dots | \dots | \dots |
| | $EOC_{n,k,1}$ | $EOC_{n,k,2}$ | \dots | $EOC_{n,k,p}$ | $m_{n,k}$ |

Table 2.3: A Mode Transition Table

are of the form in Table 2.3.

Each row in the table specifies the event causing the transition from the mode on the left to the mode on the right. A table entry in the middle of the table may be an event (@T,@F) triggered by a change of the condition, or a condition value (t,f) which are depended by the event. The global tabular specification is then converted into a CTL machine. The model checker accepts a CTL machine and a CTL formula³, and determines whether the formula holds in the machine or not, which can perform state-based model checking using CTL model checker.

2.3.3 SCR*

Parnas and Medey [26] introduced the *four-variable model* to provide a formal framework for semantic decomposition. The four-variable model describes the system behavior (functional or non-functional requirements) as a set of mathematical relations on four sets of variables — monitored and controlled variables and input and output data items. Four-Variable Model together with four other constructs — modes, terms, conditions, and events are SCR constructs [13]. The tabular notation

³Computational tree logic (CTL) is a temporal logic. It uses atomic propositions as its building blocks to make statements about the states of a system. CTL then combines theses propositions into formulae using logical operators and temporal operators.

in SCR specifications facilitates industrial application of the method. There are three kinds of tables — *event tables*, *condition tables* and *mode transition tables*. Event and condition tables are the same as those in the A-7 documentation. Mode transition tables in [13] are similar to Table 2.3 except that several depended conditions and one input triggered event are combined together to build a new event by a logical expression of form:

$@T(Cond1) \text{ WHEN } [Cond2]$

Below is an example of a mode transition table which describes a mode (monitored variable) as a function of an old mode and an event.

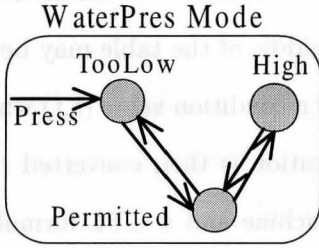


Figure 2.15: WaterPres Mode Transition Table for Press

| Old Mode | Event | New Mode |
|-----------|--|-----------|
| TooLow | $@T(\text{WaterPres} \geq \text{Low})$ | Permitted |
| Permitted | $@T(\text{WaterPres} \geq \text{Low})$ | High |
| Permitted | $@T(\text{WaterPres} < \text{Low})$ | TooLow |
| High | $@T(\text{WaterPres} < \text{Permit})$ | Permitted |

We consider WaterPres mode class are divided into three segment mode — TooLow, Permitted, and High by two constants Low and Permit. The first row illustrates that if the old mode is TooLow, then after the event which makes water pressure below Low exceed Low mark, the new mode of WaterPres is Permitted.

Illuminated by the Four-Variable Model, a formal requirement model, *finite-state automaton system model*, was invented to provide a precise and detailed semantics for the SCR method [13]. It describes the input and output variables, conditions, events, and other constructs that make up an SCR specification in terms of that automaton. The model also describes how a set of table functions, called table functions, can be derived from the SCR tables. These table functions define the transform which maps the current state and an input event to a new state.

A powerful and robust CASE tool, SCR*, was developed on this model to support automatic detection of errors expressed in the SCR tabular notation [12]. The toolset which is written in C++ and runs on SPARC workstation includes a specification editor for creating and modifying the specification, a simulator for symbolically executing the system, and two formal analysis tools. One analysis tool is a consistency checker for testing the specifications for consistency with a formal requirement model; the other is a verifier for checking that the specifications satisfy selected application properties.

2.3.4 Tablewise

Decision tables are widely used for specifying finite functions, such as finite state transitions [15]. Table 2.4 is a simple decision table that specifies an input-output relation. The first two columns list the input variables and the possible values, called *states*, that each may take. The top row lists the possible output values (operational procedures, or modes to be entered). The rest of the table is called its *body*. Each row of the body belongs to the variable listed in the first column in that row. Each column, or *scenario*, represents the conjunction of its cells.

| Operational Procedure | | Op Proc 1 | | Op Proc 2 |
|-----------------------|-------------------|-----------|--------|-----------|
| Senarios | | Scen 1 | Scen 2 | Scen 3 |
| Inputs | Values | | | |
| SI_1 | s_1, s_2 | s_1 | s_1 | s_2 |
| SI_2 | s_1, \dots, s_n | s_1 | s_2 | s_n |

Table 2.4: A Simple Decision Table

Hoover and Chen have demonstrated the utility of Tablewise in [15]. Tablewise performs three kinds of logical analysis of decision tables: detecting overlap between engagement criteria of different operational procedures (failure of consistency), detect-

ing scenarios not covered by any engagement criterion (failure of completeness), and detecting structural defects in a decision table that prevent it from being functional. Besides logical analysis, Tablewise can generate code implementing a decision table and English-language text describing it.

Completeness is more difficult to detect than overlap since no one column in the table is responsible for the absence of others. In order to localize flaws causing incompleteness, a form of structural analysis is developed. It finds minimum sets of variables that the table correlates in a way that precludes functionality. Therefore, overlap and structural analysis together is a method for analyzing functionality of decision tables.

2.3.5 TABLE Construct in PVS

In addition to these independent table tools, a simple construct for tabular specification is added to PVS [22]. This construct is useful for many purposes since it cooperates with other utilities of PVS. The side condition of the construct is that the rows and columns are disjoint and exclusive. This utility is used in requirements analysis for Space Shuttle flight software by colleagues at NASA and Lockheed-Martin [6] [29]. Lawford et al. [18] use the capability to verify decomposition of proof obligations, on second thoughts, extending the 4-variable model to an 8-variable model by adding tolerance relations. TABLE constructs of PVS has an input syntax for one-dimensional, two-dimensional and enumeration tables and allowing blank entries when a specific condition can no longer arise. These tables are translated to internal IF-THEN-ELSE constructs of PVS theorem prover and printed as true tables by \LaTeX typesetting.

One-dimensional tables have vertical and horizontal formats shown in Figure 2.6. Each row is included by `|` and `||` except that the upper header of a horizontal table

use `[[...]]` to alert parser that it present the information in a different order.

| | |
|---|---|
| <pre> sign_vtable(x): signs = TABLE %-----% x < 0 -1 %-----% x = 0 0 %-----% x > 0 1 %-----% ENDTABLE </pre> | <pre> sign_htable(x): signs = TABLE %-----% [x < 0 x = 0 x > 0] %-----% -1 0 1 %-----% ENDTABLE </pre> |
|---|---|

Figure 2.16: One-Dimensional Vertical and Horizontal Tables

Enumeration Tables are a syntactic variation of one-dimensional or two-dimensional tables where the conditions to a table are of the form $x = \text{expression}$ for some single identifier x . Figure 2.7 gives a two-dimensional table and its corresponding enumeration tables.

| | |
|--|---|
| <pre> two-dimensional(state, input): some_type = TABLE %-----% [state = a] [state = b] %-----% input = x p q %-----% input = y s t %-----% ENDTABLE </pre> | <pre> enumeration(state, input): some_type = TABLE state , input %-----% [x] [y] %-----% a p s %-----% b q t %-----% ENDTABLE </pre> |
|--|---|

Figure 2.17: A Two-Dimensional Table and Its Corresponding Enumeration Table

The PVS TABLE construct can not represent decision tables supported by Tablewise. The TABLE construct interprets a scenario of a decision table as the argument list to a function X to be applied to each cell of that scenario. Thus, function X and an operational procedure is represented as a vertical one-dimensional PVS table. We can also use tabular specifications in a theorem proving context to identify the anomaly in either our expectations or our formalization of the specification.

Transition relations provide a way to pose and examine invariant or reachable property by simulating the tabular manner of the SCR method. To model the mode transition table of Figure 2.5 in PVS, a condition is specified as a predicate on inputs to the system, then atT (@T), atF (@F), T , F , and dc ("don't care") are higher order functions. The corresponding PVS representation is shown in Figure 2.8.

```

event_constructor: TYPE = [condition -> event]
EC: TYPE = event_constructor
PC(A,B)(a,b)(p,q):bool = A(a)(p,q) & B(b)(p,q)
% Note: PC stands for "pairwise conjunction"

original(s: modes, (p, q: monitored_vars)): modes =
LET
  x = (WaterPres >= Low, WaterPres < Permit),
  X = (LAMBDA (a,b: EC) : PC (a,b) (x) (p, q))
IN TABLE s
| TooLow | TABLE
  %-----|-----|-----|
  | X (      atT,      dc ) | Permitted ||
  %-----|-----|-----|
  |      ELSE              | TooLow    ||
  %-----|-----|-----|
ENDTABLE ||

| Permitted | TABLE
  %-----|-----|-----|
  | X (      atT,      dc ) | High    ||
  %-----|-----|-----|
  | X (      atF,      dc ) | TooLow  ||
  %-----|-----|-----|
  |      ELSE              | Permitted ||
  %-----|-----|-----|
ENDTABLE ||

| High | TABLE
  %-----|-----|-----|
  | X (      dc,      atT ) | Permitted ||
  %-----|-----|-----|
  |      ELSE              | High    ||
  %-----|-----|-----|
ENDTABLE ||
ENDTABLE

```

Figure 2.18: The SRC Table Represented in PVS

A state transition relation can be derived from the corresponding mode transition table. The branching time temporal logic CTL provides a method to specify some properties of the computations induced by a transition relation. The PVS model-check command can verify formulae specified by CTL. Overall, the PVS treatment requires no customized development: it builds on capacities such as tables, typechecker-generated proof obligations, dependent typing, higher-order functions, model-checking and theorem proving.

2.4 Table Tool System

Abraham outlines the documentation methods in [2] for software products developed by *Software Engineering Research Group* (SERG) at McMaster University, and describes how a generalized model of tabular expressions has been applied to build a tool that evaluates a broad class of software documentation. An application of the *Table Tool System* (TTS) that interprets a group of tabular expressions is also provided. C code generated by the tool will evaluate the logical expressions contained in an input specification. Expression evaluation is useful when checking a specification or for testing an implementation against its specification.

The model presented in [17] covers most of the known table types for documenting Software Engineering projects, and admits precise classification and definition of new types of tables. The central concept in the approach is *cell information flow graph* (CCG) which characterizes the information flow among table cells. A raw table skeleton is defined by a header and a grid. It is extended to a medium table skeleton by adding a CCG. A well done table skeleton consists of a table predicate rule, a table relation rule, a table composition rule and a medium table skeleton. A tabular expression is a tuple of a well done table skeleton, a mapping which assigns a predicate

expression to each guard cell and a relation expression to each value cell, together with a set of inputs and outputs. Using tabular expressions in [17] enables us to use mathematical precision in the documentation of software requirements, eases the methods of extending and/or modifying tables, and most importantly, helps in building automated tools that are able to interpret tabular expressions.

Chapter 3

Tabular Verification and Refinement Overview

The utility of tabular specification in Chapter 2 are proposed for structuring complex mathematical expressions or informal operations. In their formal manipulation, tabular predicates and tabular relations are explored to be used in program verification and refinement based on pre- and post-conditions. This chapter is literally summarized from [32]. We present it here to make the thesis self-contained.

3.1 Terminology and Notation

3.1.1 Properties of Vectors

A vector pv is *disjoint* if all its elements are mutually exclusive, $\neg(pv_i \wedge pv_k)$ for all i and k with $i \neq k$. Two vectors pv and qv are *jointly disjoint* if $\neg(pv_i \wedge qv_j \wedge pv_k \wedge qv_l)$ for all i, j, k, l with either $i \neq k$ or $j \neq l$. If pv and qv are jointly disjoint, then the conjunction of any two elements of $pv(qv)$ does not need to be false in isolation but only if conjoined with an element of $qv(pv)$. Vector pv *covers (at least) c* if one of its

elements is true if c is true, $c \implies \forall i \cdot pv_i$ and *covers exactly* c if $c \equiv \forall i \cdot pv_i$. Vector pv is *total* if it covers *true*. Vector pv *partitions* c if it is disjoint and covers exactly c .

3.1.2 Relations

It is usual to describe relations using boolean expressions. A boolean expression can be presented in a tabular form [23]. Notational conventions in [27] are used to increase the readability of tables. A non-deterministic program is modeled by a relation of type $XV \rightarrow XV' \rightarrow Bool$ with initial state space XV and final state space XV' . Let P be a program specified by the characteristic predicate of a relation, and let xv_1, \dots, xv_k be variables in P which form its state space, $XV = (xv_1, \dots, xv_k)$. Then:

- " xv_i " (to be read " xv_i before") denotes the value of the program variable v_i before an execution of P ,
- " xv'_i " (to be read " xv_i after") denotes the value of the variable v_i after a terminating execution of P ,
- Since Ocaml does not support prime as its' composition of variable name, we use v_i1 instead of v'_i in the implementation of tabular specification and refinement using this programming language.

3.1.3 Relational Operation

We define the constant relations \perp (empty relation), \top (universal relation), Id (identity relation), and for relations P and Q the operations \overline{P} (complement), P^{-1} (inverse), $P \cap Q$ (intersection), $P \cup Q$ (union), $P \circ Q$ (relational composition) as well as the predicates $P \subseteq Q$ and $P \supseteq Q$ (inclusion):

$$\begin{array}{ll}
 \perp \text{ } xv \text{ } xv' \equiv \text{false} & (P \cup Q) \text{ } xv \text{ } xv' \equiv P \text{ } xv \text{ } xv' \vee Q \text{ } xv \text{ } xv' \\
 \top \text{ } xv \text{ } xv' \equiv \text{true} & (P \cap Q) \text{ } xv \text{ } xv' \equiv P \text{ } xv \text{ } xv' \wedge Q \text{ } xv \text{ } xv' \\
 Id \text{ } xv \text{ } xv' \equiv xv = xv' & (P \circ Q) \text{ } xv \text{ } xv' \equiv (\exists yv \cdot P \text{ } xv \text{ } yv \wedge Q \text{ } yv \text{ } xv') \\
 \overline{P} \text{ } xv \text{ } xv' \equiv \neg P \text{ } xv \text{ } xv' & (P \subseteq Q) \equiv (\forall xv, xv' \cdot P \text{ } xv \text{ } xv' \Rightarrow Q \text{ } xv \text{ } xv') \\
 P^{-1} \text{ } xv \text{ } xv' \equiv P \text{ } xv' \text{ } xv & (P \supseteq Q) \equiv (\forall xv, xv' \cdot P \text{ } xv \text{ } xv' \Leftarrow Q \text{ } xv \text{ } xv')
 \end{array}$$

A relation P is *functional* if $P^{-1} \circ P \subseteq Id$ and *injective* if $P \circ P^{-1} \subseteq Id$. Relation P is called a *condition* if $P \circ \top = P$. The *domain* ΔP of a relation P is defined by $\Delta P = P \circ \top$. A relation P is *total* if $\Delta P = \top$, or equivalently $Id \subseteq P \circ P^{-1}$. Relation P is *surjective* if $\Delta P^{-1} = \top$, or equivalently $Id \subseteq P^{-1} \circ P$. We make use of generalized union $\cup i \in I \cdot P_i$ and generalized intersection $\cap i \in I \cdot P_i$, for arbitrary index set I . Relations have the following facts.

Let P, Q, P_i, Q_i be relations and C a condition:

- (a) $P \circ (\cup i \in I \cdot Q_i) = \cup i \in I \cdot P \circ Q_i$
- (b) $(\cup i \in I \cdot P_i) \circ Q = \cup i \in I \cdot P_i \circ Q$
- (c) $P \circ (\cap i \in I \cdot Q_i) \subseteq \cap i \in I \cdot P \circ Q_i$
- (d) $(\cap i \in I \cdot P_i) \circ Q \subseteq \cap i \in I \cdot P_i \circ Q$
- (e) $P \circ (\cap i \in I \cdot Q_i) = \cap i \in I \cdot P \circ Q_i$ if P is functional.
- (f) $(\cap i \in I \cdot P_i) \circ Q = \cap i \in I \cdot P_i \circ Q$ if Q is injective.
- (g) $(C \cap P) \circ Q = C \cap (P \circ Q)$

3.1.4 Precondition and Weakest Precondition

Assuming that the set of program variables is fixed, we can determine for any statement a characterizing predicate over unprimed and primed variables.

The theory about deriving precondition from program statements is shown as below [34]:

If an operation S over initial state xv_1, \dots, xv_m and final state xv'_1, \dots, xv'_n is given by a predicate P ,

$$S(xv_1, \dots, xv_m)(xv'_1, \dots, xv'_n) = P$$

then its precondition is:

$$pre\ S = (\exists xv'_1, \dots, xv'_n \cdot P)$$

The domain ΔP of a program P is interpreted either as the *enabledness domain* (or guard) of P or as the *termination domain* (or *precondition*) of P . The weakest precondition $[P]C$ of program P to establish post condition C characterizes those initial states in which P is never going to lead to a state outside C :

$$[P]C = \overline{P \circ \overline{C}}$$

If ΔP is interpreted as the enabledness domain of program P , then $[P]C$ characterizes those initial states in which either P is not enabled or P is enabled and leads to a state in C . If ΔP is interpreted as the termination domain of program P , then $[P]C$ characterizes those initial states in which either P does not terminate or P terminates and leads to a state in C . In this case we would refer to $[P]C$ as the *weakest liberal precondition*. Leaving both interpretations open, we uniformly refer to $[P]C$ as the weakest precondition for P to establish C .

The weakest precondition can equivalently be defined in terms of predicates. We assume that the state consists of a vector xv of variables and that the initial and final state space are products of the same type:

Theorem 5.1 (Weakest Precondition).

$$[P]C\ xv\ xv' \equiv \forall xv' \cdot P\ xv\ xv' \Rightarrow C\ xv'\ xv'$$

3.2 Tabular Predicates

Tabular predicates are predicates written as a disjunction of conjunctions. A tabular predicate with one *header* consisting of predicates p, q, r and a *body* consisting of predicates s, t, u is defined by

$$\frac{p \mid q \mid r}{s \mid t \mid r} \equiv (p \wedge s) \vee (q \wedge t) \vee (r \wedge u)$$

In general, let I be a finite and non-empty set of indices and let pv be an indexed collection of predicates that we call a *vector*, with elements pv_i for $i \in I$. Tables with a single header are *one-dimensional*. With pv and qv vectors over the same index set, we introduce a shorthand for a table with header pv , and body qv , defined by generalizing the above example:

$$\frac{pv}{qv} \equiv \vee i \cdot pv_i \wedge qv_i$$

On vectors pv and qv over the same index set $\neg pv, pv \wedge qv, pv \vee qv, pv \Rightarrow qv, pv \Leftarrow qv$, and $pv \equiv qv$ are all defined by the pointwise extension of the corresponding operators on *Bool*, e.g. $(pv \wedge qv)_i \equiv pv_i \wedge qv_i$. On occasion we identify a predicate p with a vector with all elements being p . This also allows us to write expressions like $p \wedge pv$, with the meaning of $(p \wedge pv)_i \equiv p \wedge pv_i$, and similarly for other Boolean operators.

In general, an n -dimensional table has n headers; here we restrict ourselves to one- and two-dimensional tables. Let I and J be index sets, let pv be an I -indexed vector, let qv be a J -indexed vector, and let rm be a doubly indexed collection of predicates that we call a *matrix*, with elements $rm_{i,j}$ for $i \in I$ and $j \in J$. We introduce a shorthand for a two-dimensional tabular predicate with headers pv, qv and body rm :

$$\frac{\mid qv}{pv \mid rm} \equiv \vee i, j \cdot pv_i \wedge qv_j \wedge rm_{i,j}$$

We also use a shorthand with multiple vectors in one header, with the special case of one vector being a single predicate:

$$\frac{}{pv \mid \frac{qv \mid rv}{qm \mid rm}} \equiv \forall i \cdot (\forall j \cdot pv_i \wedge qv_j \wedge rm_{i,j}) \vee (\forall k \cdot pv_i \wedge rv_k \wedge rm_{i,k})$$

On matrices rm and sm over the same index sets $\neg rm, rm \wedge sm, rm \vee sm, rm \Rightarrow sm, rm \Leftarrow sm$, and $rm \equiv sm$ are all defined by the pointwise extension of the corresponding operators on *Bool*, e.g. $(\neg rm)_{i,j} \equiv \neg rm_{i,j}$. On occasion we identify a predicate p with a matrix with all elements being p . This also allows us to write expressions like $p \wedge pm$, with the meaning of $(p \wedge pm)_{i,j} \equiv p \wedge pm_{i,j}$, and similarly for other Boolean operators. We will also identify a vector pv with a matrix with all columns being pv . This allows us to write expressions like $pv \wedge pm$, with the meaning of $(pv \wedge pm)_{i,j} \equiv pv_i \wedge pm_{i,j}$, and similarly for other Boolean operators.

Some basic transformations of tabular predicates are:

Theorem 2.1 (Transposing).

$$\frac{}{pv \mid rm} \equiv \frac{}{qv \mid rm^T}$$

Theorem 2.2 (Swapping Rows and Columns).

$$\frac{}{pv \mid \frac{qv \mid rv}{qm \mid rm}} \equiv \frac{}{pv \mid \frac{rv \mid qv}{rm \mid qm}}$$

Theorem 2.3 (Splitting and Joining Tables).

$$\frac{}{pv \mid \frac{qv \mid rv}{qm \mid rm}} \equiv \frac{}{pv \mid \frac{qv}{qm}} \vee \frac{}{pv \mid \frac{rv}{rm}}$$

Theorem 2.4 (Extending and Contracting).

$$\frac{}{pv \mid \frac{qv \mid rv}{qm \mid rm}} \equiv \frac{}{pv \mid \frac{qv}{qm}} \equiv \frac{}{pv \mid \frac{rv}{qm}} \Rightarrow \frac{}{pv \mid \frac{qv}{qm}}$$

Theorem 2.5 (Lifting and Flattening).

$$\begin{aligned} \text{(a)} \quad & \frac{}{p \mid \frac{qv}{rv}} \equiv \frac{}{p \wedge rv} \\ \text{(b)} \quad & \frac{}{pv \mid \frac{qv}{rm}} \equiv \frac{}{\forall i \cdot pv_i \wedge rm_i} \end{aligned}$$

Theorem 2.6 (Removing Header Overlap).

$$\frac{}{pv \mid \frac{qv \mid s \mid t}{rm \mid sv \mid tv}} \equiv \frac{}{pv \mid \frac{qv \mid s \wedge \neg t \mid \neg s \wedge t \mid s \wedge t}{rm \mid sv \mid tv \mid sv \vee tv}}$$

Theorem 2.7 (Making Header Total).

$$\frac{}{pv \mid rm} \frac{qv}{\mid} \equiv \frac{}{pv \mid rm} \frac{qv \mid \neg \vee j \cdot qv_j}{\mid \mid false}$$

Theorem 2.8 (Replacing Table Elements).

$$\begin{aligned} \text{(a)} \quad & \left(\frac{}{p \mid r \mid \dots} \frac{q \mid qv}{\mid \mid} \equiv \frac{}{p \mid r' \mid \dots} \frac{q \mid qv}{\mid \mid} \right) \Leftarrow (p \wedge q \Rightarrow (r \equiv r')) \\ \text{(b)} \quad & \left(\frac{}{pv \mid rv \mid \dots} \frac{q \mid qv}{\mid \mid} \equiv \frac{}{pv \mid rv \mid \dots} \frac{q' \mid qv}{\mid \mid} \right) \Leftarrow (\wedge i \cdot pv_i \wedge rv_i \Rightarrow (q \equiv q')) \end{aligned}$$

Theorem 2.9 (Splitting and Joining Rows and Columns).

$$\frac{}{pv \mid sv \mid \dots} \frac{q \vee r \mid \dots}{\mid \mid} \equiv \frac{}{pv \mid sv \mid sv \mid \dots} \frac{q \mid r \mid \dots}{\mid \mid}$$

3.3 Operations on Tabular Predicates

We give some basic theorems about common Boolean operators applied to tables.

Theorem 3.1 (Table Negation)

$$\begin{aligned} \text{(a)} \quad & \neg \left(\frac{}{pv \mid rm} \frac{qv}{\mid} \right) \equiv \wedge i, j \cdot pv_i \wedge qv_j \Rightarrow \neg rm_{i,j} \\ \text{(b)} \quad & \neg \left(\frac{}{pv \mid rm} \frac{qv}{\mid} \right) \Rightarrow \frac{}{pv \mid \neg rm} \frac{qv}{\mid} \quad \text{if } pv, qv \text{ are total} \\ \text{(c)} \quad & \neg \left(\frac{}{pv \mid \neg rm} \frac{qv}{\mid} \right) \Leftarrow \frac{}{pv \mid \neg rm} \frac{qv}{\mid} \quad \text{if } pv, qv \text{ are jointly disjoint} \\ \text{(d)} \quad & \neg \left(\frac{}{pv \mid rm} \frac{qv}{\mid} \right) \equiv \frac{}{pv \mid \neg rm} \frac{qv}{\mid} \quad \text{if } pv, qv \text{ are total and jointly disjoint} \end{aligned}$$

Theorem 3.2 (Table Conjunction)

$$\begin{aligned} \text{(a)} \quad & \frac{}{pv \mid rm} \frac{qv}{\mid} \wedge \frac{}{pv \mid sm} \frac{qv}{\mid} \Leftarrow \frac{}{pv \mid rm \wedge sm} \frac{qv}{\mid} \\ \text{(b)} \quad & \frac{}{pv \mid rm} \frac{qv}{\mid} \wedge \frac{}{pv \mid sm} \frac{qv}{\mid} \equiv \frac{}{pv \mid rm \wedge sm} \frac{qv}{\mid} \quad \text{if } pv, qv \text{ are jointly disjoint} \end{aligned}$$

Theorem 3.3 (Table Disjunction).

$$\frac{}{pv \mid rm} \vee \frac{}{pv \mid sm} \equiv \frac{}{pv \mid rm \vee sm} \quad \frac{qv}{\mid}$$

Theorem 3.4 (Predicate-Table Implication).

- (a) $\left(\frac{}{pv \mid rm} \Rightarrow s \right) \equiv (\wedge i, j \cdot pv_i \wedge qv_j \wedge rm_{i,j} \Rightarrow s)$
- (b) $\left(s \Rightarrow \frac{}{pv \mid rm} \right) \Rightarrow (\wedge i, j \cdot s \wedge pv_i \wedge qv_j \Rightarrow rm_{i,j})$ if pv, qv are jointly disjoint.
- (c) $\left(s \Rightarrow \frac{}{pv \mid rm} \right) \Leftarrow (\wedge i, j \cdot s \wedge pv_i \wedge qv_j \Rightarrow rm_{i,j})$ if pv covers s , qv covers s .
- (d) $\left(s \Rightarrow \frac{}{pv \mid rm} \right) \equiv (\wedge i, j \cdot s \wedge pv_i \wedge qv_j \Rightarrow rm_{i,j})$
if pv covers s , qv covers s and pv, qv are jointly disjoint.
- (e) $\left(s \Rightarrow \frac{}{pv \mid rm} \right) \equiv \frac{}{s \Rightarrow pv \mid s \Rightarrow rm} \quad \frac{qv}{\mid}$
- (f) $\left(s \Rightarrow \frac{}{pv \mid rm} \right) \equiv \frac{}{pv \mid s \Rightarrow rm} \quad \frac{qv}{\mid}$ if pv, qv are total and jointly disjoint.

Theorem 3.5 (Table Implication).

- (a) $\left(\frac{}{pv \mid rm} \Rightarrow \frac{}{pv \mid sm} \right) \Leftarrow \wedge i, j \cdot pv_i \wedge qv_j \wedge rm_{i,j} \Rightarrow sm_{i,j}$
- (b) $\left(\frac{}{pv \mid rm} \Rightarrow \frac{}{pv \mid sm} \right) \Rightarrow \frac{}{pv \mid rm \Rightarrow sm} \quad \frac{qv}{\mid}$ if pv, qv are total.
- (c) $\left(\frac{}{pv \mid rm} \Rightarrow \frac{}{pv \mid sm} \right) \Leftarrow \frac{}{pv \mid rm \Rightarrow sm} \quad \frac{qv}{\mid}$ if pv, qv are jointly disjoint.
- (d) $\left(\frac{}{pv \mid rm} \Rightarrow \frac{}{pv \mid sm} \right) \equiv \frac{}{pv \mid rm \Rightarrow sm} \quad \frac{qv}{\mid}$
if pv, qv are total and jointly disjoint.

Theorem 3.6 (Table Equivalence).

- (a) $\left(\frac{}{pv \mid rm} \equiv \frac{}{pv \mid sm} \right) \Leftarrow \wedge i, j \cdot pv_i \wedge qv_j \Rightarrow (rm_{i,j} \equiv sm_{i,j})$

- (b) $\left(\frac{pq}{pv} \mid \frac{qv}{rm} \equiv \frac{qv}{pv} \mid \frac{qv}{sm} \right) \Leftarrow \frac{qv}{pv} \mid \frac{qv}{rm \equiv sm}$ if pv, qv are jointly disjoint.
- (c) $\left(\frac{pq}{pv} \mid \frac{qv}{rm} \equiv \frac{qv}{pv} \mid \frac{qv}{sm} \right) \equiv \frac{qv}{pv} \mid \frac{qv}{rm \equiv sm}$
if pv, qv are total and jointly disjoint.

3.4 Tabular Relations

Tabular relations are defined in analogy to tabular predicates using generalized intersection and union. Let PV and QV be vectors of relations and let RM be a matrix of relations:

$$\frac{PV}{PV} \mid \frac{QV}{RM} \equiv \cup_{i,j} PV_i \cap QV_i \cap RM_{i,j}$$

All operations on relations are pointwise extended to operations on vectors and matrices. On occasion we identify a relation P with a vector or a matrix with all elements being P . For example, this allows us to write $P \circ PV$, with the meaning of $(P \circ PV)_i = P \circ PV_i$. There is a direct relationship between tabular predicates and tabular relations. Let pv and qv be vectors of predicates, let rm be a matrix of predicates, let PV and QV be vectors of relations, and let RM be a matrix of relations. If

$$PV_i \ x \ y \equiv pv_i, \quad QV_i \ x \ y \equiv qv_i, \quad RM_{i,j} \ x \ y \equiv rm_{i,j}$$

then the following two definitions of relation S are equivalent:

$$S = \frac{PV}{PV} \mid \frac{QV}{RM}, \quad S \ x \ y \equiv \frac{qv}{pv} \mid \frac{qv}{rm}$$

This relationship between tabular predicates and tabular relations allows us to switch between them as convenient. This also allows us to lift all theorems on tabular predicates to tabular relations as needed. In particular the notions of disjointness and coverage carry over to relations.

Operations on tabular relations are:

Theorem 4.1 (Table Domain).

$$\Delta\left(\frac{}{BV} \middle| \frac{CV}{PM}\right) = \frac{}{BV} \middle| \frac{CV}{\Delta PM}$$

Theorem 4.2 (Table Composition).

$$\begin{aligned} \text{(a)} \quad & \frac{}{BV} \middle| \frac{CV}{PM} \circ Q = \frac{}{BV} \middle| \frac{CV}{PM \circ Q} \\ \text{(b)} \quad & S \circ \frac{}{PV} \middle| \frac{QV}{RM} \subseteq \frac{}{S \circ PV} \middle| \frac{S \circ QV}{S \circ RM} \\ \text{(c)} \quad & S \circ \frac{}{PV} \middle| \frac{QV}{RM} \equiv \frac{}{S \circ PV} \middle| \frac{S \circ QV}{S \circ RM} \quad \text{if } S \text{ is functional} \end{aligned}$$

3.5 Tabular Verification

A typical use of weakest preconditions is for checking invariance properties: an operation P *establishes* condition C if $[P]C = \top$ and P *preserves* C if $C \subseteq [P]C$. Consequently we give theorems for deducing that a weakest precondition—if expressed as a predicate—is either universally true or is weaker than a given precondition. We make use of the following facts about weakest preconditions. Assume P, P_i are relations, for an arbitrary index set I , and B, C are conditions:

Lemma 5.1.

$$\begin{aligned} \text{(a)} \quad & [\cup i \in I \cdot P_i]C = \cap i \in I \cdot [P_i]C \\ \text{(b)} \quad & [B \cap P]C = \overline{B} \cup [P]C \end{aligned}$$

We give some theorems for determining weakest preconditions of operations in tabular form. For a matrix PV and a condition C let $[PM]C$ stand for PM with the

weakest precondition applied to each element, formally $([PM]C)_{i,j} = [PM_{i,j}]C$:

Theorem 5.2 (Tabular Weakest Precondition).

$$\begin{aligned}
 \text{(a)} \quad & \frac{}{BV \mid [PM]C} \subseteq \left[\frac{}{BV \mid PM} \right] C \quad \text{if } BV, CV \text{ are total} \\
 \text{(b)} \quad & \frac{}{BV \mid [PM]C} \supseteq \left[\frac{}{BV \mid PM} \right] C \quad \text{if } BV, CV \text{ are jointly disjoint} \\
 \text{(c)} \quad & \frac{}{BV \mid [PM]C} = \left[\frac{}{BV \mid PM} \right] C \quad \text{if } BV, CV \text{ are total and jointly disjoint.}
 \end{aligned}$$

We note that typically only (c) is useful as (a) results in a precondition may be too restrictive and (b) may not result in a precondition for the given postcondition at all. While (c) allows the precondition to be determined by considering each case in the body of the program in turn, it does have the side conditions of totality and disjointness. We give an alternative theorem that does not have these side conditions but allows only inclusion to be shown, although it gives a necessary and sufficient condition for it. Thus it can always be used to verify that a tabular relation under a given precondition establishes a given postcondition:

Theorem 5.3 (Tabular Verification).

$$B \subseteq \left[\frac{}{BV \mid PM} \right] C \equiv i, j \cdot B \cap BV_i \cap CV_j \subseteq [PM_{i,j}]C$$

For the case that the table is given by a tabular predicate and the postcondition by a predicate, we can give the analogue of Theorem 5.2. For brevity, we give only the analogue of Theorem 5.2(c). We assume that the state consists of a vector xv of variables. If pm is a matrix of predicates, we write $\forall x \cdot pm$ for every matrix element universally quantified over x , formally $(\forall x \cdot pm)_{i,j} \equiv (\forall x \cdot pm_{i,j})$. Let $f[xv \setminus ev]$ stand for expression f with each variable in xv simultaneously substituted by the corresponding expressions in ev .

Theorem 5.4 (Weakest Precondition with Predicates). If standard relation P and condition C are given by

$$P \ xv \ xv' \equiv \frac{}{bv \mid pv} \frac{cv}{} , \quad C \ xv \ xv' \equiv c$$

and if bv, cv are total and jointly disjoint we have

$$[P]C \ xv \ xv' \equiv \frac{}{bv \mid \forall xv' \cdot pm \Rightarrow c[xv \setminus xv']} \frac{cv}{}$$

Next we give a theorem that does not have the side conditions of totality and disjointness of the headers and does not even require the table to be in standard form. Hence it can also be applied to inverted tables:

Theorem 5.5 (Tabular Verification with Predicates). If conditions B, C and relation P are given by

$$B \ xv \ xv' \equiv b , \quad P \ xv \ xv' \equiv \frac{}{pv \mid rm} \frac{qv}{} , \quad C \ xv \ xv' \equiv c$$

we have

$$B \subseteq [P]C \equiv \wedge i, j \cdot b \wedge pv_i \wedge qv_j \wedge rm_{i,j} \Rightarrow c[xv \setminus xv']$$

For an operation given by a vector table we have a simplified rule for determining its precondition. Let $f[xv \setminus em]$ stand for a vector of expressions, with each element obtained by substituting xv with one column of matrix em in f , formally $(f[xv \setminus em])_j = f[xv \setminus em^j]$.

Theorem 5.6 (Weakest Precondition of Vector Table). If standard vector relation V and condition C are given by

$$V \ xv \ xv' \equiv \frac{}{xv' = \mid} \frac{bv}{em} , \quad C \ xv \ xv' \equiv c$$

we have

$$[V]C \ xv \ xv' \equiv \wedge j \cdot bv_j \Rightarrow c[xv \setminus em^j]$$

While the theorem allows the precondition to be calculated, the precondition is a conjunction rather than a table. We can give an alternative theorem that gives a tabular precondition but has side conditions:

Theorem 5.7 (Tabular Weakest Precondition of Vector Table). If standard vector relation V and condition C are given by

$$V \text{ } xv \text{ } xv' \equiv \frac{\quad}{xv' = \mid \begin{array}{c} bv \\ em \end{array}}, \quad C \text{ } xv \text{ } xv' \equiv c$$

and if bv is total and disjoint we have

$$[V]C \text{ } xv \text{ } xv' \equiv \frac{bv}{c[xv \setminus em]}$$

Finally we give a theorem that does not have the side conditions of totality and disjointness. It follows directly from Theorem 5.5.

Theorem 5.8 (Verification with Vector Table). If conditions B, C and vector relation V are given by

$$B \text{ } xv \text{ } xv' \equiv b, \quad V \text{ } xv \text{ } xv' \equiv \frac{\quad}{xv' = \mid \begin{array}{c} pv \\ em \end{array}}, \quad C \text{ } xv \text{ } xv' \equiv c$$

we have:

$$B \subseteq [V]C \equiv \wedge j \cdot b \wedge pv_j \Rightarrow c[xv \setminus em^j]$$

3.6 Refinement

In general, refinement is the process of deriving an implementation from a specification and verifying the correctness of the derivation [20].

Programs are given a new semantics with the merit that a specification written as a first-order predicate can be refined, step by step, to a program via the rules of Predicate Calculus. The semantics allows a free mixture of predicate and programming notations, and manipulation of programs [10].

We formalize the notation of program refinement defined on partial relations [34]. We say that S is *refined by* T , written $S \sqsubseteq T$, if

1. $\mathbf{dom} S \subseteq \mathbf{dom} T$
2. $|\mathbf{dom} S| \cap T \subseteq S$

The operator $|s|$ "lifts" a set to a relation, $|s| x y = (x \in s)$.

Algorithmic refinement is simplified assuming program P and Q are defined on total relations with the same domain. If $P \subseteq Q$, then we say that P *refines* Q [32]. Refinement is a process that allows non-determinism to be reduced. Refinement is reflexive, $P \subseteq P$, meaning that each programs is refined by itself. Refinement is also transitive, $P \subseteq Q$ and $Q \subseteq R$ implies $P \subseteq R$, meaning that programs can be refined in a stepwise manner. If $P \subseteq Q$ holds, then P is called the (more) concrete and Q the (more) abstract program.

Data refinement is the systematic replacement of a state space (abstract data) by another one (concrete data) in program development. Data refinement is considered as an operator rather than a relation within the *refinement calculus* framework [4]. The *encoding* operator \downarrow is defined so that $S \downarrow D$ is the most general (least refined) data refinement of statement S with respect to an *abstraction statement* D (an abstraction statement models the relationship between the concrete and the abstract state space). Using Galois connections it is found that under certain restrictions there exists a dual *decoding* operator \uparrow which allows us to calculate the least general (most refined) abstraction $S \uparrow D$ of a given (concrete) statement S with respect to abstraction statement D .

We consider two variants of data refinement, *downward (forward) data refinement* and *upward (backward) data refinement* [9]. The encoding and decoding operators are similar to those of [4]. Suppose P, Q are homogeneous relations of possibly different types, an *encoding operator* $P \downarrow R$ is introduced:

$$P \downarrow R = R^{-1} \circ P \circ R \quad \text{provided } R \text{ is injective}$$

Program Q downward refines program P via relation R if $R \circ Q \subseteq P \circ R$ holds.

Symmetrically, a *decoding operator* $P \uparrow R$ is introduced:

$$P \uparrow R = R^{-1} \circ P \circ R \quad \text{provided } R \text{ is total}$$

Program P upward refines program Q via relation R if $P \circ R \subseteq R \circ Q$ holds.

If R is either injective or total, a *coding operator* is introduced:

$$P \downarrow R = R^{-1} \circ P \circ R$$

We give a theorem that apply only to encoding:

Theorem 6.1 (Soundness of Encoding).

$$Q \subseteq P \downarrow R \Rightarrow R \circ Q \subseteq P \circ R \quad \text{if } R \text{ is injective}$$

We give a theorem that apply only to decoding:

Theorem 6.2 (Soundness of Decoding).

$$P \uparrow R \subseteq Q \Rightarrow P \circ R \subseteq R \circ Q \quad \text{if } R \text{ is total}$$

We note that coding is monotonic in its first argument (but not in its second), which follows directly from its definition:

Theorem 6.3 (Monotonicity of Coding).

$$P \subseteq Q \Rightarrow P \downarrow R \subseteq Q \downarrow R$$

We state some facts about the first argument of the coding operator.

Theorem 6.4. Suppose I is an index set and C is a condition:

- (a) $\perp \downarrow R = \perp$
- (b) $(\cup i \in I \cdot P_i) \downarrow R = (\cup i \in I \cdot P_i \downarrow R)$
- (c) $(\cap i \in I \cdot P_i) \downarrow R \subseteq (\cap i \in I \cdot Q_i \downarrow R)$
- (d) $(C \cap P) \downarrow R \subseteq (R^{-1} \circ C) \cap (P \downarrow R)$

We note that for a relation R and condition C , the condition $R^{-1} \circ C$ is the image of C under R . As Theorem 6.4(c) and (d) state inclusion and not equality, they are only useful for decoding when distributing the decoding operator into conjunctions. Next we state how coding behaves in its second argument:

Theorem 6.5 Suppose I is an index set:

- (a) $P \uparrow \perp = \perp$
- (b) $P \uparrow \top = \top$ if $P \neq \perp$
- (c) $P \uparrow Id = P$
- (d) $P \uparrow (R \circ S) = (P \uparrow R) \uparrow S$
- (e) $(\cup i \in I \cdot P \uparrow R_i) \subseteq P \uparrow (\cup i \in I \cdot R_i)$
- (f) $P \uparrow (\cap i \in I \cdot R_i) \subseteq (\cap i \in I \cdot P \uparrow R_i)$

We continue with theorems that apply only to encoding. Distributivity through conjunctions in the first argument can be strengthened to equality with an injective encoding relation. Encoding subdistributes through relational composition:

Theorem 6.6 Suppose R is an injective relation:

- (a) $(\cap i \in I \cdot P_i) \downarrow R = (\cap i \in I \cdot Q_i) \downarrow R$
- (b) $(C \cap P) \downarrow R = (R^{-1} \circ C) \cap (P \downarrow R)$
- (c) $(P_1 \downarrow R) \circ (P_2 \downarrow R) \subseteq (P_1 \circ P_2) \downarrow R$

We conclude with a theorem that applies only to decoding. Decoding also subdistributes through relational composition, though in the other direction than encoding:

Theorem 6.7 Suppose R is a total relation:

$$(P_1 \circ P_2) \uparrow R \subseteq (P_1 \uparrow R) \circ (P_2 \uparrow R)$$

3.7 Tabular Refinement

We give theorems on how specifications can be transformed into more concrete or more abstract ones, where either the concrete or the abstract or both are given in tabular form.

First we consider that both specifications are over the same state space. Assume PV and QV are vectors of relations, RM is a matrix of relations, and S is a relation:

Theorem 7.1 (Refining to Table).

$$\begin{aligned} \text{(a)} \quad & \frac{}{PV \mid RM} \frac{QV}{RM} \subseteq S \equiv \wedge i, j. PV_i \cap QV_j \cap RM_{i,j} \subseteq S \\ \text{(b)} \quad & \frac{}{PV \mid RM} \frac{QV}{RM} \subseteq \frac{}{PV \mid SM} \frac{QV}{SM} \Leftarrow \wedge i, j. PV_i \cap QV_j \cap RM_{i,j} \subseteq SM_{i,j} \end{aligned}$$

Refining to a vector table allows for a simplified rule:

Theorem 7.2 (Refining to Vector Table). If vector relation P and relation Q are given by

$$P \ xv \ xv' \equiv \frac{}{xv' = \mid \frac{pv}{em}}, \quad Q \ xv \ xv' \equiv \frac{pv}{qv}$$

we have

$$P \subseteq Q \Leftarrow (\wedge j. pv_j \Rightarrow qv[xv' \setminus em^j])$$

Note that while above theorem can be applied even if Q is not a standard relation, P is a standard vector relation only if Q is a standard relation. We now give a general theorem when the concrete and abstract state are related through relation R :

Theorem 7.3 (Data Refining a Table). Assume BV, CV are vectors of conditions:

$$\begin{aligned} \text{(a)} \quad & \left(\frac{}{BV \mid PM} \frac{CV}{PM} \right) \downarrow R = \frac{}{R^{-1} \circ BV \mid PM \downarrow R} \frac{R^{-1} \circ CV}{PM \downarrow R} \quad \text{if } R \text{ is injective.} \\ \text{(b)} \quad & \left(\frac{}{BV \mid PM} \frac{CV}{PM} \right) \uparrow R = \frac{}{R^{-1} \circ BV \mid PM \uparrow R} \frac{R^{-1} \circ CV}{PM \uparrow R} \quad \text{if } R \text{ is injective.} \end{aligned}$$

To allow a direct application of above theorem, we derive the corresponding theorem when the relation is given by a tabular predicate. We extend the use of existential quantifications to matrices of predicates, with the meaning that the quantification is applied to each element, formally $(\exists x \cdot pm)_{i,j} \equiv (\exists x \cdot pm_{i,j})$:

Theorem 7.4 (Data Refining with Predicates). Given relation P in standard form and relation R by

$$P \ xv \ xv' \equiv \frac{}{bv} \left| \frac{cv}{pm} \right., \quad R \ xv \ yv \equiv r$$

and writing r' for r with xv, yv substituted by xv', yv' we have:

$$\begin{aligned} \text{(a) } (P \downarrow R) \ yv \ yv' &= \left(\frac{}{\exists xv \cdot r \wedge bv} \left| \frac{\exists xv \cdot r \wedge cv}{\exists xv, xv' \cdot r \wedge pm \wedge r'} \right. \right) \\ &\text{if } R \text{ is injective.} \\ \text{(b) } (P \uparrow R) \ yv \ yv' &\subseteq \left(\frac{}{\exists xv \cdot r \wedge bv} \left| \frac{\exists xv \cdot r \wedge cv}{\exists xv, xv' \cdot r \wedge pm \wedge r'} \right. \right) \end{aligned}$$

We consider the case that the refinement relation rather than the specification is in tabular form. More precisely, we consider the refinement relation being defined by an inverted vector table, that is a table in which only the variables of the initial state appear in the left header and variables of the final state appear only in the upper header and body. For simplicity we consider a refinement relation with only two columns.

Theorem 7.5 (Data Refinement with Vector Table). Assume inverted vector relation R is given by:

$$R \ xv \ yv \equiv \frac{}{xv =} \left| \frac{c}{ev} \right| \frac{d}{fv}$$

Writing c', d', ev', fv' for c, d, ev, fv with yv substituted by yv' we have

$$(P \uparrow R) \ yv \ yv' \equiv \frac{}{c} \left| \frac{c'}{P \ ev \ ev'} \right| \frac{d'}{P \ ev \ fv'}$$

Chapter 4

Design Features

4.1 Interface with Simplify

Simplify accepts a sequence of first order formulas as input, and attempts to prove each one [21].

Simplify [-print] [-ax *axfile*] [-nosc] [-noprun] [-help] [-version] [*file*]

Simplify implements a semi-decision procedure for its inputs: it can sometimes fail to prove a valid formula. But it is conservative in that it never claims that an invalid formula is valid [21]. Simplify handles propositional connectives by backtracking search and includes complete decision procedures for the theory of equality and for linear rational arithmetic, together with some heuristics for linear integer arithmetic that are not complete. Simplify's handling of quantifiers by pattern-driven instantiation is also incomplete [7].

Complex valid formulae including quantifiers may require much longer running time for Simplify to prove, or even cause Simplify fail to prove their correctness. If Simplify can prove the formula, it prints valid. If it cannot prove the formula, it normally prints a conjunction of literals that it believes to satisfy the negation of the

formula.

Three options are used in our application:

1. The *-nosc* options causes Simplify to simply output "valid" or "invalid";
2. The *-ax* flag allows us to specify an alternate axiom set, and the AXIOMDIR environment variable allows us to specify where Simplify should look for that axiom set.
3. The *file* argument is provided such that S-expression formulae are read one at a time from the file, and proved.

The syntax of formulae is based on S-expressions, with one S-expression per formula.

```

formula ::= "(" ( AND | OR ) { formula } ")" |
           "(" NOT formula ")" |
           "(" IMPLIES formula formula ")" |
           "(" IFF formula formula ")" |
           "(" FORALL "(" var* ")" formula ")" |
           "(" EXISTS "(" var* ")" formula ")" |
           "(" PROOF formula* ")" |
           literal

literal  ::= "(" ( "EQ" | "NEQ" | "<" | "<=" | ">" | ">=" )
           term term ")" |
           "(" "DISTINCT" term term+ ")" |
           "TRUE" | "FALSE" | <propVar>

```

```

term    ::= var | integer | "(" func { term } ")"

```

"var"'s (variables), "func"'s (functions), and "propVar"'s (propositional variables) are represented as "Atom.T"'s.

The formula

```
(DISTINCT term1 ... termN)
```

represents a conjunction of distinctions between all pairs of terms in the list.

<funcs>'s are uninterpreted, except for "+", "-", and "**", which represent the obvious operations on integers. "/" is interpreted by our self-defined functions in our axioms.

4.2 Pattern Matching in Function Definition

The input to Simplify is a formula of untyped first-order logic with function and relations, including equality. That is, the language includes the propositional connectives $\wedge, \vee, \neg, \implies$, and \Leftrightarrow ; the universal quantifier \forall , and the existential quantifier \exists [7]. To define functions we need to apply universal quantifier to all independent variables such that the variable can be replaced by any value. Simplify handles quantifiers by **pattern-driven instantiation**. Pattern matching starting with keyword **PATS** is used to find the relevant structure and to substitute the matching part (function name and its inputs) with function outputs. Pattern matching can benefit from guard. Guards can be used to augment pattern matching with the possibility to skip a pattern even if the structure matches. Guards are realized in Simplify by the left part of implication. Following S-expression in Simplify is a function which defines the arithmetic division operation on two positive integers.

```
(FORALL (x y)
  (PATS (div1 x y))
  (AND (IMPLIES (>= x y) (EQ (div1 x y) (+ (div1 (+ x (* -1 y)) y) 1)))
    (IMPLIES (< x y) (EQ (div1 x y) 0)))))
```

This is a recursive definition and similar to the respective mathematic notation:

$$\text{div1}(x, y) = \begin{cases} \text{div1}(x - y) + 1 & x \geq y \\ \text{div1}(x, y) = 0 & x < y \end{cases}$$

An absolute function can be defined without pattern matching since the disjunction of the left side of all the implication are total:

```
(FORALL (x y)
  (AND (IMPLIES (>= x 0) (EQ (abs x) x))
        (IMPLIES (< x 0) (EQ (abs x) (* -1 x))))))
```

A division function on two integers are extended to:

```
(FORALL (x y)
  (PATS (/ x y))
  (AND (IMPLIES (AND (>= x 0) (> y 0))
                (EQ (/ x y) (div1 x y)))
        (IMPLIES (AND (>= x 0) (< y 0))
                (EQ (/ x y) (* -1 (div1 x (abs y)))))
        (IMPLIES (AND (< x 0) (> y 0))
                (EQ (/ x y) (* -1 (div1 (abs x) y))))
        (IMPLIES (AND (< x 0) (< y 0))
                (EQ (/ x y) (div1 (abs x) (abs y))))))
```

The pattern matching here plays a role of guard to skip a division x/y with $y = 0$. Note that if a pattern matches a function defined on the right side of an implication, then the left side of the implication states the domain of the function. It is similar to an alternative statement. These functions are the additional axioms to which Simplify refers before validating predicates.

4.3 Unified Data Type

Our data type *form* is defined to match Simplify grammar. Besides that, we also add properties and operations of sets to *form*. There are two kinds of variables—variables in the left header of a vector table, represented by *VECVAR*("varname") and otherwise, represented by *VAR*("varname"). Another type expression *OP* is applied in Chapter 6. *OP* takes a predicate name and a predicate definition as its parameter, passes the predicate definition as inputs to Simplify but prints the predicate names to the screen and L^AT_EX files. Such a use is also sugar for reading the foregone predicate into the current background predicate by our parser.

For simplicity we only define the data type of two-dimensional tables; one dimensional table can be deemed as a special two-dimensional table with row header or column header being TRUE and other columns or rows of table body being copied from the first column or row. Table structure is modeled by a *TABLE* construct followed by a record as its parameter which consists of left header, upper header and table body. Row header and column header are expressed by one dimensional arrays; table body is expressed by a two dimensional array. Elements of arrays are predicates including tables. So this is a mutually recursive definition.

Among ten kinds of tables summarized by Parnas [23], *TABLE* construct can represent normal, inverted and vector function tables with integer return type, normal, inverted and vector relation tables, predicate expression tables, and characteristic predicate tables. In this thesis we use one class of tables, called *characteristic predicate tables*, together with its variation of vector tables.

```
type form = CONST of int
  | VAR of string
  | VECVAR of string
  | SUM of form × form
```


| *DIFF* of *form* \times *form*
 | *PROD* of *form* \times *form*
 | *QUOT* of *form* \times *form*
 | *FUN* of *string* \times *form list*
 | *OP* of *string* \times *form*
 | *EMPTY*
 | *INSERT* of *form* \times *form*
 | *DELETE* of *form* \times *form*
 | *MEMBER* of *form* \times *form*
 | *UNION* of *form* \times *form*
 | *SUBSET* of *form* \times *form*
 | *TRUE*
 | *FALSE*
 | *EQ* of *form* \times *form*
 | *NEQ* of *form* \times *form*
 | *LT* of *form* \times *form*
 | *LE* of *form* \times *form*
 | *GT* of *form* \times *form*
 | *GE* of *form* \times *form*
 | *AND* of *form list*
 | *OR* of *form list*
 | *IMPLIES* of *form* \times *form*
 | *IFF* of *form* \times *form*
 | *NOT* of *form*
 | *FORALL* of *form list* \times *form*
 | *EXISTS* of *form list* \times *form*
 | *TABLE* of *tables*
 and *tables* = {*headm* : *form array*; *headn* : *form array*; *body* : *form array array*}

4.4 Variable Types of Theorems

In the logic of [32] and this thesis, typing is implicit, so all variables of the theorems (distinguishing from program variables) are implicitly (if not explicitly) universal

quantified over the *right* type. In Simplify, individual variables range over the space of individual values which includes integers and maps [7].

If a program variable is of numerical or string type, we can simply map it to the space of individual values with the same name; if a program variable belongs to boolean or enumeration type where the *domain* of the program variable may have effect on the behavior of the program, we have to explicitly state the space of values of that type. In Simplify, we do this by two steps:

1. make all values in *Type* space distinct by a *general distinction* of the form $DISTINCT(t1, ..., tn)$.
2. make the variable t total over its *Type* by a *disjunction* of the form $OR(t = t1, ..., t = tn)$

For each *typed* program variable, we set these two formulae into our self-defined axiom file and they will be loaded each time the main program is initialized. One typical example is modeling car seat movement where longitudinal adjustment motor only has three states—*forward*, *backward* and *stop*. Another example is elevator button pressed refinement where program variable r takes a boolean value *true* or *false*. These two examples will be illustrated in the following chapters.

If a program variable x in specification Z has a type Y , Y is a set and the elements of Y are not specified, we can limit the domain of x by the predicate $x \in Y \implies Z$.

If a program variable is of abstract type (e.g. set), functions have to be defined to model properties and operations of that type (e.g. insert, delete and member). A program variable of composite type (e.g. tuple and array) follow the same rule as that of abstract type. These functions are also stored in our axiom file. The modeling of visitor information system in Chapter 7 illustrates how to define functions that specify the properties and operations of set and relation (a set of pairs). Note that

a program variable of initial and final states should be typed if they both occur in a specification.

Type checking is not implemented in our application of theorems since the types of program variables are confined by functions.

4.5 Structure of Implementation

Our implementation of theorems in OCaml includes three parts:

1. Parser
2. Printing
3. Theorem proving

We introduce each part briefly in this section.

4.5.1 Parser

The idea of designing a parser comes from difficulties we met when inputting a long and complicated predicate, especially a table or a set manipulation, directly by our data type *form*. The parser also assists in better understanding and error checking procedures or operations in our examples.

The characters are first *scanned*: processed into *tokens* such as keywords, identifiers, special symbols and numbers. The parser is supplied a list of tokens. A token is either an identifier, an integer constant or a keyword. Calling *scan* performs lexical analysis on a string and returns the resulting list of tokens [28].

We use *recursive decent parsing* technique [33] to construct a top-down parser directly in OCaml. Literal *TRUE* and *FALSE* are inputted as they are; variable

| Precedence | Operator | Input | Example | <i>form</i> expressions |
|------------|-------------------|-------|-------------------------------|------------------------------------|
| 0 | () | () | $(a + b)/4$ | QUOT(ADD(a, b),4) |
| 1 | \times | * | $a * b$ | PROD(a, b) |
| 1 | \div | / | a/b | QUOT(a, b) |
| 2 | + | + | $a + b$ | ADD(a, b) |
| 2 | - | - | $a - b$ | SUB(a, b) |
| 3 | = | = | $a = b$ | EQ(a, b) |
| 3 | \neq | /= | $a / = b$ | NEQ(a, b) |
| 3 | < | < | $a < b$ | LT(a, b) |
| 3 | \leq | <= | $a <= b$ | LE(a, b) |
| 3 | > | > | $a > b$ | GT(a, b) |
| 3 | \geq | >= | $a >= b$ | GE(a, b) |
| 4 | \neg | not | not $a > b$ | NOT(GT(a, b)) |
| 5 | \wedge | & | $a > 0 \& b > 0$ | AND([GT($a, 0$);GT($b, 0$)]) |
| 6 | \vee | or | $a > 0$ or $b > 0$ | OR([GT($a, 0$);GT($b, 0$)]) |
| 7 | \implies | => | $a > 0 \implies b > 0$ | IMPLIES(GT($a, 0$),GT($b, 0$)) |
| 7 | \Leftrightarrow | <=> | $a > 0 \Leftrightarrow b > 0$ | IFF(GT($a, 0$),GT($b, 0$)) |
| 8 | \forall | ! | ! $x x > 0$ | FORALL([x],GT($x, 0$)) |
| 8 | \exists | # | # $x, y x > y$ | EXISTS([$x; y$],GT(x, y)) |

Table 4.1: Logical Operators and Quantifiers

VAR("id") is inputted as *id*; variable VECVAR("id") in the left header of a vector table is inputted as *id* = ; integer CONST(12) is inputted as 12. Tables 3.1 list logical operators and quantifiers.

The inputting environment of tables is similar to the *tabular* environment in L^AT_EX typesetting. The keyword *BEGTAB* starts a tabular input environment while *ENDTAB* finishes it. The left header starts by keyword *LHEADER*, ends by symbol //, and its elements are separated by symbol \$; Upper header start by keyword *UHEADER*, ends by symbol //, and its elements are separated by symbol \$; there is no prefix keyword for the tabular body, symbol \$ is used to separate elements of each row, symbol // starts a new line. Left header or upper header can be omitted but table body can not. For example, Table 4.2 can be inputted by

| | $c > 0$ | $d > 0$ | $e > 0$ |
|---------|---------|---------|---------|
| $a > 0$ | $x < y$ | $y < z$ | $x < z$ |
| $b > 0$ | $y < z$ | $x < y$ | $x < z$ |

Table 4.2: A Sample Table

```
"BEGTAB LHEADER  $a > 0$  $  $b > 0$  // UHEADER  $c > 0$  $  $d > 0$  $  $e > 0$  //
 $x < y$  $  $y < z$  $  $x < z$  //  $y < z$  $  $x < y$  $  $x < z$  // ENDTAB"
```

A vector table is a table in which only the variables of the initial state appear in the left header and variables of the final state appear only in the upper header and body. In this project, vector tables have the same input environment as normal tables except that the left header should be an identifier followed by symbol = and table body is a matrix of arithmetic expressions. For instance, the input of Table 4.3 is "BEGTAB LHEADER $x =$ \$ $y = z$ // UHEADER $c > 0$ \$ $d > 0$ \$ $e > 0$ // 3 \$ 7 \$ 9 // 2 \$ 4 \$ 8 //ENDTAB"

| | $c > 0$ | $d > 0$ | $e > 0$ |
|---------|---------|---------|---------|
| $x =$ | 3 | 7 | 9 |
| $y = z$ | 2 | 4 | 8 |

Table 4.3: A Sample Vector Table

4.5.2 Notations for Sets and Relations

Other *form* expressions relate to the properties and operations of set, relation and function. We list their input syntax, their meaning is explained by plain words or conventional symbols, and translation into our *form* type expressions.

- {}

– An empty set ϕ .

– EMPTY

54

- $\{x_1, x_2, \dots, x_n\}$

– Inserting element x_1, x_2, \dots, x_n sequentially into Empty set

- (INSERT (... (INSERT (INSERT EMPTY x_1) x_2)...) x_n)
- $xs - ys$
 - Deleting elements of set ys from set xs (i.e. $xs - ys$).
 - DELETE(xs, ys)
- $xs \cup ys$
 - The union of set xs and set ys (i.e. $xs \cup ys$).
 - UNION(xs, ys)
- $xs \subseteq ys$
 - If xs is a subset of ys (i.e. $xs \subseteq ys$), then it is evaluated to true.
 - EQ(SUBSET($xs; ys$), VAR("true"))
- $x \in xs$
 - If x is an element of set xs (i.e. $x \in xs$), then it is evaluated to true.
 - MEMBER(xs, ys), VAR("true"))
- PAIR(a, b)
 - A relation is represented by a set of pairs (i.e. $\{(a_1, b_1), (a_2, b_2)\}$).
 - FUN("PAIR", [$x; y$])
- dom(xs)
 - The domain of binary relation xs .
 - FUN("dom", [xs])

- $\text{ran}(xs)$
 - The range of binary relation xs .
 - $\text{FUN}(\text{"ran"}, [xs])$
- $\text{relate}(x, xs)$
 - A relation which is the subset of binary relation xs where the domain of the relation is $\{x\}$.
 - $\text{FUN}(\text{"relate"}, [x; xs])$
- $\text{revrelate}(x, xs)$
 - A relation which is the subset of binary relation xs where the range of the relation is $\{x\}$.
 - $\text{FUN}(\text{"revrelate"}, [x; xs])$
- $\text{injective}(xs)$
 - If binary relation xs is injective, then it evaluates to true.
 - $\text{FUN}(\text{"injective"}, [xs])$
- $\text{map}(xs)$
 - If binary relation xs is a function, then it evaluates to true.
 - $\text{FUN}(\text{"map"}, [xs])$
- $\text{card}(xs)$
 - The cardinality of set xs (i.e. $\#xs$).
 - $\text{FUN}(\text{"card"}, [xs])$

- `compose(xs, ys)`
 - The composition of the relations *xs* and *ys*.
 - `FUN("compose", [xs])`
- `EMPSTR`
 - The empty string `""`.
 - `EMPSTR`
- `concat(xs)`
 - Concatenate a set of strings separated with spaces.
 - `FUN("concat", [xs])`

4.5.3 Printing

Module `Print` fulfills three printing functions:

1. Function *prints* with parameters *predi* and *margin*

One auxiliary function *predtable* takes a predicate table and returns a plain predicate corresponding to the table. Function *prints* takes a predicate with possible embedded tables and returns an equivalent predicate without tables, printing that predicate in Simplify syntax based on S-expression. A formula is printed in a nested structure and a margin is added to the output file if needed.


```

let predtable t1 =
  let t2 = ref [] in
  for i = 0 to (Array.length t1.body) - 1 do
    for j = 0 to (Array.length t1.body.(0)) - 1 do
      if t1.headm ≠ [] ∧ t1.headn ≠ [] then
        match t1.headm.(i) with
          | VECVAR(x) → t2 := AND([t1.headn.(j); EQ(VAR(x),
            t1.body.(i).(j))] :: !t2
          | _ → t2 := AND([t1.headm.(i); t1.headn.(j); t1.body.(i).(j)] :: !t2
        else if t1.headm = [] ∧ t1.headn ≠ [] then
          t2 := AND([TRUE; t1.headn.(j); t1.body.(i).(j)] :: !t2
        else if t1.headm ≠ [] ∧ t1.headn = [] then
          (match t1.headm.(i) with
            | VECVAR(x) → t2 := EQ(VAR(x), t1.body.(i).(j)) :: !t2
            | _ → t2 := AND([t1.headm.(i); TRUE; t1.body.(i).(j)] :: !t2)
          else
            t2 := AND([TRUE; TRUE; t1.body.(i).(j)] :: !t2
        done
    done;
  OR(!t2)

```

```

let prints predi margin =
  let space = ref margin in
  let flag = ref 0 in
  let rec transformlist = function
    | [] → []
    | head :: tail → transstring head :: transformlist tail
  and transstring = function
    | CONST(x) → string_of_int x
    | VAR(x) → if x = "" then "EMPSTR" else x
    | SUM(term1, term2) →
      "(" ^ transstring term1 ^ "+" ^ transstring term2 ^ ")"
    | DIFF(term1, term2) →
      "(" ^ transstring term1 ^ "-" ^ transstring term2 ^ ")"

```

```

| DELETE(term1, term2) →
    "(DELETE□" ^ transstring term1 ^ "□" ^ transstring term2 ^ ")"
| PROD(term1, term2) →
    "(*□" ^ transstring term1 ^ "□" ^ transstring term2 ^ ")"
| QUOT(term1, term2) →
    "(/□" ^ transstring term1 ^ "□" ^ transstring term2 ^ ")"
| FUN(x, termlist) → (String.concat "" ["("; x; "□";
    String.concat "□" (transformlist termlist))] ^ ")")
| EMPTY → "EMPTY"
| INSERT(term1, term2) →
    "(INSERT□" ^ transstring term1 ^ "□" ^ transstring term2 ^ ")"
| MEMBER(term1, term2) →
    "(MEMBER□" ^ transstring term1 ^ "□" ^ transstring term2 ^ ")"
| UNION(term1, term2) →
    "(UNION□" ^ transstring term1 ^ "□" ^ transstring term2 ^ ")"
| SUBSET(term1, term2) →
    "(SUBSET□" ^ transstring term1 ^ "□" ^ transstring term2 ^ ")"
| TRUE →
    if !flag = 1 then begin space := !space ^ "□□";
        let s = "\n" ^ !space ^ "TRUE" in
            space := String.sub !space 0 (String.length !space - 2); s end
        else begin let s = !space ^ "TRUE" in flag := 1; s end
| FALSE →
    if !flag = 1 then begin space := !space ^ "□□";
        let s = "\n" ^ !space ^ "FALSE" in
            space := String.sub !space 0 (String.length !space - 2); s end
        else begin let s = !space ^ "FALSE" in flag := 1; s end
| EQ(term1, term2) →
    space := !space ^ "□□";
    let s = "\n" ^ !space ^ "(EQ□" ^ transstring term1 ^ "□"
        ^ transstring term2 ^ ")"
    in space := String.sub !space 0 (String.length !space - 2); s
| NEQ(term1, term2) →
    space := !space ^ "□□";

```

```

    let s = "\n" ^ !space ^ "(NEQ_" ^ transstring term1 ^
        "_" ^ transstring term2 ^ ")"
    in space := String.sub !space 0 (String.length !space - 2); s
| LT(term1, term2) →
    space := !space ^ "_";
    let s = "\n" ^ !space ^ "<_" ^ transstring term1 ^
        "_" ^ transstring term2 ^ ")"
    in space := String.sub !space 0 (String.length !space - 2); s
| LE(term1, term2) →
    space := !space ^ "_";
    let s = "\n" ^ !space ^ "<=__" ^ transstring term1 ^
        "_" ^ transstring term2 ^ ")"
    in space := String.sub !space 0 (String.length !space - 2); s
| GT(term1, term2) →
    space := !space ^ "_";
    let s = "\n" ^ !space ^ ">_" ^ transstring term1 ^
        "_" ^ transstring term2 ^ ")"
    in space := String.sub !space 0 (String.length !space - 2); s
| GE(term1, term2) →
    space := !space ^ "_";
    let s = "\n" ^ !space ^ ">=__" ^ transstring term1 ^ "_" ^
        transstring term2 ^ ")"
    in space := String.sub !space 0 (String.length !space - 2); s
| AND(formlist) →
    if !flag = 1 then begin space := !space ^ "_";
        let s = "\n" ^ !space ^ "(AND" ^
            (String.concat "_" (transformlist formlist)) ^ ")" in
        space := String.sub !space 0 (String.length !space - 2);
        s end else begin flag := 1; let s = (!space ^ "(AND" ^
            (String.concat "_" (transformlist formlist)) ^ ")")
        in s end
| OR(formlist) →
    if !flag = 1 then begin space := !space ^ "_";
        let s = "\n" ^ !space ^ "(OR" ^

```

```

        (String.concat "⊂" (transformlist formlist)) ^ ")" in
        space := String.sub !space 0 (String.length !space - 2); s
    end else begin
        flag := 1; let s = (!space ^ "(OR" ^ (String.concat "⊂"
            (transformlist formlist)) ^ ")") in s end
| IMPLIES(form1, form2) →
    if !flag = 1 then begin space := !space ^ "⊂⊂";
        let s = "\n" ^ !space ^ "(IMPLIES" ^
            transstring form1 ^ "⊂" ^ transstring form2 ^ ")" in
        space := String.sub !space 0 (String.length !space - 2); s
    end else begin
        flag := 1; let s = !space ^ "(IMPLIES" ^
            transstring form1 ^ "⊂" ^ transstring form2 ^ ")" in s end
| IFF(form1, form2) →
    if !flag = 1 then begin space := !space ^ "⊂⊂";
        let s = "\n" ^ !space ^ "(IFF" ^
            transstring form1 ^ transstring form2 ^ ")" in
        space := String.sub !space 0 (String.length !space - 2); s end
    else begin flag := 1; let s = !space ^ "(IFF" ^
        transstring form1 ^ transstring form2 ^ ")" in s end
| NOT(form1) →
    if !flag = 1 then begin space := !space ^ "⊂⊂";
        let s = "\n" ^ !space ^ "(NOT" ^ transstring form1 ^ ")" in
        space := String.sub !space 0 (String.length !space - 2); s end
    else begin flag := 1; let s = !space ^
        "(NOT" ^ transstring form1 ^ ")" in s end
| FORALL(y, form1) →
    let shell x ys = match x with
    | VAR(xx) → xx ^ "⊂" ^ ys
    | _ → ys
    in let y2 = List.fold_right shell y "" in
    if !flag = 1 then begin space := !space ^ "⊂⊂";
        let s = "\n" ^ !space ^ "(FORALL(" ^ y2 ^ ")" ^
            transstring form1 ^ ")" in

```

```

    space := String.sub !space 0 (String.length !space - 2); s end
  else begin flag := 1; let s = !space ^
    "(FORALL("^y2^")" ^ transstring form1 ^ ")" in s end
| EXISTS(y, form1) →
  let shell x ys = match x with
  | VAR(xx) → xx ^ "□" ^ ys
  | _ → ys
  in let y2 = List.fold_right shell y "" in
  if !flag = 1 then begin space := !space ^ "□□";
    let s = "\n" ^ !space ^
    "(EXISTS("^y2^")" ^ transstring form1 ^ ")" in
    space := String.sub !space 0 (String.length !space - 2); s end
  else begin flag := 1; let s = !space ^
    "(EXISTS("^y2^")" ^ transstring form1 ^ ")" in s end
| TABLE(table1) → transstring (predtable table1)
| OP(x, form1) → let s = !space ^ transstring form1 in s
| _ → raise
  (Failure "the□parameter□of□transstring□should□be□a□formula")
in transstring predi

```

2. Function *pretty_print* with parameter *predi*

It prints all formulae, including tables, on standard output. Plain formulae are expressed by infix structure with parenthesis to override operator precedence. The ASCII code operators refer to those of B language. Tables are expressed by their original structure with parallel lines composed of '-' character and vertical lines composed of '|' character. Changing lines are controlled in both plain formulae and tables.

(* A record type used in function *pretty_print* and *latex_print*, it is similar to record *table* except that each element of arrays is a string representing the formula which is supposed to send output to terminals or files *)

```
type tstr = {hdm : string array; hdn : string array; bd : string array array}
```

(* A record type used in function *pretty-print* only in order to record the number of lines of each element and the maximum characters of each line. Since a single cell of a table may occupy several lines, if the length of the content in this cell is greater than the default maximum value, we need to memorize the number of lines and characters for each element. We also have to divide the original string element in a cell into a list of strings according to the content for each line. *)

```

type rcst = {hdmtri : (int × int × string list) array;
             hdntri : (int × int × string list) array;
             bdtri : (int × int × string list) array array}

let pretty-print predi =
  let tt = ref (-1) in
  let flag = ref 0 in
  let rec transformlist prec = function
    | [] → []
    | head :: tail →
        transstring prec head :: transformlist prec tail
  and transset = function
    | EMPTY → "{"
    | INSERT(term1, term2) → (match term1 with
        INSERT(t1, t2) → transset term1 ^ "," ^ transstring 0 term2
    | EMPTY → transset term1 ^ transstring 0 term2
    | _ → "")
    | _ → raise
        (Failure "the parameter of transset should be a pair of terms")
  (*A function called to print table in ASCII*)
  and print-table t1 =
    let mh = Array.length t1.headm in
    let nh = Array.length t1.headn in
    let mb = Array.length t1.body in
    let nb = Array.length t1.body.(0) in
    let tst = {hdm = Array.create mh ""; hdn = Array.create nh "";
               bd = Array.make_matrix mb nb ""} in
    for i = 0 to mh - 1 do
      tst.hdm.(i) ← transstring 0 t1.headm.(i)

```

```

done;
for  $j = 0$  to  $nh - 1$  do
   $tst.hdn.(j) \leftarrow transstring\ 0\ t1.headn.(j)$ 
done;
for  $i = 0$  to  $mb - 1$  do
  for  $j = 0$  to  $nb - 1$  do
     $tst.bd.(i).(j) \leftarrow transstring\ 0\ t1.body.(i).(j)$ 
  done
done;

(* Compute the default number of characters for m header and body. *)
let  $compmax = let\ maxt = ref\ 0\ in$ 
  for  $j = 0$  to  $mh - 1$  do
     $maxt := max\ !maxt\ (String.length\ tst.hdm.(j))\ done;$ 
   $!maxt\ in$ 
  let  $bdefault = if\ mh = 0\ then\ (120 - nb - 1)/nb$ 
    else if  $compmax < 18\ then\ (120 - nb - 1 - compmax - 1)/nb$ 
    else  $(102 - nb - 2)/nb\ in$ 
  let  $hdefault = if\ mh = 0\ then\ 0$ 
    else if  $compmax < 18\ then\ compmax\ else\ 18\ in$ 

(* compute function return a record matching the structure of rcs by computing
each element of a tabular formula in terms of string*)
let  $compute\ remain\ default =$ 
  let  $temp = ref\ ""\ in$ 
  let  $left = ref\ []\ in$ 
  let  $col = if\ (String.length\ !remain) > default$ 
    then  $default\ else\ (String.length\ !remain)\ in$ 
  let  $row = ref\ 0\ in$ 
  while  $(String.length\ !remain) > default\ do$ 
     $row := !row + 1;$ 
    let  $s = ref\ default\ in$ 
    while  $!s \geq 0 \wedge String.get\ !remain\ !s \neq ' '\ do$ 
       $s := !s - 1$ 
    done;
     $temp := !remain;$ 

```

```

    remain :=
      if !s ≥ 0 then
        if !s = (String.length !remain) - 1
        then "" else
          String.sub !remain (!s + 1) ((String.length !remain) - (!s) - 1)
        else String.sub !remain default ((String.length !remain)
          - default);
      left := (if !s ≥ 0 then String.sub !temp 0 !s
        else String.sub !temp 0 default) :: !left
    done;
    if String.length !remain = 0 then (!row, col, List.rev !left)
    else (!row + 1, col, List.rev (!remain :: !left)) in
(*This record stores the information of a table. Each element of the table
consists of a triple. The first element of a triple stores the number of lines
of each cell; the second stores the number of characters for each line of each
cell; the third stores the list of strings of each cell, and each string in the list
corresponds to the line to be filled in each cell.*)
    let rcs = {hdmtri = Array.create mh (0,0,[]);
      hdntri = Array.create nh (0,0,[]);
      bdtri = Array.make_matrix mb nb (0,0,[])} in
    for i = 0 to mh - 1 do
      rcs.hdmtri.(i) ← compute (ref tst.hdm.(i)) hdefault
    done;
    for j = 0 to nh - 1 do
      rcs.hdntri.(j) ← compute (ref tst.hdn.(j)) bdefault
    done;
    for i = 0 to mb - 1 do
      for j = 0 to nb - 1 do
        rcs.bdtri.(i).(j) ← compute (ref tst.bd.(i).(j)) bdefault
      done
    done;
done;
(* Compute the number of lines of each row and distance of each column by
computing the maximum number of lines of all elements in the same row and the
maximum distance of all elements in the same column, and put those numbers

```


into two one-dimensional arrays.*)

```

let rowar = Array.create (mb + 1) 0 in
let colar = Array.create (nb + 1) 0 in
let fsttri = function | (a, -, -) → a in
let sndtri = function | (-, b, -) → b in
let thdtri = function | (-, -, c) → c in
let maxt = ref 0 in
for j = 0 to nh - 1 do
  maxt := max !maxt (fsttri rcs.hdntri.(j)) done;
rowar.(0) ← !maxt;
let maxt = ref 0 in
for i = 0 to mh - 1 do
  maxt := max !maxt (sndtri rcs.hdmtri.(i)) done;
colar.(0) ← !maxt;
for i = 0 to mb - 1 do
  let maxt = ref 0 in
  rowar.(i + 1) ← if mh > i then begin
    for j = 0 to nb - 1 do
      maxt := max !maxt (fsttri rcs.bdtri.(i).(j)) done;
      max !maxt (fsttri rcs.hdmtri.(i)) end
    else begin
      for j = 0 to nb - 1 do
        maxt := max !maxt (fsttri rcs.bdtri.(i).(j)) done;
        !maxt end
  done;
  for j = 0 to nb - 1 do
    let maxt = ref 0 in
    colar.(j + 1) ← if nh > j then
      begin for i = 0 to mb - 1 do
        maxt := max !maxt (sndtri rcs.bdtri.(i).(j)) done;
        max !maxt (sndtri rcs.hdntri.(j)) end
      else begin for i = 0 to mb - 1 do
        maxt := max !maxt (sndtri rcs.bdtri.(i).(j)) done;!maxt end
      done;
  done;

```

(*Draw the outline of a table according to the horizontal and vertical distances for each row and column, at the same time record the position of each line of each cell corresponding to a string to fill in contents, and store the contents and positions in a list of pairs in order to replace blank by contents in the next step.*)

```

let s = ref "\n|" in
let p = ref [] in
for j = 0 to nb do
  if j ≠ 0 ∨ mh ≠ 0 then begin
    for k = 0 to colar.(j) - 1 do s := !s ^ "_"; done;
    s := if j = nb then !s else !s ^ "_"; end
done;
s := !s ^ "|\n|";
for i = 0 to mb do
  if i ≠ 0 ∨ nh ≠ 0 then begin
    for rowi = 0 to rowar.(i) - 1 do
      for j = 0 to nb do
        if j ≠ 0 ∨ mh ≠ 0 then begin
          p := if i ≠ 0 ∧ j ≠ 0 then
            if (List.length (thdtri rcs.bdtri.(i - 1).(j - 1))) > rowi
            then (String.length !s,
                  List.nth (thdtri rcs.bdtri.(i - 1).(j - 1)) rowi) :: !p
            else !p
          else if i ≡ 0 ∧ j ≠ 0
            then if rcs.hdntri ≠ [[]] ∧
                  (List.length (thdtri rcs.hdntri.(j - 1))) > rowi
            then (String.length !s,
                  List.nth (thdtri rcs.hdntri.(j - 1)) rowi) :: !p
            else !p
          else if i ≠ 0 ∧ j ≡ 0
            then if rcs.hdmtri ≠ [[]] ∧
                  (List.length (thdtri rcs.hdmtri.(i - 1))) > rowi
            then (String.length !s,
                  List.nth (thdtri rcs.hdmtri.(i - 1)) rowi) :: !p

```

```

        else !p
        else !p;
        for colj = 0 to colar.(j) - 1 do s := !s ^ "␣" done;
        s := !s ^ "|" end
    done;
    s := !s ^ "\n| "
done;
if i ≠ mb then begin
    for j = 0 to nb do
        for colj = 0 to colar.(j) - 1 do s := !s ^ "-" done;
        s := if j = nb ∨ (j = 0 ∧ mh = 0) then !s else !s ^ "+"
        done;
        s := !s ^ "|\n| " end;
    end
done;
for j = 0 to nb do
    if j ≠ 0 ∨ mh ≠ 0 then begin
        for colj = 0 to colar.(j) - 1 do s := !s ^ "-" done;
        s := if j = nb then !s else !s ^ "-"
    end
done;
s := !s ^ "|\n";
(*Fill the contents into cells starting from the recording position*)
for i = 0 to (List.length !p) - 1 do
    for j = 0 to (String.length (snd (List.nth !p i))) - 1 do
        String.set !s ((fst (List.nth !p i)) + j)
            (String.get (snd (List.nth !p i)) j)
    done
done;
(* Set flag before returning to the execution of plain formula*)
flag := 0;
!s

and transstring prec = function
| CONST(x) → let s11 = string_of_int x in

```

```

    if !flag = 0 then tt := !tt + (String.length s11); s11
| VAR(x) → if x = "" then
    begin if !flag = 0 then tt := !tt + 2; "\"\" end
    else begin if !flag = 0 then tt := !tt + (String.length x); x end
| VECVAR(x) → if !flag = 0 then tt := !tt + (String.length x) + 1; x ^= ""
| SUM(term1, term2) → if prec > 9 then begin
    if !flag = 0 then tt := !tt + 5;
    "(" ^ transstring 9 term1 ^ "_" ^ transstring 9 term2 ^ ")"
end else begin
    if !flag = 0 then tt := !tt + 3;
    transstring 9 term1 ^ "_" ^ transstring 9 term2 end
| DIFF(term1, term2) → if prec > 9 then begin
    if !flag = 0 then tt := !tt + 5;
    "(" ^ transstring 9 term1 ^ "-" ^ transstring 9 term2 ^ ")"
end else begin
    if !flag = 0 then tt := !tt + 3;
    transstring 9 term1 ^ "-" ^ transstring 9 term2 end
| DELETE(term1, term2) → if prec > 9 then begin
    if !flag = 0 then tt := !tt + 5;
    "(" ^ transstring 9 term1 ^ "-" ^ transstring 9 term2 ^ ")"
end else begin
    if !flag = 0 then tt := !tt + 3;
    transstring 9 term1 ^ "-" ^ transstring 9 term2 end
| PROD(term1, term2) → if prec > 10 then begin
    if !flag = 0 then tt := !tt + 5;
    "(" ^ transstring 10 term1 ^ "*" ^ transstring 10 term2 ^ ")"
end else begin
    if !flag = 0 then tt := !tt + 3;
    transstring 10 term1 ^ "*" ^ transstring 10 term2 end
| QUOT(term1, term2) → if prec > 10 then begin
    if !flag = 0 then tt := !tt + 5;
    "(" ^ transstring 10 term1 ^ "/" ^ transstring 10 term2 ^ ")"
end else begin
    if !flag = 0 then tt := !tt + 3;

```

```

    transstring 10 term1 ^ "␣/␣" ^ transstring 10 term2 end
| FUN("PAIR",[VAR(s1); VAR(s2)]) →
    if !flag = 0 then tt := !tt + 3 + (String.length s1) + (String.length s2);
    "(" ^ s1 ^ ", " ^ s2 ^ ")"
| FUN("compose",[VAR(s1); VAR(s2)]) → if !flag = 0 then
    tt := !tt + 1 + (String.length s1) + (String.length s2);
    s1 ^ "." ^ s2
| FUN(x, termlist) → if !flag = 0 then begin let ss1 = ref "" in
    flag := 1;
    ss1 := (String.concat "" [x;"(";String.concat "(",
    (transformlist 0 termlist)]) ^ ")";
    tt := !tt + String.length !ss1;
    flag := 0;
    !ss1 end else (String.concat "" [x;"(";String.concat "(",
    (transformlist 0 termlist)]) ^ ")")
| EMPTY → if !flag = 0 then tt := !tt + 2; "{}"
| INSERT(term1, term2) → let ss1 = ref "" in if !flag = 0 then begin
    flag := 1;
    ss1 := transset (INSERT(term1, term2)) ^ "}";
    tt := !tt + String.length !ss1;
    flag := 0;
    !ss1 end else begin ss1 := transset (INSERT(term1, term2)) ^ "}";
    !ss1 end
| UNION(term1, term2) → if prec > 9 then begin
    if !flag = 0 then tt := !tt + 6;
    "(" ^ transstring 9 term1 ^ "␣\\␣" ^ transstring 9 term2 ^ ")"
    end else begin
    if !flag = 0 then tt := !tt + 4; transstring 9 term1 ^ "␣\\␣" ^
    transstring 9 term2 end
| SUBSET(term1, term2) → if prec > 9 then begin
    if !flag = 0 then tt := !tt + 6;
    "(" ^ transstring 9 term1 ^ "␣<:␣" ^
    transstring 9 term2 ^ ")" end else begin
    if !flag = 0 then tt := !tt + 4; transstring 9 term1 ^ "␣<:␣" ^

```

```

    transstring 9 term2 end
| TRUE → if !flag = 0 then tt := !tt + 4; "TRUE"
| FALSE → if !flag = 0 then tt := !tt + 5; "FALSE"
| OP(x, form1) → if !flag = 0 then tt := !tt + (String.length x); x
| EQ(MEMBER(term1, term2), VAR("true")) →
    if prec > 7 then begin if !flag = 0 then tt := !tt + 5;
        "(" ^ transstring 7 term1 ^ "⊂:⊂" ^ transstring 7 term2 ^ ")"
    end else begin if !flag = 0 then tt := !tt + 3;
        transstring 7 term1 ^ "⊂:⊂" ^ transstring 7 term2 end
| EQ(term1, term2) →
    if prec > 7 then begin if !flag = 0 then tt := !tt + 5;
        "(" ^ transstring 7 term1 ^ "⊂=⊂" ^ transstring 7 term2 ^ ")" end
    else begin if !flag = 0 then tt := !tt + 3;
        transstring 7 term1 ^ "⊂=⊂" ^ transstring 7 term2 end
| NEQ(MEMBER(term1, term2), VAR("true")) →
    if prec > 7 then begin if !flag = 0 then tt := !tt + 6;
        "(" ^ transstring 7 term1 ^ "⊂/⊂" ^ transstring 7 term2 ^ ")" end
    else begin if !flag = 0 then tt := !tt + 4;
        transstring 7 term1 ^ "⊂/⊂" ^ transstring 7 term2 end
| NEQ(term1, term2) →
    if prec > 7 then begin if !flag = 0 then tt := !tt + 6;
        "(" ^ transstring 7 term1 ^ "⊂/=⊂" ^ transstring 7 term2 ^ ")" end
    else begin if !flag = 0 then tt := !tt + 4;
        transstring 7 term1 ^ "⊂/=⊂" ^ transstring 7 term2 end
| LT(term1, term2) →
    if prec > 7 then begin if !flag = 0 then tt := !tt + 5;
        "(" ^ transstring 7 term1 ^ "⊂<⊂" ^ transstring 7 term2 ^ ")" end
    else begin if !flag = 0 then tt := !tt + 3;
        transstring 7 term1 ^ "⊂<⊂" ^ transstring 7 term2 end
| LE(term1, term2) →
    if prec > 7 then begin if !flag = 0 then tt := !tt + 6;
        "(" ^ transstring 7 term1 ^ "⊂<=⊂" ^ transstring 7 term2 ^ ")" end
    else begin if !flag = 0 then tt := !tt + 4;
        transstring 7 term1 ^ "⊂<=⊂" ^ transstring 7 term2 end

```

```

| GT(term1, term2) →
  if prec > 7 then begin if !flag = 0 then tt := !tt + 5;
    "(" ^ transstring 7 term1 ^ "□>□" ^ transstring 7 term2 ^ ")" end
  else begin if !flag = 0 then tt := !tt + 3;
    transstring 7 term1 ^ "□>□" ^ transstring 7 term2 end
| GE(term1, term2) →
  if prec > 7 then begin if !flag = 0 then tt := !tt + 6;
    "(" ^ transstring 7 term1 ^ "□>=□" ^ transstring 7 term2 ^ ")" end
  else begin if !flag = 0 then tt := !tt + 4;
    transstring 7 term1 ^ "□>=□" ^ transstring 7 term2 end
| AND(formlist) → if prec > 4 then begin
  if !flag = 0 then begin
    tt := !tt + 1;
    let s11 = ref "(" in
    let temp = ref "" in
    let rec andcon = function
      | [] → s11 := !s11 ^ ")"
      | hd :: tl → tt := !tt + 1; temp := !s11 ^ transstring 4 hd;
        if !tt > 120 then if !s11 ≠ "(" then
          begin tt := 0; s11 := !s11 ^ "\n" ^ transstring 4 hd end
        else begin tt := 1; s11 := "\n" ^ !s11 ^ transstring 4 hd end
        else s11 := !temp;
        andcon []
      | hd :: tl → tt := !tt + 3; temp := !s11 ^ transstring 4 hd;
        if !tt > 123 then if !s11 ≠ "(" then begin tt := 3;
          s11 := !s11 ^ "\n" ^ transstring 4 hd ^ "□&□" end
        else begin tt := 4;
          s11 := "\n" ^ !s11 ^ transstring 4 hd ^ "□&□" end
        else if !tt > 120 then begin tt := 3; s11 := !temp ^ "\n□&□"
          end else s11 := !temp ^ "□&□";
          andcon tl
        in andcon formlist; !s11 end
    else "(" ^ (String.concat "□&□" (transformlist 4 formlist)) ^ ")"
  end else begin

```

```

if !flag = 0 then begin
  let s11 = ref "" in
  let temp = ref "" in
  let rec andcon = function
    | [] → s11 := !s11
    | hd :: [] → temp := !s11 ^ transstring 4 hd;
      if !tt > 120 then begin tt := 0;
        s11 := !s11 ^ "\n" ^ transstring 4 hd end
      else s11 := !temp;
        andcon []
    | hd :: tl → tt := !tt + 3; temp := !s11 ^ transstring 4 hd;
      if !tt > 123 then begin
        tt := 3; s11 := !s11 ^ "\n" ^ transstring 4 hd ^ "␣&␣" end
      else if !tt > 120 then begin tt := 3; s11 := !temp ^ "\n␣&␣" end
        else s11 := !temp ^ "␣&␣";
          andcon tl
      in andcon formlist;
    !s11 end
  else String.concat "␣&␣" (transformlist 4 formlist) end
| OR(formlist) → if prec > 3 then begin
  if !flag = 0 then begin tt := !tt + 1;
    let s11 = ref "(" in
    let temp = ref "" in
    let rec orcon = function
      | [] → s11 := !s11 ^ ")"
      | hd :: [] → tt := !tt + 1; temp := !s11 ^ transstring 3 hd;
        if !tt > 120 then if !s11 ≠ "(" then begin tt := 0;
          s11 := !s11 ^ "\n" ^ transstring 3 hd end
        else begin tt := 1;
          s11 := "\n" ^ !s11 ^ transstring 3 hd end
        else s11 := !temp;
          orcon []
      | hd :: tl → tt := !tt + 4; temp := !s11 ^ transstring 3 hd;
        if !tt > 124 then if !s11 ≠ "(" then begin tt := 4;

```



```

        s11 := !s11 ^ "\n" ^ transstring 3 hd ^ "┐or┐" end
    else begin tt := 5;
        s11 := "\n" ^ !s11 ^ transstring 3 hd ^ "┐or┐" end
    else if !tt > 120 then begin tt := 4;
        s11 := !temp ^ "\n┐or┐" end else s11 := !temp ^ "┐or┐";
    orcon tl
in orcon formlist;
!s11 end else "(" ^ (String.concat "┐or┐"
    (transformlist 3 formlist)) ^ ")"
end else begin
    if !flag = 0 then begin
        let s11 = ref "" in
        let temp = ref "" in
        let rec orcon = function
            | [] → s11 := !s11
            | hd :: [] → temp := !s11 ^ transstring 3 hd;
                if !tt > 120 then begin tt := 0;
                    s11 := !s11 ^ "\n" ^ transstring 3 hd end
                else s11 := !temp;
                orcon []
            | hd :: tl → tt := !tt + 4; temp := !s11 ^ transstring 3 hd;
                if !tt > 124 then begin tt := 4;
                    s11 := !s11 ^ "\n" ^ transstring 3 hd ^ "┐or┐" end
                else if !tt > 120 then begin tt := 4;
                    s11 := !temp ^ "\n┐or┐" end else s11 := !temp ^ "┐or┐";
                orcon tl
        in orcon formlist;
        !s11 end
    else String.concat "┐or┐" (transformlist 3 formlist) end
| IMPLIES(form1, form2) → if prec > 2 then begin
    if !flag = 0 then begin tt := !tt + 1;
        let s11 = ref "(" in
        let temp = ref "" in
        let temp1 = ref "" in

```

```

    temp := !s11 ^ transstring 2 form1;
    if !tt > 120 then begin tt := 1;
        s11 := "\n" ^ !s11 ^ transstring 2 form1 end
    else s11 := !temp;
    tt := !tt + 4;
    if !tt > 120 then begin tt := 4;
        s11 := !s11 ^ "\n" ^ "␣=>␣" end
    else s11 := !s11 ^ "␣=>␣";
    temp1 := !s11 ^ transstring 2 form2;
    tt := !tt + 1;
    if !tt > 120 then begin tt := 1;
        s11 := !s11 ^ "\n" ^ transstring 2 form2 ^ ")" end
    else s11 := !temp1 ^ ")";
    !s11 end
else "(" ^ transstring 2 form1 ^ "␣=>␣" ^
    transstring 2 form2 ^ ")" end
else begin
    if !flag = 0 then begin let s11 = ref "" in
        let temp = ref "" in
        let temp1 = ref "" in
        temp := !s11 ^ transstring 2 form1;
        if !tt > 120 then begin tt := 0;
            s11 := "\n" ^ !s11 ^ transstring 2 form1 end
        else s11 := !temp;
        tt := !tt + 4;
        if !tt > 120 then begin tt := 4; s11 := !s11 ^ "\n" ^ "␣=>␣" end
        else s11 := !s11 ^ "␣=>␣";
        temp1 := !s11 ^ transstring 2 form2;
        if !tt > 120 then begin tt := 1;
            s11 := !s11 ^ "\n" ^ transstring 2 form2 end
        else s11 := !temp1;
        !s11 end
    else transstring 2 form1 ^ "␣=>␣" ^ transstring 2 form2 end
| IFF(form1, form2) → if prec > 2 then begin

```

```

if !flag = 0 then begin tt := !tt + 1;
  let s11 = ref "(" in
  let temp = ref "" in
  let temp1 = ref "" in
  temp := !s11 ^ transstring 2 form1;
  if !tt > 120 then begin tt := 1;
    s11 := "\n" ^ !s11 ^ transstring 2 form1 end
  else s11 := !temp;
  tt := !tt + 5;
  if !tt > 120 then begin tt := 5; s11 := !s11 ^ "\n_<=>_" end
  else s11 := !s11 ^ "_<=>_";
  temp1 := !s11 ^ transstring 2 form2;
  tt := !tt + 1;
  if !tt > 120 then begin tt := 1;
    s11 := !s11 ^ "\n" ^ transstring 2 form2 ^ ")" end
  else s11 := !temp1 ^ ")"; !s11 end
else "(" ^ transstring 2 form1 ^ "_<=>_" ^ transstring 2 form2 ^ ")"
end else begin
  if !flag = 0 then begin let s11 = ref "" in
    let temp = ref "" in
    let temp1 = ref "" in
    temp := !s11 ^ transstring 2 form1;
    if !tt > 120 then begin tt := 0;
      s11 := "\n" ^ !s11 ^ transstring 2 form1 end
    else s11 := !temp;
    tt := !tt + 5;
    if !tt > 120 then begin tt := 5; s11 := !s11 ^ "\n_<=>_" end
    else s11 := !s11 ^ "_<=>_";
    temp1 := !s11 ^ transstring 2 form2;
    if !tt > 120 then begin tt := 0;
      s11 := !s11 ^ "\n" ^ transstring 2 form2 end
    else s11 := !temp1;
    !s11 end
  else transstring 2 form1 ^ "_<=>_" ^ transstring 2 form2 end

```

```

| NOT(form1) → if prec > 5 then begin
  if !flag = 0 then begin tt := !tt + 6;
    let s11 = ref "" in
    let temp = ref "" in
    temp := "(not_␣" ^ transstring 5 form1 ^ ")";
    if !tt > 120 then begin tt := 6;
      s11 := "\n" ^ "(not_␣" ^ transstring 5 form1 ^ ")" end
    else s11 := !temp;
    !s11 end
  else "(not_␣" ^ transstring 5 form1 ^ ")" end
else begin
  if !flag = 0 then begin tt := !tt + 4;
    let s11 = ref "" in
    let temp = ref "" in
    temp := "not_␣" ^ transstring 5 form1;
    if !tt > 120 then begin tt := 4;
      s11 := "\n" ^ "not_␣" ^ transstring 5 form1 end
    else s11 := !temp;
    !s11 end
  else "not_␣" ^ transstring 5 form1 end
| FORALL(y, form1) →
(* Shell VAR() of expression VAR(m), leaving m to be concatenated into a
string of variables*)
  let shell x ys = match x with
    | VAR(xx) → xx ^ ", " ^ ys
    | _ → ", "
  in let y1 = List.fold_right shell y ""
  in let y2 = if y1 = "" then ""
    else String.sub y1 0 ((String.length y1) - 1) in
  if !flag = 0 then begin tt := !tt + 5 + (String.length y2);
    let s11 = ref "" in
    let temp = ref "" in
    if !tt > 120 then begin tt := 5 + (String.length y2);
      s11 := "\n" ^ "(" ^ y2 ^ ".␣" end

```

```

    else s11 := "(" ^ y2 ^ ".\u";
    temp := !s11 ^ transstring 1 form1 ^ ")";
    if !tt > 120 then begin tt := 0;
        s11 := !s11 ^ "\n" ^ transstring 1 form1 ^ ")" end
    else s11 := !temp; !s11 end
    else "(" ^ y2 ^ ".\u" ^ transstring 1 form1 ^ ")"
| EXISTS(y, form1) →
    let shell x ys = match x with
    | VAR(xx) → xx ^ ", " ^ ys
    | _ → ", "
    in let y1 = List.fold_right shell y ""
    in let y2 = if y1 = "" then ""
        else String.sub y1 0 ((String.length y1) - 1) in
    if !flag = 0 then begin tt := !tt + 5 + (String.length y2);
    let s11 = ref "" in
    let temp = ref "" in
    if !tt > 120 then begin tt := 5 + (String.length y2);
        s11 := "\n" ^ "(" ^ y2 ^ ".\u" end
    else s11 := "\n" ^ "#(" ^ y2 ^ ".\u";
    temp := !s11 ^ transstring 1 form1 ^ ")";
    if !tt > 120 then begin tt := 0; s11 := "\n" ^
        transstring 1 form1 ^ ")" end
    else s11 := !temp;
    !s11 end else "(" ^ y2 ^ ".\u" ^ transstring 1 form1 ^ ")"
(*A table included in a formula should be printed as a new line*)
| TABLE(table1) → tt := 0; flag := 1; print_table table1
| _ → raise (Failure
    "the_\u parameter_\u of_\u transstring_\u should_\u be_\u a_\u formula")
in transstring 0 predi

```

3. Function *latex_print* with parameter *predi*

It prints all formulae, including tables, on \LaTeX files. The formulae have the same structure as those of 2 except that all characters and tables are in \LaTeX

typesetting. Plain predicates are broken into individual lines of the appropriate size by \LaTeX typesetting. In order to control changing lines in tables, we set the alphabetic characters to the typewriter font where each glyph has the same width as all others.

```

let latex_print predi =
  let flag = ref 0 in
  (* Replace symbols at last in order to count the number of characters the latex
  output.*)
  let rep st =
    let tmp = ref "" in
    let i = ref 0 in
    while !i < (String.length st) do
      (match st.[i] with
        | '!' → tmp := !tmp ^ "$\\forall$"
        | '@' → tmp := !tmp ^ "$\\in$"
        | '#' → tmp := !tmp ^ "$\\exists$"
        | '$' → tmp := !tmp ^ "$\\notin$"
        | '%' → tmp := !tmp ^ "$\\neq$"
        | '<' → if !i + 1 < (String.length st) ∧ st.[i + 1] = '>' then
            begin i := !i + 1; tmp := !tmp ^ "$\\Leftrightarrow$" end
          else tmp := !tmp ^ "$< $"
        | '^' → tmp := !tmp ^ "$\\leq$"
        | '>' → tmp := !tmp ^ "$> $"
        | '?' → tmp := !tmp ^ "$\\geq$"
        | '&' → tmp := !tmp ^ "$\\wedge$"
        | '-' → if !i + 1 < (String.length st) ∧ st.[i + 1] = '>' then
            begin i := !i + 1; tmp := !tmp ^ "$\\Longrightarrow$" end
          else tmp := !tmp ^ "$\\neg$"
        | '~' → tmp := !tmp ^ "$\\neg$"
        | '*' → tmp := !tmp ^ "$\\times$"
        | '/' → tmp := !tmp ^ "$\\div$"
        | '{' → tmp := !tmp ^ "\\{"
        | '}' → tmp := !tmp ^ "\\}"
        | '|' → tmp := !tmp ^ "$\\cup$"

```

```

| '\001' → tmp := !tmp ^ "$\\subsepeq$"
| '\002' → tmp := !tmp ^ "$\\circ$"
| '\003' → tmp := !tmp ^ "$\\vee$"
| c → if (c ≥ '0' ∧ c ≤ '9') ∨ (c ≥ 'A' ∧ c ≤ 'Z') ∨
      (c ≥ 'a' ∧ c ≤ 'z') ∨ c = '.'
      then tmp := !tmp ^ "\\texttt{" ^ (Char.escaped c) ^ "}"
      else tmp := !tmp ^ (Char.escaped c);
  i := !i + 1
done;
!tmp in

let rec transformlist prec = function
| [] → []
| head :: tail → transstring prec head :: transformlist prec tail

and transset = function
| EMPTY → if !flag = 0 then "\\{" else "{"
| INSERT(term1, term2) →
  (match term1 with
   | INSERT(t1, t2) → transset term1 ^ "," ^ transstring 0 term2
   | EMPTY → transset term1 ^ transstring 0 term2
   | _ → "")
| _ → raise
  (Failure "the parameter of transset should be a pair of terms")

(* Print table on a latex file, the difference from function pretty_print is that
horizontal and vertical distances for each cell is not computed, the only separate
symbol for a column is a & and for a row is a \\hline *)

and print_table t1 =
  let mh = Array.length t1.headm in
  let nh = Array.length t1.headn in
  let mb = Array.length t1.body in
  let nb = Array.length t1.body.(0) in
  let tst = {hdm = Array.create mh ""; hdn = Array.create nh "";
    bd = Array.make_matrix mb nb ""} in
  for i = 0 to mh - 1 do

```

```

    tst.hdm.(i) ← transstring 0 t1.headm.(i)
done;
for j = 0 to nh – 1 do
    tst.hdn.(j) ← transstring 0 t1.headn.(j)
done;
for i = 0 to mb – 1 do
    for j = 0 to nb – 1 do
        tst.bd.(i).(j) ← transstring 0 t1.body.(i).(j)
    done
done;

(*Compute the default number of characters for m header and body. *)
let compmax = let maxt = ref 0 in
for j = 0 to mh – 1 do
    maxt := max !maxt (String.length tst.hdm.(j)) done; !maxt in
let bdefault = if mh = 0 then (75 – 5 × nb / 2 – 3) / nb else if compmax < 15
    then (75 – 5 × nb / 2 – 5 – compmax) / nb else (60 – 5 × nb / 2 – 5) / nb in
let hdefault = if mh = 0 then 0 else if compmax < 15
    then compmax else 15 in

(* compute function return a record matching the structure of rcs by computing
each element of a tabular formula in terms of string*)
let compute remain default =
    let temp = ref "" in
    let left = ref [] in
    let col = if (String.length !remain) > default
        then default else (String.length !remain) in
    let row = ref 0 in
    while (String.length !remain) > default do
        row := !row + 1;
        let s = ref default in
        while !s ≥ 0 ∧ String.get !remain !s ≠ ' ' do
            s := !s – 1
        done;
        temp := !remain;
        remain := if !s ≥ 0 then if !s = (String.length !remain) – 1 then ""

```



```

        else String.sub !remain (!s + 1)
            ((String.length !remain) – (!s) – 1)
        else String.sub !remain default
            ((String.length !remain) – default);
    left := (if !s ≥ 0 then String.sub !temp 0 !s
        else String.sub !temp 0 default) :: !left
done;
if String.length !remain = 0 then (!row, col, List.rev !left)
    else (!row + 1, col, List.rev (!remain :: !left)) in

```

(* This record stores the information of a table, and each element of the table consists of a triple. The first element of a triple stores the number of lines of each cell; the second stores the number of characters for each line of each cell; the third stores the list of strings of each cell, and each string in the list corresponds to the line to be filled in each cell. *)

```

let rcs = {hdmtri = Array.create mh (0,0,[]);
    hdntri = Array.create nh (0,0,[]);
    bdtri = Array.make_matrix mb nb (0,0,[])} in
for i = 0 to mh – 1 do
    rcs.hdmtri.(i) ← compute (ref tst.hdm.(i)) hdefault
done;
for j = 0 to nh – 1 do
    rcs.hdntri.(j) ← compute (ref tst.hdn.(j)) bdefault
done;
for i = 0 to mb – 1 do
    for j = 0 to nb – 1 do
        rcs.bdtri.(i).(j) ← compute (ref tst.bd.(i).(j)) bdefault
    done
done;

```

(* Compute the number of lines of each row and distance of each column by computing the maximum number of lines of all elements in the same row and the maximum distance of all elements in the same column, and put those numbers into two one-dimensional arrays. *)

```

let rowar = Array.create (mb + 1) 0 in

```

```

let colar = Array.create (nb + 1) 0 in
let fsttri = function | (a, -, -) → a in
let sndtri = function | (-, b, -) → b in
let thdtri = function | (-, -, c) → c in
let maxt = ref 0 in
for j = 0 to nh - 1 do maxt := max !maxt (fsttri rcs.hdntri.(j)) done;
rowar.(0) ← !maxt;
let maxt = ref 0 in
for i = 0 to mh - 1 do maxt := max !maxt (sndtri rcs.hdmtri.(i)) done;
colar.(0) ← !maxt;
for i = 0 to mb - 1 do
  let maxt = ref 0 in
  rowar.(i + 1) ← if mh > i then begin
    for j = 0 to nb - 1 do
      maxt := max !maxt (fsttri rcs.bdtri.(i).(j)) done;
      max !maxt (sndtri rcs.hdmtri.(i)) end
    else begin for j = 0 to nb - 1 do
      maxt := max !maxt (fsttri rcs.bdtri.(i).(j)) done;
      !maxt end
  done;
  for j = 0 to nb - 1 do
    let maxt = ref 0 in
    colar.(j + 1) ← if nh > j then
      begin for i = 0 to mb - 1 do
        maxt := max !maxt (sndtri rcs.bdtri.(i).(j)) done;
        max !maxt (sndtri rcs.hdntri.(j)) end
      else begin for i = 0 to mb - 1 do
        maxt := max !maxt (sndtri rcs.bdtri.(i).(j)) done; !maxt end
    done;
  done;
(* Insert some material in the table preamble *)
let str = ref "\\begin{longtable}{| " in
for i = 0 to nb do
  if i ≠ 0 ∨ mh ≠ 0 then str := !str ^ "c|"
done;

```

```
str := !str ^ "}\n\\hline\n";
```

(* Upper header of a table *)

```
if nh ≠ 0 then begin
  for rowi = 0 to rowar.(0) - 1 do
    if mh ≠ 0 then str := !str ^ "␣&␣";
    for j = 0 to nb - 2 do
      str := !str ^ (if List.length (thdtri rcs.hdntri.(j)) > rowi
        then rep (List.nth (thdtri rcs.hdntri.(j)) rowi)
        else "␣") ^ "␣&␣"
    done;
    str := !str ^ (if List.length (thdtri rcs.hdntri.(nb - 1)) > rowi
      then rep (List.nth (thdtri rcs.hdntri.(nb - 1)) rowi)
      else "␣") ^ "\\\\"
    done;
    str := !str ^ "\n\\hline\n"
  end;
```

(* Left header and body of a table *)

```
for i = 0 to mb - 1 do
  for rowi = 0 to rowar.(i + 1) - 1 do
    str := !str ^ (if mh = 0 then "␣"
      else if List.length (thdtri rcs.hdmtri.(i)) > rowi then
        (rep (List.nth (thdtri rcs.hdmtri.(i)) rowi)) ^ "␣&␣"
      else "␣&␣");
    for j = 0 to nb - 2 do
      str := !str ^ (if List.length (thdtri rcs.bdtri.(i).(j)) > rowi
        then rep (List.nth (thdtri rcs.bdtri.(i).(j)) rowi)
        else "␣") ^ "␣&␣"
    done;
    str := !str ^ (if List.length (thdtri rcs.bdtri.(i).(nb - 1)) > rowi
      then rep (List.nth (thdtri rcs.bdtri.(i).(nb - 1)) rowi)
      else "␣") ^ "\\\\";
    done;
    str := !str ^ "\n\\hline\n"
```

```

done;

(* End of a table, set flag before returning to the execution of plain formula.
*)

    str := !str ^ "\\end{longtable}\\n";
    flag := 0;
    !str

and transstring prec = function
| CONST(x) → string_of_int x
| VAR(x) → if x = "" then "\\\"" else x
| VECVAR(x) → x ^ "="
| SUM(term1, term2) → if prec > 9 then
    "transstring 9 term1 ^ \"_+\" ^ transstring 9 term2 ^\""
    else transstring 9 term1 ^ \"_+\" ^ transstring 9 term2
| DIFF(term1, term2) → if prec > 9 then
    "transstring 9 term1 ^ \"_−\" ^ transstring 9 term2 ^\""
    else transstring 9 term1 ^ \"_−\" ^ transstring 9 term2
| DELETE(term1, term2) → if prec > 9 then
    "transstring 9 term1 ^ \"_−\" ^ transstring 9 term2 ^\""
    else transstring 9 term1 ^ \"_−\" ^ transstring 9 term2
| PROD(term1, term2) → if !flag = 0 then if prec > 10 then
    "transstring 10 term1 ^ \"_\\times_\" ^ transstring 10 term2 ^\""
    else transstring 10 term1 ^ \"_\\times_\" ^ transstring 10 term2
    else if prec > 10 then
    "transstring 10 term1 ^ \"_*_\" ^ transstring 10 term2 ^\""
    else transstring 10 term1 ^ \"_*_\" ^ transstring 10 term2
| QUOT(term1, term2) → if !flag = 0 then if prec > 10 then
    "transstring 10 term1 ^ \"_\\div_\" ^ transstring 10 term2 ^\""
    else transstring 10 term1 ^ \"_\\div_\" ^ transstring 10 term2
    else if prec > 10 then
    "transstring 10 term1 ^ \"_/_\" ^ transstring 10 term2 ^\""
    else transstring 10 term1 ^ \"_/_\" ^ transstring 10 term2
| FUN("PAIR",[VAR(s1); VAR(s2)]) → ("s1 ^ \"_\" , s2 ^ \"_\" )"
| FUN("compose",[VAR(s1); VAR(s2)]) →

```

```

    if !flag = 0 then s1 ^ "\circ_" ^ s2 else s1 ^ "\002_" ^ s2
| FUN(x, termlist) → (String.concat ""
    [x;"(";String.concat ", " (transformlist 0 termlist)]) ^ ")")
| EMPTY → if !flag = 0 then "\\{\}" else "{}"
| INSERT(term1, term2) → if !flag = 0 then
    transset (INSERT(term1, term2)) ^ "\\}"
    else transset (INSERT(term1, term2)) ^ "}"
| UNION(term1, term2) → if !flag = 0 then if prec > 9 then
    "(" ^ transstring 9 term1 ^ "\cup_" ^ transstring 9 term2 ^ ")"
    else transstring 9 term1 ^ "\cup_" ^ transstring 9 term2
    else if prec > 9 then
    "(" ^ transstring 9 term1 ^ "|_" ^ transstring 9 term2 ^ ")"
    else transstring 9 term1 ^ "|_" ^ transstring 9 term2
| SUBSET(term1, term2) → if !flag = 0 then if prec > 9 then
    "(" ^ transstring 9 term1 ^ "\subseteq_" ^
    transstring 9 term2 ^ ")"
    else transstring 9 term1 ^ "\subseteq_" ^ transstring 9 term2
    else if prec > 9 then
    "(" ^ transstring 9 term1 ^ "\001_" ^
    transstring 9 term2 ^ ")"
    else transstring 9 term1 ^ "\001_" ^ transstring 9 term2
| TRUE → "TRUE"
| FALSE → "FALSE"
| OP(x, form1) → x
| EQ(MEMBER(term1, term2), VAR("true")) →
    if !flag = 0 then if prec > 7 then
    "(" ^ transstring 7 term1 ^ "\in_" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "\in_" ^ transstring 7 term2
    else if prec > 7 then
    "(" ^ transstring 7 term1 ^ "@_" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "@_" ^ transstring 7 term2
| EQ(term1, term2) → if prec > 7 then
    "(" ^ transstring 7 term1 ^ "=_" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "=_" ^ transstring 7 term2

```

```

| NEQ(MEMBER(term1, term2), VAR("true")) → if !flag = 0 then
    if prec > 7 then
        "(" ^ transstring 7 term1 ^ "⊄\notin" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊄\notin" ^ transstring 7 term2
    else if prec > 7 then
        "(" ^ transstring 7 term1 ^ "⊄$" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊄$" ^ transstring 7 term2
| NEQ(term1, term2) → if !flag = 0 then if prec > 7 then
    "(" ^ transstring 7 term1 ^ "⊄\neq" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊄\neq" ^ transstring 7 term2
    else if prec > 7 then
        "(" ^ transstring 7 term1 ^ "⊄%" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊄%" ^ transstring 7 term2
| LT(term1, term2) → if !flag = 0 then if prec > 7 then
    "(" ^ transstring 7 term1 ^ "⊂<" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊂<" ^ transstring 7 term2
    else if prec > 7 then
        "(" ^ transstring 7 term1 ^ "⊂<" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊂<" ^ transstring 7 term2
| LE(term1, term2) → if !flag = 0 then if prec > 7 then
    "(" ^ transstring 7 term1 ^ "⊂\leq" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊂\leq" ^ transstring 7 term2
    else if prec > 7 then
        "(" ^ transstring 7 term1 ^ "⊂⊆" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊂⊆" ^ transstring 7 term2
| GT(term1, term2) → if !flag = 0 then if prec > 7 then
    "(" ^ transstring 7 term1 ^ "⊃>" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊃>" ^ transstring 7 term2
    else if prec > 7 then "(" ^ transstring 7 term1 ^ "⊃⊇" ^
        transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊃⊇" ^ transstring 7 term2
| GE(term1, term2) → if !flag = 0 then if prec > 7 then
    "(" ^ transstring 7 term1 ^ "⊂\geq" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "⊂\geq" ^ transstring 7 term2

```

```

else if prec > 7 then
    "(" ^ transstring 7 term1 ^ "∪?" ^ transstring 7 term2 ^ ")"
    else transstring 7 term1 ^ "∪?" ^ transstring 7 term2
| AND(formlist) → if !flag = 0 then if prec > 4 then
    "(" ^ (String.concat "∪\wedge" (transformlist 4 formlist)) ^ ")"
    else String.concat "∪\wedge" (transformlist 4 formlist)
else if prec > 4 then
    "(" ^ (String.concat "∪&" (transformlist 4 formlist)) ^ ")"
    else String.concat "∪&" (transformlist 4 formlist)
| OR(formlist) → if !flag = 0 then if prec > 3 then
    "(" ^ (String.concat "∪\vee" (transformlist 3 formlist)) ^ ")"
    else String.concat "∪\vee" (transformlist 3 formlist)
else if prec > 3 then
    "(" ^ (String.concat "∪\003" (transformlist 3 formlist)) ^ ")"
    else String.concat "∪\003" (transformlist 3 formlist)
| IMPLIES(form1, form2) →
    if !flag = 1 then begin space := !space ^ "∪";
        let s = "\n" ^ !space ^ "(IMPLIES" ^
            transstring form1 ^ "∪" ^ transstring form2 ^ ")" in
        space := String.sub !space 0 (String.length !space - 2); s
    end else begin
        flag := 1; let s = !space ^ "(IMPLIES" ^
            transstring form1 ^ "∪" ^ transstring form2 ^ ")" in s end
| IFF(form1, form2) →
    if !flag = 1 then begin space := !space ^ "∪";
        let s = "\n" ^ !space ^ "(IFF" ^
            transstring form1 ^ transstring form2 ^ ")" in
        space := String.sub !space 0 (String.length !space - 2); s end
    else begin flag := 1; let s = !space ^ "(IFF" ^
        transstring form1 ^ transstring form2 ^ ")" in s end
| NOT(form1) → if !flag = 0 then
    if prec > 5 then "(" ^ "\neg" ^ transstring 5 form1 ^ ")"
    else "\neg" ^ transstring 5 form1
else

```

```

    if prec > 5 then "~" ^ transstring 5 form1 ^ ")"
    else "~" ^ transstring 5 form1
| forall(y, form1) →
(* Shell VAR() of expression VAR(m), leaving m to be concatenated into a
string of variables*)
    let shell x ys = match x with
    | VAR(xx) → xx ^ ", " ^ ys
    | _ → ", "
    in let y1 = List.fold_right shell y ""
    in let y2 = if y1 = "" then "" else
        String.sub y1 0 ((String.length y1) - 1) in
    if !flag = 0 then
        "(\\forall_" ^ y2 ^ "._" ^ transstring 1 form1 ^ ")"
    else "(!_)" ^ y2 ^ "._" ^ transstring 1 form1 ^ ")"
| exists(y, form1) →
    let shell x ys = match x with
    | VAR(xx) → xx ^ ", " ^ ys
    | _ → ", "
    in let y1 = List.fold_right shell y ""
    in let y2 = if y1 = "" then "" else
        String.sub y1 0 ((String.length y1) - 1) in
    if !flag = 0 then
        "(\\exists_" ^ y2 ^ "._" ^ transstring 1 form1 ^ ")"
    else "(#_" ^ y2 ^ "._" ^ transstring 1 form1 ^ ")"
| TABLE(table1) → flag := 1; "$" ^ print_table table1 ^ "$_"
| _ → raise
(Failure "the_parameter_of_transstring_should_be_a_formula1") in
(* Before calling transstring function, insert some material in the latex preamble*)
"\\documentclass{article}\\n\\usepackage{longtable}\\n"
^"\\begin{document}\\n$_"^(transstring 0 predi)^"$\\n"^^"\\end{document}"

```


4.5.4 Theorem Proving

It is the main part of our program which contains the following functions:

1. Function *valid pred* checks if the given predicate is valid or not by invoking Simplify to prove it.

```
let valid pred =
  try
    (* Open the named file for writting *)
    let outchannel =
      open_out_gen [Open_creat; Open_trunc; Open_wronly] 6448 "outfile"
    and outfile = prints pred "" in
    (* Write the string on the given output channel *)
    output_string outchannel (outfile);
    (* Flush the buffer associated with the given output channel, performing all
    pending writes on that channel *)
    flush outchannel;
    (* Close the given channel, flushing all buffered write operations *)
    close_out outchannel;
    (* Unix system call, the -nosc options causes Simplify to simply output valid
    or invalid, if the argument -ax file is given, Simplify looks for file.ax. Set the
    AXIOMDIR environment variable in .bash_profile, Simplify looks for that file
    in the given directory. *)
    if Unix.system ("Simplify_ax_myaxiom_nosc_outfile>_abc")
      = WEXITED 127
    then exit 0;
    (* Open the named file for reading *)
    let inchannel = open_in_gen [Open_rdonly] 6448 "abc"
    (* Read characters from the given input channel, until a newline character is
    encountered *)
```

in let *infile* = *input_line inchannel*

(*The output *invalid* contains a character 'n' which is not in output *valid*, this function is used to check if the given Simplify formula is valid or invalid *)

in \neg (*String.contains infile 'n'*)

(* If encountering the end of file, terminate program normally *)

with *End_of_file* \rightarrow *exit* 0

2. Some auxiliary functions

Since a variable x in its final state is expressed by $x1$, the functions below are used to replace x by $x1$ in its context:

(a) Function *replace* ($u1, u2$) t

It replaces each occurrence of $u1$ in predicate t by $u2$. Specially, if $VAR(x)$ occurs in t and $u1 = VECVAR(x)$, it is replaced by $u2$. This function could be used to replace a variable in the left header of a vector table by its expressions in body matrix.

let rec *replace* ($u1, u2$) t =

match t with

$VAR(x) \rightarrow$ if $VAR(x) = u1 \vee VECVAR(x) = u1$ then $u2$ else $VAR(x)$

| $VECVAR(x) \rightarrow$ if $VECVAR(x) = u1$ then $u2$ else $VECVAR(x)$

| $CONST(x) \rightarrow CONST(x)$

| $DELETE(term1, term2) \rightarrow$

$DELETE(replace(u1, u2) term1, replace(u1, u2) term2)$

| $SUM(term1, term2) \rightarrow$

$SUM(replace(u1, u2) term1, replace(u1, u2) term2)$

| $DIFF(term1, term2) \rightarrow$

$DIFF(replace(u1, u2) term1, replace(u1, u2) term2)$

| $PROD(term1, term2) \rightarrow$

$PROD(replace(u1, u2) term1, replace(u1, u2) term2)$

| $QUOT(term1, term2) \rightarrow$

$$\begin{aligned}
 & \text{QUOT}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{FUN}(x, \text{termlist}) \rightarrow \\
 & \quad \text{FUN}(x, \text{List.map } (\text{replace } (u1, u2)) \text{ termlist}) \\
 | & \text{EMPTY} \rightarrow \text{EMPTY} \\
 | & \text{INSERT}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{INSERT}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{MEMBER}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{MEMBER}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{UNION}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{UNION}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{SUBSET}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{SUBSET}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{TRUE} \rightarrow \text{TRUE} \\
 | & \text{FALSE} \rightarrow \text{FALSE} \\
 | & \text{EQ}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{EQ}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{NEQ}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{NEQ}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{LT}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{LT}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{LE}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{LE}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{GT}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{GT}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{GE}(\text{term1}, \text{term2}) \rightarrow \\
 & \quad \text{GE}(\text{replace } (u1, u2) \text{ term1}, \text{replace } (u1, u2) \text{ term2}) \\
 | & \text{OP}(x, \text{form1}) \rightarrow \text{OP}(x, \text{replace } (u1, u2) \text{ form1}) \\
 | & \text{AND}(\text{formlist}) \rightarrow \text{AND}(\text{List.map } (\text{replace } (u1, u2)) \text{ formlist}) \\
 | & \text{OR}(\text{formlist}) \rightarrow \text{OR}(\text{List.map } (\text{replace } (u1, u2)) \text{ formlist}) \\
 | & \text{IMPLIES}(\text{form1}, \text{form2}) \rightarrow \\
 & \quad \text{IMPLIES}(\text{replace } (u1, u2) \text{ form1}, \text{replace } (u1, u2) \text{ form2}) \\
 | & \text{IFF}(\text{form1}, \text{form2}) \rightarrow \\
 & \quad \text{IFF}(\text{replace } (u1, u2) \text{ form1}, \text{replace } (u1, u2) \text{ form2}) \\
 | & \text{NOT}(\text{form1}) \rightarrow \text{NOT}(\text{replace } (u1, u2) \text{ form1})
 \end{aligned}$$

```

| FORALL(x, form1) →
  let rec replaces (u1, u2) s =
    match s with
    | [] → []
    | hd :: tail → replace (u1, u2) hd :: replaces (u1, u2) tail
    in FORALL(replaces (u1, u2) x, replace (u1, u2) form1)
| EXISTS(x, form1) →
  let rec replaces (u1, u2) s =
    match s with
    | [] → []
    | hd :: tail → replace (u1, u2) hd :: replaces (u1, u2) tail
    in EXISTS(replaces (u1, u2) x, replace (u1, u2) form1)
| TABLE(table1) → replace (u1, u2) (predtable table1)

```

(b) Function *reformlist pred varl*

It concatenates each variable in the variable list *varl* with the character '1' and generates a new predicate from *pred* by replacing each occurrence of the variable which belongs to *varl* with the concatenated one.

```

let rec reformlist pred varl =
  match varl with
  | [] → pred
  | VAR(hd) :: tail → let pred1 = (replace (VAR(hd), VAR(hd ^ "1")) pred)
    in reformlist pred1 tail
  | VECVAR(hd) :: tail → let pred1 = (replace (VECVAR(hd),
    VECVAR(hd ^ "1")) pred) in reformlist pred1 tail
  | _ → pred

```

(c) Function *replacelist predlist*

It concatenates each variable in the variable list *predlist* with the character '1' and generates a new variable list from *predlist* by replacing each variable with the concatenated one.

```

let rec replacelist predlist =
  match predlist with
  | [] → []
  | VAR(hd) :: tail → VAR(hd ^ "1") :: replacelist tail
  | _ → []

```

3. Validation functions of side conditions

Some theorems require tables to have particular properties, we represent those through functions that return if a table has these properties or not. All vectors below are represented by one dimensional arrays and an Ocaml code segment is followed by each interpretation.

- (a) Function *disjoint pv* checks if vector *pv* is disjoint.

```

let disjoint pv =
  let len = Array.length pv in
  let reference = ref true in
  for i = 0 to len - 1 do
    for j = 0 to len - 1 do
      if (i ≠ j) ∧ (valid (AND([pv.(i); pv.(j)]))) then
        reference := false
    done
  done;
  !reference

```

- (b) Function *jointdisjoint (pv, qv)*

It checks if *pv* and *qv* are jointly disjoint.

```

let jointdisjoint (pv, qv) =
  let lenpv = Array.length pv in
  let lenqv = Array.length qv in
  let reference = ref true in
  if lenpv = 0 then reference := disjoint qv;
  if lenqv = 0 then reference := disjoint pv;
  for i = 0 to lenpv - 1 do
    for j = 0 to lenqv - 1 do
      for k = 0 to lenpv - 1 do
        for l = 0 to lenqv - 1 do
          if ((i ≠ k) ∨ (j ≠ l)) ∧
            (valid (AND([pv.(i); qv.(j); pv.(k); qv.(l)]))) then
            reference := false
        done
      done
    done
  done;
  !reference

```

(c) Function *coversatleast* *pv* *c*

It checks if vector *pv* covers at least *c*.

```

let coversatleast pv c =
  let len = Array.length pv in
  let reference = ref [FALSE] in
  for i = 0 to len - 1 do
    reference := pv.(i) :: !reference
  done;
  len = 0 ∨ valid (IMPLIES(c, OR(!reference)))

```

(d) Function *covers* *pv* *c*

It checks if vector *pv* covers exactly *c*.

```

let covers pv c =
  let len = Array.length pv in
  let reference = ref [FALSE] in
  for i = 0 to len - 1 do
    reference := pv.(i) :: !reference
  done;
  len = 0  $\vee$  valid (IFF(c, OR(!reference)))

```

(e) Function *total pv*

It checks if vector *pv* is total.

```

let total pv =
  covers pv TRUE

```

(f) Function *partition pv c*

It checks if vector *pv* partitions *c*.

```

let partition pv c =
  disjoint pv  $\wedge$  covers pv c

```

(g) Function *injectiver r vls*

Since a relation between elements of types *VL* and *VLR* is a function of type $VL \rightarrow VLR \rightarrow Bool$, lists *vl* and *vlr* are used to represent types *VL* and *VLR* of relation *r* respectively. It checks if relation *r* between elements of types *VL* and *VLR* is injective.

```

let injectiver  $r$  ( $vl, vlr$ ) =
  let  $r1$  = reformlist  $r$   $vl$ 
  in let rec identity varlist =
    match varlist with
    | [] → TRUE
    | VAR( $hd$ ) :: tail → AND([EQ(VAR( $hd$  ^ "1"), VAR( $hd$ )),
                             identity tail])
    | _ → FALSE
  in valid (IMPLIES(EXISTS( $vlr$ , AND([ $r$ ;  $r1$ ])), identity  $vl$ ))

```

(h) Function *totalr r vls*

It checks if relation r between elements of types VL and VLA is total.

```

let totalr  $r$   $vls$  = match  $vls$  with
  ( $vl, vla$ ) → valid (EXISTS( $vla$ ,  $r$ ))

```

4. Implementation of theorems

This part is introduced in Chapter 5.

5. Case study and examples

This part is illustrated in Chapter 6, 7, and 8 separately.

Chapter 5

Implementation of Theorems

5.1 Principle of Proof

We divide our theorem implementations into four kinds.

1. Transforming tables that turn out to be useful intermediate steps when combining larger tables.

Each goal is a series of logical and boolean operations on elements of two tables and comes directly from applying a theorem.

2. Operations on tabular predicates.

Each goal is a logical operation between a tabular predicate and a plain predicate or two tabular predicates.

3. Precondition or weakest precondition.

In rare cases, we will use the result directly from applying a theorem as our goal.

A simpler formula which is equivalent to the result is preferred. Therefore, two kinds of parameters are included in each function, one specifies the information to be conveyed on the conditional part of a theorem and another one is our goal

formula. A result is first constructed from the information provided and the theorems applied. This result is then compared with our goal formula by \Leftrightarrow operator, that is, a new formula is constructed. If Simplify can prove our new formula, our goal is valid.

4. Tabular verification and refinement.

Each theorem is considered as an alternative statement. It is implemented by taking variables in *if* part as parameters.

A function definition in OCaml which implement a theorem consists of a self-explaining function name and some parameters in their own formats. An error will be raised when a calling function pass parameters which do not match the requiring format. If a theorem has side conditions, the validation functions of its side conditions will be executed before its implementation. Some functions may implement two or more theorems together since our goals may require applying a list of theorems in a certain order without thinking of intermediate formulae.

In the following sections, we illustrate the implementation for some particular theorems. Not all the theorems are implemented in our project because of space limit, others can be written in a similar way.

5.2 Implementation of Tabular Transformation

- Implementation of Theorem 2.8(a)

Function *rtea* implements Theorem 2.8 (replacing table elements)(a) by taking a pair of tables ($TABLE(t1), TABLE(t2)$) as its parameter. It is required that *t1* and *t2* have the same structures; if not, an error will be raised.

```

let rtea = function
| (TABLE(t1), TABLE(t2)) →
  let pt = ref true in
  let mh1 = Array.length t1.headm and mh2 = Array.length t2.headm in
  let nh1 = Array.length t1.headn and nh2 = Array.length t2.headn in
  let mb1 = Array.length t1.body and mb2 = Array.length t2.body in
  let nb1 = Array.length t1.body.(0) and nb2 = Array.length t2.body.(0) in
  if mh1 = mh2 ∧ nh1 = nh2 ∧ mb1 = mb2 ∧ nb1 = nb2 then begin
    for i = 0 to mh1 - 1 do
      pt := !pt ∧ valid (IMPLIES(t1.headm.(i), t2.headm.(i)))
    done;
    for j = 0 to nh1 - 1 do
      pt := !pt ∧ valid (IMPLIES(t1.headn.(j), t2.headn.(j)))
    done;
    for i = 0 to mb1 - 1 do
      for j = 0 to nb1 - 1 do
        if t1.headm ≠ [] ∧ t1.headn ≠ [] then
          pt := !pt ∧ valid (IMPLIES(AND([t1.headm.(i); t1.headn.(j)]),
            IFF(t1.body.(i).(j), t2.body.(i).(j))))
        else if t1.headm = [] ∧ t1.headn ≠ [] then
          pt := !pt ∧ valid (IMPLIES(AND([TRUE; t1.headn.(j)]),
            IFF(t1.body.(i).(j), t2.body.(i).(j))))
        else if t1.headm ≠ [] ∧ t1.headn = [] then
          pt := !pt ∧ valid (IMPLIES(AND([t1.headm.(i); TRUE]),
            IFF(t1.body.(i).(j), t2.body.(i).(j))))
        else
          pt := !pt ∧ valid (IMPLIES(AND([TRUE; TRUE]),
            IFF(t1.body.(i).(j), t2.body.(i).(j))))
      done
    done; !pt end
  else raise (Failure "Two tabular predicates have
    different structures")
| _ → raise (Failure "the first parameter of function
    rtea should be a pair of predicates in tabular form")

```

- Implementation of Theorem 2.9

Function *sjrc ifrow from1 from2* is a procedure to implement Theorem 2.9 (splitting and joining rows and columns). It takes *ifrow*, *from1*, *from2*, and a pair of tables (*TABLE(t1)*, *TABLE(t2)*) as parameters. *ifrow* is a boolean value representing splitting and joining a row if *true* and splitting and joining a column if *false*. The positions of rows (columns) to be joined in table *t2* are stored in parameters *from1* and *from2* where *from1* < *from2*. The position of row(column) to be splitted in table *t1* is equal to *from1*. Positions are counted from number 0.

```

let sjrc ifrow from1 from2 = function
| (TABLE(t1), TABLE(t2)) →
    let pt = ref true in
    let mh1 = Array.length t1.headm in
    let nh1 = Array.length t1.headn in
    let mb1 = Array.length t1.body and mb2 = Array.length t2.body in
    let nb1 = Array.length t1.body.(0) and nb2 = Array.length t2.body.(0) in
    if ((ifrow ∧ mb2 = mb1 + 1 ∧ from2 < mb2 ∧ from1 < from2) ∨
        ((¬ ifrow) ∧ nb2 = nb1 + 1 ∧ from2 < nb2 ∧ from1 < from2))
    then begin
        for i = 0 to mh1 - 1 do
            if ifrow ∧ i < from1 then
                pt := !pt ∧ valid (IFF(t1.headm.(i), t2.headm.(i)))
            else if ifrow ∧ i = from1 then
                pt := !pt ∧ valid (IFF(t1.headm.(i),
                    OR([t2.headm.(from1); t2.headm.(from2)])))
            else if ifrow ∧ i > from1 ∧ i < from2 then
                pt := !pt ∧ valid (IFF(t1.headm.(i), t2.headm.(i)))
            else if ifrow ∧ i = from2 then
                pt := !pt
            else if ifrow ∧ i > from2 then
                pt := !pt ∧ valid (IFF(t1.headm.(i), t2.headm.(i + 1)))
    end

```

```

    else  $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{headm}.(i), t2.\text{headm}.(i)))$ 
done;
for  $j = 0$  to  $nh1 - 1$  do
  if  $\text{ifrow} = \text{false} \wedge j < \text{from1}$  then
     $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{headn}.(j), t2.\text{headn}.(j)))$ 
  else if  $\text{ifrow} = \text{false} \wedge j = \text{from1}$  then
     $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{headn}.(j),$ 
       $\text{OR}([t2.\text{headn}.(from1); t2.\text{headn}.(from2)])))$ 
  else if  $\text{ifrow} = \text{false} \wedge j > \text{from1} \wedge j < \text{from2}$  then
     $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{headn}.(j), t2.\text{headn}.(j)))$ 
  else if  $\text{ifrow} = \text{false} \wedge j = \text{from2}$  then
     $pt := !pt$ 
  else if  $\text{ifrow} = \text{false} \wedge j > \text{from2}$  then
     $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{headn}.(j), t2.\text{headn}.(j + 1)))$ 
  else  $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{headn}.(j), t2.\text{headn}.(j)))$ 
done;
for  $i = 0$  to  $mb1 - 1$  do
  for  $j = 0$  to  $nb1 - 1$  do
    if  $\text{ifrow} \wedge i < \text{from1}$  then
       $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{body}.(i).(j), t2.\text{body}.(i).(j)))$ 
    else if  $\text{ifrow} \wedge i = \text{from1}$  then
       $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{body}.(i).(j),$ 
         $\text{OR}([t2.\text{body}.(from1).(j); t2.\text{body}.(from2).(j)])))$ 
    else if  $\text{ifrow} \wedge i > \text{from1} \wedge i < \text{from2}$  then
       $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{body}.(i).(j), t2.\text{body}.(i).(j)))$ 
    else if  $\text{ifrow} \wedge i = \text{from2}$  then
       $pt := !pt$ 
    else if  $\text{ifrow} \wedge i > \text{from2}$  then
       $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{body}.(i).(j), t2.\text{body}.(i + 1).(j)))$ 
    else if  $\text{ifrow} = \text{false} \wedge j < \text{from1}$  then
       $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{body}.(i).(j), t2.\text{body}.(i).(j)))$ 
    else if  $\text{ifrow} = \text{false} \wedge j = \text{from1}$  then
       $pt := !pt \wedge \text{valid}(\text{IFF}(t1.\text{body}.(i).(j),$ 
         $\text{OR}([t2.\text{body}.(i).(from1); t2.\text{body}.(i).(from2)])))$ 

```

```

    else if ifrow = false ∧ j > from1 ∧ j < from2 then
        pt := !pt ∧ valid (IFF(t1.body.(i).(j), t2.body.(i).(j)))
    else if ifrow = false ∧ j = from2 then
        pt := !pt
    else pt := !pt ∧ valid (IFF(t1.body.(i).(j), t2.body.(i).(j + 1)))
    done
done; !pt
end else raise (Failure "Two_table_structures_or_the_positions_for
    splitting_and_joining_do_not_follow_the_requirement")
| _ → raise (Failure "the_fourth_parameter_of_function_sjrc
    should_be_a_pair_of_predicates_in_tabular_form")

```

5.3 Implementation of Operations on Tabular Predicates

- Implementation of Theorem 3.4(a)

Function *ptimpa1 t s* is the implementation of Theorem 3.4 (predicate-table implication)(a) where $t = TABLE(t1)$ and s is a plain predicate.

```

let ptimpa1 t s = match t with
TABLE(t1) →
    let pt = ref TRUE in
    let pt1 = ref true in
    for i = 0 to (Array.length t1.body) - 1 do
        for j = 0 to (Array.length t1.body.(0)) - 1 do
            if t1.headm ≠ [] ∧ t1.headn ≠ [] then
                pt := AND([t1.headm.(i); t1.headn.(j); t1.body.(i).(j)])
            else if t1.headm = [] ∧ t1.headn ≠ [] then
                pt := AND([t1.headn.(j); t1.body.(i).(j)])
            else if t1.headm ≠ [] ∧ t1.headn = [] then
                pt := AND([t1.headm.(i); t1.body.(i).(j)])

```

```

    else
      pt := t1.body.(i).(j)
    done;
    pt1 := valid (IMPLIES(!pt, s)) ∧ !pt1
  done;
  !pt1
| _ → raise (Failure "the_first_parameter_of_function_ptimpa1
    should_be_a_table")

```

- Implementation of Theorem 3.4(c)

Function *ptimpc t s* is the implementation of theorem 3.4 (Predicate-Table Implication)(c). It takes the same parameter as function *ptimpa1*. It has side condition that headers of table *t1* covers *s*. The side condition is checked before executing theorem proving. An error will be raised if it is not satisfied.

```

let ptimpc t s = match t with
  TABLE(t1) →
    if ¬ ((covers t1.headm s) ∧ (covers t1.headn s)) then raise (Failure
      "the_side_conditions_of_theorem_3.4_are_not_satisfied");
    let pt = ref TRUE in
    let pt1 = ref true in
    for i = 0 to (Array.length t1.body) - 1 do
      for j = 0 to (Array.length t1.body.(0)) - 1 do
        if t1.headm ≠ [] ∧ t1.headn ≠ [] then
          pt := AND([t1.headm.(i); t1.headn.(j); t1.body.(i).(j)])
        else if t1.headm = [] ∧ t1.headn ≠ [] then
          pt := AND([t1.headn.(j); t1.body.(i).(j)])
        else if t1.headm ≠ [] ∧ t1.headn = [] then
          pt := AND([t1.headm.(i); t1.body.(i).(j)])
        else
          pt := t1.body.(i).(j)
      done;
    pt1 := valid (IMPLIES(s, !pt)) ∧ !pt1

```



```

done;
!pt1
| _ → raise (Failure "the first parameter of function ptimpc
      should be a table")

```

5.4 Proof of Precondition and Weakest Precondition

- Proof of Precondition

Suppose $pres$ is our stated precondition for operation S , and $pre\ S$ is derived precondition from program statements. If $pres \Leftrightarrow pre\ S$ is valid, then our conjecture is proved. A statement can be represented by a formula (a relation between initial and final state spaces) and a list of variables of the initial state space.

$type\ statement = ST\ of\ form \times form\ list$

Function $pre\ b\ p$ is defined where p is a procedure of type $statement$, b is our stated precondition of procedure p .

```

let pre b p =
  match p with
  | ST(op, vl) →
    let vl1 = replacelist vl
    in let pred = IFF(b, (EXISTS(vl1, op)))
    in valid pred

```

- Proof of Precondition with Tabular Predicates

Similarly, if a procedure p is a tabular predicate, function $\text{pret } b \ p$ is called to derive its precondition.

```

let pret b p = match p with
  ST(TABLE(t1), vl) →
    let pt = ref TRUE in
    let pt1 = ref true in
    let vl1 = replacelist vl in
    for i = 0 to (Array.length t1.body) - 1 do
      for j = 0 to (Array.length t1.body.(i)) - 1 do
        pt := EXISTS(vl1, t1.body.(i).(j));
        pt1 := valid (IFF(b, !pt)) ∧ !pt1
      done;
    done;
    !pt1
  | _ → raise (Failure "the second parameter of function pret,
    should be a pair of a table and a variable list")

```

- Proof of Weakest Precondition

In our implementation of $\text{wp } b \ p \ c$, p is a program of type *statement*, c is the postcondition of p . We justify that b is the weakest precondition for p to establish c .

```

let wp b p c =
  match p with
  ST(op, vl) →
    let c1 = reformlist c vl
    in let pred = IFF(b, IMPLIES(op, c1))
    in valid pred

```

- Implementation of Theorem 5.4

Function $wpt\ b\ p\ c$ is the implementation of Theorem 5.4 (Weakest Precondition with Predicates)(a). It is different from $wp\ b\ p\ c$ in that operation $p = ST(TABLE(t1), vl)$ is in tabular form. It has side condition that headers of table $t1$ are total and disjoint.

```

let wpt b p c = match p with
  ST(TABLE(t1), vl) →
    if ¬ ((total t1.headm) ∧ (total t1.headn) ∧
      (jointdisjoint (t1.headm, t1.headn))) then raise (Failure
      "the side conditions of theorem 5.4 are not satisfied");
    let pt = ref TRUE in
    let pt1 = ref true in
    let c1 = reformlist c vl in
    for i = 0 to (Array.length t1.body) - 1 do
      for j = 0 to (Array.length t1.body.(0)) - 1 do
        pt := IMPLIES(t1.body.(i).(j), c1);
        pt1 := valid (IFF(b, !pt))
      done;
    done;
    !pt1
| _ → raise (Failure "the second parameter of function wpt
  should be a pair of a table and a variable list")

```

5.5 Implementation of Verification with Predicates

Theorem 5.5 (Tabular Verification with Predicates) does not have the side conditions of totality and disjointness of the headers and does not even require the table to be in standard form. Hence it can always be used to verify that a tabular relation under a given precondition establishes a given postcondition. A special application of it is to verify that an invariant is preserved by an operation in tabular form. Al-

ternatively we can verify the invariant by first deriving the weakest precondition for operation to establish the invariant and then showing in a second step that the invariant implies the weakest precondition. This again results in proof conditions that are identical to those by applying Theorem 5.5 and then eliminating the primed variables with the one-point rule.

Our tool will automatically create proof obligation of Theorem 5.5, one for each body element, with functional program $tv\ p\ b\ p\ c$. Parameter p is the specification for an operation of type *statement*, consisting of a characteristic predicate relation in tabular form and the domain of this relation represented by a list structure; parameters b and c are the precondition and postcondition of p respectively.

```

let tv\ p\ b\ p\ c = match p with
  ST(TABLE(t1), vl) →
    let pt = ref TRUE in
    let pt1 = ref true in
    let c1 = reformlist c vl in
    for i = 0 to (Array.length t1.body) - 1 do
      for j = 0 to (Array.length t1.body.(0)) - 1 do
        if t1.headm ≠ [] ∧ t1.headn ≠ [] then
          pt := AND([b; t1.headm.(i); t1.headn.(j); t1.body.(i).(j)])
        else if t1.headm = [] ∧ t1.headn ≠ [] then
          pt := AND([b; TRUE; t1.headn.(j); t1.body.(i).(j)])
        else if t1.headm ≠ [] ∧ t1.headn = [] then
          pt := AND([b; t1.headm.(i); TRUE; t1.body.(i).(j)])
        else
          pt := AND([b; TRUE; TRUE; t1.body.(i).(j)]);
          pt1 := valid (IMPLIES(!pt, c1)) ∧ !pt1
      done;
    done;
    !pt1
| _ → raise (Failure "the second parameter of function tv\p\

```

```
should_be_a_pair_of_a_table_and_a_variable_list")
```

A variation of Theorem 5.5 verifies the correctness of a program in form of plain predicate relation. Its implementation $vp\ b\ p\ c$ takes the same parameter as the function $tpv\ b\ p\ c$ except that p is $ST(op, vl)$ instead of $ST(TABLE(t1), vl)$.

```
let vp b p c = match p with ST(op, vl) →
  let c1 = reformlist c vl in valid (IMPLIES(AND([b; op]), c1))
```

5.6 Implementation of Refinement

5.6.1 Implementation of Algorithmic Refinement

Specifications can be transformed into more concrete or more abstract ones, where either the concrete or the abstract or both are given in tabular form. In algorithmic refinement, both specifications are over the same state space. A new data type RE is created to combine the concrete specification and abstract specification in terms of their *statement* type. Although both specifications have the same set of variables, they have to be listed to show their integrity. Theorem 7.1 (Refining to Table) is implemented by function $rtt\ p$ where p represents the concrete and abstract specifications of type RE .

```
let rtt p = match p with
  RE(ST(TABLE(t1), vl), ST(TABLE(t2), vlr)) →
    let pt = ref true in
    let mh1 = Array.length t1.headm and mh2 = Array.length t2.headm in
    let nh1 = Array.length t1.headn and nh2 = Array.length t2.headn in
    let mb1 = Array.length t1.body and mb2 = Array.length t2.body in
    let nb1 = Array.length t1.body.(0) and nb2 = Array.length t2.body.(0) in
    if mh1 = mh2 ∧ nh1 = nh2 ∧ mb1 = mb2 ∧ nb1 = nb2 then begin
      for i = 0 to mh1 - 1 do
```

```

    pt := !pt ∧ valid (IFF(t1.headm.(i), t2.headm.(i)));
done;
for j = 0 to nh1 - 1 do
    pt := !pt ∧ valid (IFF(t1.headn.(j), t2.headn.(j)));
done;
for i = 0 to mb1 - 1 do
    for j = 0 to nb1 - 1 do
        pt := !pt ∧ valid (IMPLIES(AND([t1.headm.(i); t1.headn.(j);
            t1.body.(i).(j)]), t2.body.(i).(j)));
    done
done; !pt end
else raise (Failure "Two tabular predicates for algorithmic
    refinement have different structures")
| _ → raise (Failure "the parameter of function rtt
    should be a pair of statements in tabular form")

```

5.6.2 Implementation of Data Refinement

- Implementation of Theorem 6.1 and Theorem 6.2 in one Function

Encoding $Q \subseteq P \downarrow R$ is sound if R is injective; decoding $P \uparrow R \subseteq Q$ is sound if R is total.

As encoding and decoding differ only in the restriction of R , we define a function $drpp\ p\ r\ flag$ to implement coding operation with parameter $flag = 0$ representing an encoding validation and $flag = 1$ representing a decoding validation. Parameter p is specifications of type RE ; r is the coding relation in plain predicate form.

```

let drpp p r flag = match p with
  RE(ST(op1, vl), ST(op2, vlr)) →
    let vl1 = replacelist vl in
    let r1 = reformlist r (vl@vlr) in

```

```

if flag = 0 then
  injectiver r (vl, vlr) ∧
  valid (IMPLIES(op2, EXISTS(vl@vl1, AND([r; op1; r1])))
else
  totalr r (vl, vlr) ∧
  valid (IMPLIES(EXISTS(vl@vl1, AND([r; op1; r1])), op2))

```

- Implementation of Theorem 7.4(b) and 7.1(b) in one Function

Specifically, we can push these three refinement operators into the cells of tables like we do for other relational operators. In this section, we illustrate the use of Theorem 7.4(b) for a data abstraction that reduces the state space and theorem 7.1(b) for a algorithmic abstraction that reduces the non-determinism in one function.

In first order logic, \Leftrightarrow is used to denote equivalent predicate (e.g. bv and bv' are equivalent iff $bv \Leftrightarrow bv'$). The symbol is also extended to relations represented by boolean functions (e.g. let $BV, BV' : X \rightarrow Y \rightarrow Bool$ and $BVxy \equiv bv, BV'xy \equiv bv', BV, BV'$ are equivalent iff $bv \Leftrightarrow bv'$ for any x, y). A relation or matrix can be replaced by its equivalent relation or matrix within any formula or tabular expression. Given two operations in standard form and their abstraction relation by

$$P_1 \begin{array}{c|c} & cv_1 \\ \hline xv & \\ \hline bv_1 & pm_1 \end{array}, \quad P_2 \begin{array}{c|c} & cv_2 \\ \hline yv & \\ \hline bv_2 & pm_2 \end{array}, \quad R \begin{array}{c|c} & \\ \hline xv & yv \\ \hline & r \end{array}$$

If

- $(\exists xv \cdot r \wedge cv_1) \Leftrightarrow cv_2$
- $(\exists xv \cdot r \wedge bv_1) \Leftrightarrow bv_2$
- $bv_2 \wedge cv_2 \wedge (\exists xv, xv' \cdot r \wedge pm_1 \wedge r') \Rightarrow pm_2$

then $P_1 \downarrow R \subseteq P_2$

To prove the above conclusion, we start from the third condition and a logic rule, say $a \wedge b \Rightarrow a$ for any a, b :

$$\begin{aligned}
 & bv_2 \wedge cv_2 \wedge (\exists xv, xv' \cdot r \wedge pm_1 \wedge r') \Rightarrow pm_2 \\
 & bv_2 \wedge cv_2 \wedge (\exists xv, xv' \cdot r \wedge pm_1 \wedge r') \Rightarrow bv_2 \wedge cv_2 \\
 \Rightarrow & \langle\langle \text{if } a \Rightarrow b, a \Rightarrow c \text{ then } a \Rightarrow b \wedge c \rangle\rangle \\
 & bv_2 \wedge cv_2 \wedge (\exists xv, xv' \cdot r \wedge pm_1 \wedge r') \Rightarrow bv_2 \wedge cv_2 \wedge pm_2 \\
 \equiv & \langle\langle \text{condition 1 and 2} \rangle\rangle \\
 & (\exists xv \cdot r \wedge bv_1) \wedge (\exists xv \cdot r \wedge cv_1) \wedge (\exists xv, xv' \cdot r \wedge pm_1 \wedge r') \Rightarrow bv_2 \wedge cv_2 \wedge pm_2 \\
 \equiv & \langle\langle \text{definition of tabular predicate} \rangle\rangle \\
 & \left(\frac{\quad}{\exists xv \cdot r \wedge bv_1} \middle| \frac{\exists xv \cdot r \wedge cv_1}{\exists xv, xv' \cdot r \wedge pm_1 \wedge r'} \right) \Rightarrow \left(\frac{\quad}{bv_2} \middle| \frac{cv_2}{pm_2} \right) \\
 \Rightarrow & \langle\langle \text{Theorem 7.4(b) and transitivity of } \Rightarrow \rangle\rangle \\
 & (P_1 \downarrow R) \ yv \ yv' \Rightarrow \left(\frac{\quad}{bv_2} \middle| \frac{cv_2}{pm_2} \right) \\
 \equiv & \langle\langle \text{definition of } P_2 \ yv \ yv' \rangle\rangle \\
 & (P_1 \downarrow R) \ yv \ yv' \Rightarrow P_2 \ yv \ yv' \\
 \equiv & \langle\langle \text{any } yv, yv' \text{ and definition of } \subseteq \rangle\rangle \\
 & P_1 \downarrow R \subseteq P_2
 \end{aligned}$$

The validation of such an abstraction is defined by function $drpb \ p \ r$ where p is a pair of operations representing the concrete specification (P_1) and abstract specification (P_2) in terms of refinement datatype RE , r is the abstraction relation (R) between two specifications.

```

let drpb p r = match p with
  RE(ST(TABLE(t1), vl), ST(TABLE(t2), vlr)) →
    let pt = ref true in
    let vl1 = replacelist vl in

```



```

let r1 = reformlist r (vl@v1r) in
let mh1 = Array.length t1.headm and mh2 = Array.length t2.headm
in let nh1 = Array.length t1.headn and nh2 = Array.length t2.headn
in let mb1 = Array.length t1.body and mb2 = Array.length t2.body in
let nb1 = Array.length t1.body.(0) and nb2 = Array.length t2.body.(0)
in if mh1 = mh2 ∧ nh1 = nh2 ∧ mb1 = mb2 ∧ nb1 = nb2
then begin
  for i = 0 to mh1 - 1 do
    pt := !pt ∧
      valid (IFF(EXISTS(vl, AND([r; t1.headm.(i)])), t2.headm.(i)))
  done;
  for j = 0 to nh1 - 1 do
    pt := !pt ∧
      valid (IFF(EXISTS(vl, AND([r; t1.headn.(j)])), t2.headn.(j)))
  done;
  for i = 0 to mb1 - 1 do
    for j = 0 to nb1 - 1 do
      pt := !pt ∧ totalr r (vl, v1r) ∧ valid (IMPLIES(AND
        ([t2.headm.(i); EXISTS(vl@v1l, AND([r; t1.body.(i).(j);
          r1]))]; t2.headn.(j)]), t2.body.(i).(j)))
    done
  done; !pt end
else raise (Failure "Two tabular predicates for data
  abstraction have different structures")
| - → raise (Failure "the first parameter of function drpb
  should be a pair of statements in tabular form")

```

- Implementation of Theorem 7.5

We consider the case that the refinement relation rather than the specification is in tabular form. More precisely, we consider the refinement relation being defined by an inverted vector table. This theorem is applied in our elevator floor reached operation refinement. To represent an encoding relation as a

vector table can reduce largely the running time of Simplify validation.

Theorem 7.5 (Data Refinement with Vector Table) is implemented by calling function $drv\ RE(ST(op1, vl), ST(op2, vlr))\ r$ where $op1$ is abstract specification in plain predicate form, $op2$ is refining specification in tabular form, and r is the vector table of refinement relation.

```

let drv p r = match (p, r) with
  (RE(ST(op1, vl), ST(TABLE(t2), vlr)), TABLE(rt)) →
    let pt = ref (injectiver r (vl, vlr)) in
    let op11 = ref TRUE and op12 = ref TRUE in
    let headm1 = Array.create (Array.length rt.body) TRUE in
    let mh1 = Array.length rt.headn and mh2 = Array.length t2.headm in
    let nh1 = Array.length rt.headn and nh2 = Array.length t2.headn in
    let mb1 = Array.length rt.body.(0) and mb2 = Array.length t2.body in
    let nb1 = Array.length rt.body.(0) and nb2 = Array.length t2.body.(0)
    in if mh1 = mh2 ∧ nh1 = nh2 ∧ mb1 = mb2 ∧ nb1 = nb2
    then begin
      for i = 0 to mh1 - 1 do
        pt := !pt ∧ valid (IFF(rt.headn.(i), t2.headm.(i)));
      done;
      for j = 0 to nh1 - 1 do
        pt := !pt ∧ valid (IFF(reformlist rt.headn.(j) vlr, t2.headn.(j)));
      done;
      for i = 0 to mb1 - 1 do
        for j = 0 to nb1 - 1 do
          for k = 0 to (Array.length rt.body) - 1 do
            headm1.(k) ← reformlist rt.headm.(k) [rt.headm.(k)];
            op11 := replace (rt.headm.(k), rt.body.(k).(i)) op1;
            op12 := replace (headm1.(k), rt.body.(k).(j)) !op11
          done;
          pt := !pt ∧ valid (IFF(t2.body.(i).(j), !op12));
        done
      done; !pt end

```

```
    else raise (Failure "The result of applying tabular refinement  
relation on specification have different structure with t2")  
  | - → raise (Failure "the first parameter of function drp  
    should be a pair of statements with one in tabular form,  
    the second parameter of function drp should be a table")
```

Chapter 6

Luxury Sedan Car Seat Case Study

In order to demonstrate how to apply tabular specification theorems to reactive systems, in this chapter we give a luxury sedan car seat example consisting of several operations. We first study how to formalize the example, then derive properties supported by the system and come up with a specification for each operation. Since no parallel operations are allowed, a control has to be enforced by first order logic. The priority control can be implemented by a condition lying in the header of an operation table. Sequential operations are more difficult to define in a first order logic style. A sequence of events are dependent each other, the post-condition of one event will be the pre-condition of the next event.

At last we state how to implement a proof that each single operation including tables or plain formulae preserves a property with the theorem functions. We focus on the proof of two parallel groups of operations where motors per group are active in a time order. The disjoint events of our car seat system can be realized by making two or more events in one table and distinguishing them with different parameters. A priority control can be stated in plain words or fulfilled by adding conditions conjuncted with original conditions. A formal proof is implemented and its logical principle is stated

for the latter one.

6.1 Requirement

Our single front car seat is an automated control system. In total five motors are to be controlled:

- Rear Height Motor (RH)
- Longitudinal Adjustment Motor (LA)
- Front Height Motor (FH)
- Backrest Motor (B)
- Head Restraint Motor (HR)

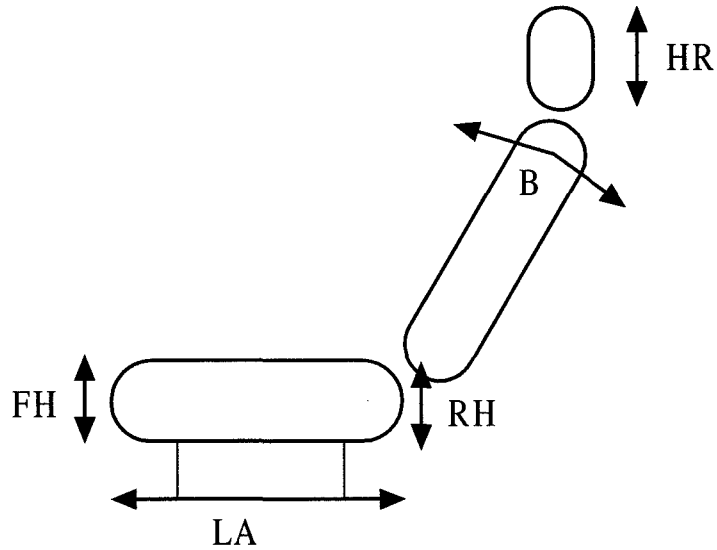


Figure 6.1: Controlling a Luxury Sedan Car Seat

These determine the position of the car seat as shown in Figure 6.1

Each seat has a panel through which the adjustments are made (panel is not shown).

The panel has following 13 buttons:

- RH up, RH down
- LA forward, LA backward
- FH up, FH down
- B forward, B backward
- HR up, HR down
- M, M1, M2 (memory)

The first ten buttons control the respective motors: while RH up is pressed, the RH motor moves up until the stop position (see below about stop positions) and similarly for the other buttons. Pressing the M1 button while holding the M button causes the current adjustment positions to be stored in memory 1, while pressing the M2 button while holding the M button causes the current adjustment positions to be stored in memory 2. Pressing M1 causes the adjustments stored in memory 1 to be retrieved and set. Likewise, pressing M2 causes the adjustments stored in memory 2 to be retrieved and set.

Calibration. When supplied with electricity for the first time, or after a power failure, the seat has to be calibrated by moving all motors to their respective home positions. This is done by turning on each motor and waiting until its home position sensor signals that the home position has been reached. The home positions are the front stop for LA and B and the upper stop for RH, RH, and HR. The sensors generate the events "LA home", "B home", etc.

Priorities. The seat adjustment motors are divided into two groups. Only one motor per group can be active at a time:

- Group 1 with LA and RH: LA has priority over RH.
- Group 2 with B, FH, and HR: B has priority over FH, which has priority over HR.

For example, if both LA forward and RH up are pressed on the panel, first LA is moved and only then RH. Likewise, these priorities have to be considered when moving the seat to an adjustment stored in memory and during calibration.

Buttons. Pressing and releasing a button of the panel generate each an event. The events are called "HR up pressed", "HR up released", etc.

Motors. Each motor can be in the state *up*, *stop*, *down* or *forward*, *stop*, *backward*, respectively.

Positioning. Each motor is equipped with a Hall sensor. The Hall sensor generates a "tick" on each rotation. The position of each motor is determined by incrementing and decrementing the number of ticks after calibration. The number of ticks required for movement from one stop to the other is:

- RH: 250 ticks
- LA: 600 ticks
- FH: 250 ticks
- B: 1100 ticks
- HR: 130 ticks

The events generated are called "RH tick", "LA tick", etc. After calibration only the ticks are used for keeping track of the position, in particular for reaching either stop position.

6.2 Organization

We distinguish four modes of the car seat controller:

- Calibrating mode, the initial mode;
- Normal mode, when the seat can be adjusted through the panel;
- Memory mode, when the adjustments from memory 1 or memory 2 are retrieved and set;
- Memory Set mode, when current adjustments are stored in memory 1 or memory 2.

We use following global variables, all of which are of type integer:

- `lpos`, `rhpos`, `bpos`, `fhpos`, `hrpos`: for the current positions of the motors;
- `lam1`, `rhm1`, `bm1`, `fhm1`, `hrm1`: the positions stored as memory 1;
- `lam2`, `rhm2`, `bm2`, `fhm2`, `hrm2`: the positions stored as memory 2;
- `laset`, `rhset`, `bset`, `fhset`, `hrset`: used for passing the positions to be set.

The events created by pressing an up button or down button of the same motor are considered as disjoint events and represented as a single table with different parameters. "RH up pressed" and "RH down pressed" are formalized as $p = rhup$ and

$p = rhdown$. If first "RH down" and then "RH up" is pressed without releasing "RH down", the event "RH up pressed" is ignored.

The motor movement events will trigger the motor *tick* events. When a motor behavior cause its Hall sensor increment or decrement one rotation, a tick is generated to determine the position change of the motor. To make the tick table total, we add a column with condition motor equals stop to indicate no change of position generated in that state.

In *Normal* mode, the priorities are taken into account as follows: If, for example, RH is in state *up* and an *LA forward pressed* event is received, the RH motor is stopped and LA goes to state *forward*. Once an *LA released* event is received, the LA motor stopped and require that the user releases and pressed the button for RH up again.

In *Calibration* mode, a series of motor movements within a group is arranged as follows: If B motor, FH motor and HR motor are not in their home positions, B motor is turned on first, FH motor and HR motor wait until B's home position sensor signals that the home position has been reached, then B motor is stopped and FH goes to state *up*. When both B motor and FH motor are stopped, HR motor goes to its upper stop. *Memory* mode has the similar movement as *Calibration* mode, the only difference is that LA and B goes *forward*, and RH, FH, and HR goes *up* in former, while LA and B goes *forward* or *backward*, and RH, FH, and HR goes *up* or *down* in latter depending on the current positions of the motors and the positions to be set to mode *Memory*.

There are two groups of states, corresponding to motor movement: *forward*, *backward*, *stop* and *up*, *down*, *stop*. The field of button selection f, b, u, d correspond a control of *forward*, *backward*, *up*, *down* movement respectively. We use $button = m?$ to indicate the memory button $m?$ being pressed. Table 6.1 list all variables in car

| Variable | Representation | Type |
|-----------------|--------------------|---------------------------|
| <i>labutton</i> | LA motor buttons | f, b |
| <i>bbutton</i> | B motor buttons | |
| <i>rhbutton</i> | RH motor buttons | u, d |
| <i>fhbutton</i> | FH motor buttons | |
| <i>hrbutton</i> | HR motor buttons | |
| <i>button</i> | Memory buttons | $m1, m2$ |
| <i>lamotor</i> | LA motor movements | $forward, backward, stop$ |
| <i>bmotor</i> | B motor movements | |
| <i>rhmotor</i> | RH motor movements | $upward, downward, stop$ |
| <i>fhmotor</i> | FH motor movements | |
| <i>hrmotor</i> | HR motor movements | |

Table 6.1: Variables in Car Seat Control

seat control. As we stated before, any variable belonging to an enumeration type should have its type specified. We specify motor movement variables in our defined axiom file.

```
(DISTINCT forward backward stop)

(OR (EQ lamotor backward) (EQ lamotor forward) (EQ lamotor stop))
(OR (EQ lamotor1 backward) (EQ lamotor1 forward) (EQ lamotor1 stop))
(OR (EQ bmotor backward) (EQ bmotor forward) (EQ bmotor stop))
(OR (EQ bmotor1 backward) (EQ bmotor1 forward) (EQ bmotor1 stop))

(DISTINCT up down stop)

(OR (EQ rhmotor down) (EQ rhmotor up) (EQ rhmotor stop))
(OR (EQ rhmotor1 down) (EQ rhmotor1 up) (EQ rhmotor1 stop))
(OR (EQ fhmotor down) (EQ fhmotor up) (EQ fhmotor stop))
(OR (EQ fhmotor1 down) (EQ fhmotor1 up) (EQ fhmotor1 stop))
(OR (EQ hrmotor down) (EQ hrmotor up) (EQ hrmotor stop))
(OR (EQ hrmotor1 down) (EQ hrmotor1 up) (EQ hrmotor1 stop))
```

6.3 Normal Mode and Its Properties

6.3.1 Types of Motor Adjustment Buttons

The variables in normal mode are specified in file *myaxiom.ax* as follows.

```
(DISTINCT f b)
(OR (EQ labutton f) (EQ labutton b))
(OR (EQ bbutton f) (EQ bbutton b))
(DISTINCT u d)
(OR (EQ rhbutton u) (EQ rhbutton d))
(OR (EQ fhbutton u) (EQ fhbutton d))
(OR (EQ hrbutton u) (EQ hrbutton d))
```

Note that we only specify the initial state for motor button variables since they occur only in conditions.

6.3.2 Releasing Motor Adjustment Buttons

Motor movement events generated by releasing the first ten buttons always make motors stop. The definitions of the relations over variables *lamotor*, *rhmotor*, *bmotor*, *fhmotor*, and *hrmotor* are as listed:

```
laupreleased = (lamotor1 = stop); lardownreleased = (lamotor1 = stop)
rhupreleased = (rhmotor1 = stop); rhdownreleased = (rhmotor1 = stop)
bupreleased = (bmotor1 = stop); bdownreleased = (bmotor1 = stop)
fhupreleased = (fhmotor1 = stop); fhdownreleased = (fhmotor1 = stop)
hrupreleased = (hrmotor1 = stop); hrdownreleased = (hrmotor1 = stop)
```

6.3.3 Pressing Group 1 Motor Adjustment Buttons

Motor movement events generated by pressing the first ten buttons trigger the corresponding motor tick events. We consider these two events of each motor into one relation which is represented by a tabular predicate. We describe group 1 event tables in detail. Group 2 event tables in Normal mode follow the same rule.

Table 6.2 is the tabular representation of LA Motor Button Pressed *lapressed* over variables *lamotor*, *lapos*. The left header indicates the condition whether forward or backward button is pressed. The upper header is the conditional predicate relating to the current position of LA motor.

| | $lapos = 0$ | $lapos > 0 \wedge lapos < 599$ | $lapos = 599$ |
|----------------|---|---|--|
| $labutton = f$ | $lamotor1 = stop \wedge lapos1 = lapos$ | $lamotor1 = forward \wedge lapos1 = lapos - 1$ | $lamotor1 = forward \wedge lapos1 = lapos - 1$ |
| $labutton = b$ | $lamotor1 = backward \wedge lapos1 = lapos + 1$ | $lamotor1 = backward \wedge lapos1 = lapos + 1$ | $lamotor1 = stop \wedge lapos1 = lapos$ |

Table 6.2: LA Motor Button Pressed *lapressed*

An invariant holds in LA Normal mode movement: The range of motor movement is from 0 to 599, formally expressed by *InvLA* :

$$0 \leq lapos < 600.$$

To prove the invariant is preserved by *lapressed*, we simply call function *tpv* *b p c* with parameter $b = c = InvLA, p = (op, d)$. The formula *op* is *lapressed* parsed to type *form* and the domain *d* is [*lamotor*, *lapos*].

An export function `reader` in `parse.ml` is used to read *lamotor*, *lapos* as variable names, *lapressed* as program specification, *b1* as precondition, *c1* as postcondition.

```
let lapressed = reader "BEGTAB_LHEADER_labutton=f_$labutton=b//
    UHEADER_lapos=0_$lapos>0_&_lapos<599_$lapos=599//_lamotor1=stop_&
    lapos1=lapos_$lamotor1=forward_&_lapos1=lapos-1_$lamotor1=forward_&
    lapos1=lapos-1//_lamotor1=backward_&_lapos1=lapos+1_$lamotor1=backward
    &_lapos1=lapos+1_$lamotor1=stop_&_lapos1=_lapos//_ENDTAB"
in let p1 = ST(lapressed,[reader "lamotor";reader "lapos"])
in let b1 = reader "lapos_>=_0_&_lapos_<_600"
in let c1 = reader "lapos_>=_0_&_lapos_<_600"
in if (typ b1 p1 c1) then begin
    ...print lapressed to screen and a latex file...
    ...inform the invariant hold...
end;
```

The \LaTeX printout of LA motor button pressed operation is shown below.

```
\begin{longtable}{|c|c|c|c|}
\hline & \texttt{1}\texttt{a}\texttt{p}\texttt{o}\texttt{s}=\texttt{0}
& \texttt{1}\texttt{a}\texttt{p}\texttt{o}\texttt{s} \$>$ \texttt{0}
$&\texttt{1}\texttt{a}\texttt{p}\texttt{o}\texttt{s}&\texttt{1}
\texttt{a}\texttt{p}\texttt{o}\texttt{s} = \texttt{5}\texttt{9}
\texttt{9}&& &\texttt{5}\texttt{9}\texttt{9}&\texttt{9}& \texttt{1}
\texttt{a}\texttt{b}\texttt{u}\texttt{t}\texttt{t}\texttt{o}\texttt{n}
= \texttt{f} & \texttt{1}\texttt{a}\texttt{m}\texttt{o}\texttt{t}\texttt{o}\texttt{r}
\texttt{1} = \texttt{s}\texttt{t}\texttt{o}
\texttt{p} \$&\texttt{1}\texttt{a}\texttt{m}\texttt{o}\texttt{r}
\texttt{t}\texttt{o}\texttt{r} \texttt{1} = & \texttt{1}\texttt{a}
\texttt{m}\texttt{o}\texttt{r}\texttt{t} \texttt{o}\texttt{r}\texttt{1} =\&
\texttt{1}\texttt{a}\texttt{p}\texttt{o}\texttt{s}\texttt{1} =
```

```

\texttt{l}\texttt{a}\texttt{p}\texttt{o}\texttt{s} & \texttt{f}
\texttt{o}\texttt{r}\texttt{w}\texttt{a}\texttt{r}\texttt{d} $\wedge$
\texttt{l}\texttt{a}\texttt{p}\texttt{o}\texttt{s}\texttt{l} &
\texttt{f}\texttt{o}\texttt{r}\texttt{w}\texttt{a}\texttt{r}\texttt{d}
$\wedge$ \texttt{l}\texttt{a}\texttt{p}\texttt{o}\texttt{s}\texttt{l}
\\ & = \texttt{l}\texttt{a} \texttt{p}\texttt{o}\texttt{s} - \texttt{l}
& = \texttt{l}\texttt{a}\texttt{p}\texttt{o}\texttt{s} - \texttt{l}
\hline \texttt{l}\texttt{a}\texttt{b}\texttt{u}\texttt{t}\texttt{t}
\texttt{o}\texttt{n} = \texttt{b} & \texttt{l}\texttt{a}\texttt{m}
\texttt{o}\texttt{t}\texttt{o}\texttt{r}\texttt{l} = & \texttt{l}
\texttt{a}\texttt{m}\texttt{o}\texttt{t}\texttt{o}\texttt{r}\texttt{l}
= & \texttt{l}\texttt{a}\texttt{m}\texttt{o}\texttt{t}\texttt{t}\texttt{o}
\texttt{r}\texttt{l} = \texttt{s}\texttt{t}\texttt{o}\texttt{p}
$\wedge$ & \texttt{b}\texttt{a}\texttt{c}\texttt{k}\texttt{w}
\texttt{a}\texttt{r}\texttt{d} $\wedge$ \texttt{l}\texttt{a}\texttt{p}
\texttt{o}\texttt{s}\texttt{l} & \texttt{b}\texttt{a}\texttt{c}
\texttt{k}\texttt{w}\texttt{a}\texttt{r}\texttt{d} $\wedge$ \texttt{l}
\texttt{a}\texttt{p}\texttt{o}\texttt{s}\texttt{l} & \texttt{l}
\texttt{a}\texttt{p}\texttt{o}\texttt{s}\texttt{l} = \texttt{l}
\texttt{a}\texttt{p}\texttt{o}\texttt{s}\\ & = \texttt{l}\texttt{a}
\texttt{p}\texttt{o}\texttt{s} + \texttt{l} & = \texttt{l}\texttt{a}
\texttt{p}\texttt{o}\texttt{s} + \texttt{l} & \\ \hline
\caption{LA Motor Button Pressed $\lapressed$}\end{longtable}

```

To consider the priority in group 1 motor movements triggered by LA forward, LA backward, RH up, and RH down button pressed events, we add a condition *lamotor* =

stop conjoined with the original condition into the left header of RH movement table:

| | $rhpos = 0$ | $rhpos < 249 \wedge$ $rhpos > 0$ | $rhpos = 249$ |
|---|---|---|---|
| $rhbutton = u \wedge$ $lamotor = stop$ | $rhmotor1 = stop$ $\wedge rhpos1 = rhpos$ | $rhmotor1 = up \wedge$ $rhpos1 = rhpos -$ 1 | $rhmotor1 = up \wedge$ $rhpos1 = rhpos -$ 1 |
| $rhbutton = d \wedge$ $lamotor = stop$ | $rhmotor1 = down$ $\wedge rhpos1 = rhpos$ $+ 1$ | $rhmotor1 = down$ $\wedge rhpos1 = rhpos$ $+ 1$ | $rhmotor1 = stop$ $\wedge rhpos1 = rhpos$ |
| $(rhbutton = u \vee$ $rhbutton = d) \wedge$ $lamotor \neq stop$ | $rhmotor1 = stop$ $\wedge rhpos1 = rhpos$ | $rhmotor1 = stop$ $\wedge rhpos1 = rhpos$ | $rhmotor1 = stop$ $\wedge rhpos1 = rhpos$ |

Table 6.3: RH Motor Button Pressed *rhpressed*

The invariant of RH movement in normal mode is similar to LA movement: The range of RH motor movement is from 0 to 249, formally expressed by *InvRH* :

$$0 \leq rhpos < 250.$$

The sketch of the proof can be referred to those stated for LA. The implementation of our proof in Ocaml is:

```
let rhpressed = reader "BEGTAB_LHEADER_rhbutton=u&lamotor=stop_$
rhbutton=d&lamotor=stop$(rhbutton=u_or_rhbutton=d)&lamotor/=stop//
UHEADER_rhpos=0$_rhpos<249&_rhpos>0$_rhpos=249_//_rhmotor1=stop&
rhpos1=rhpos$rhmotor1=up&rhpos1=rhpos-1$rhmotor1=up&rhpos1=rhpos-1
//rhmotor1=down&rhpos1=rhpos+1$_rhmotor1=down&rhpos1=rhpos+1_$
rhmotor1=stop&rhpos1=rhpos_//_rhmotor1=stop&rhpos1=rhpos_$
rhmotor1=stop&rhpos1=rhpos$_rhmotor1=stop&rhpos1=rhpos_//_ENDTAB"
in let p1 = ST(rhpressed,[reader "rhmotor";reader "rhpos"])
```

```

in let b1 = reader "rhpos<=<0_&_rhpos<_250"
in let c1 = reader "rhpos<=<0_&_rhpos<_250"
in if (tvp b1 p1 c1) then begin
  ...print rhpressed to screen and a latex file...
  ...inform the invariant holds...
end;

```

6.3.4 Pressing Group 2 Motor Adjustment Buttons

In principle, tabular representations of group 2 motor movements controlled through the panel have the similar features as those of group 1. For B motor forward or backward button pressed event and B tick event triggered by it, we have the table:

| | $bpos = 0$ | $bpos < 1099 \wedge$ $bpos > 0$ | $bpos = 1099$ |
|---------------|--|--|---|
| $bbutton = f$ | $bmotor1 = stop \wedge$ $bpos1 = bpos$ | $bmotor1 = forward$ $\wedge bpos1 = bpos -$ 1 | $bmotor1 = forward$ $\wedge bpos1 = bpos -$ 1 |
| $bbutton = b$ | $bmotor1 =$ $backward \wedge bpos1$ $= bpos + 1$ | $bmotor1 =$ $backward \wedge bpos1$ $= bpos + 1$ | $bmotor1 = stop \wedge$ $bpos1 = bpos$ |

Table 6.4: B Motor Button Pressed *bpressed*

The invariant of B motor movement in normal mode is formally presented as:

$$0 \leq bpos < 1100$$

The formal proof in terms of implementation is:

```

let bpressed = reader "BEGTAB_LHEADER_bbutton=f_$bbutton=b_/"

```



```

UHEADER_bpos=_0_$bpos<_1099_&bpos>_0_$bpos=_1099_//
bmotor1=_stop_&bpos1=bpos_$bmotor1=forward_&bpos1=bpos-1_$
bmotor1=_forward_&bpos1=bpos-1//bmotor1=backward&bpos1=bpos+1_$
bmotor1=backward&bpos1=bpos+1_$bmotor1=stop&bpos1=bpos//_ENDTAB"
in let p1 = ST(bpressed,[reader "bmotor";reader "bpos"])
in let b1 = reader "bpos">=_0_&bpos<_1100"
in let c1 = reader "bpos">=_0_&bpos<_1100"
in if (tvp b1 p1 c1) then begin
  ...print bpressed to screen and a latex file...
  ...inform the invariant holds...
end;

```

For FH motor up or down button pressed event and FH tick event triggered by it, we have the following table:

| | fhpos = 0 | fhpos < 249 ∧ fhpos > 0 | fhpos = 249 |
|---|--|--|--|
| fhbutton = u ∧ bmotor = stop | fhmotor1 = stop ∧ fhpos1 = fhpos | fhmotor1 = up ∧ fhpos1 = fhpos - 1 | fhmotor1 = up ∧ fhpos1 = fhpos - 1 |
| fhbutton = d ∧ bmotor = stop | fhmotor1 = down ∧ fhpos1 = fhpos + 1 | fhmotor1 = down ∧ fhpos1 = fhpos + 1 | fhmotor1 = stop ∧ fhpos1 = fhpos |
| (fhbutton = u ∨ fhbutton = d) ∧ bmotor ≠ stop | fhmotor1 = stop ∧ fhpos1 = fhpos | fhmotor1 = stop ∧ fhpos1 = fhpos | fhmotor1 = stop ∧ fhpos1 = fhpos |

Table 6.5: FH Motor Button Pressed *fhpressed*

The invariant of FH motor movement in normal mode is formally presented as:

$$0 \leq fhpos < 250$$

The formal proof in terms of implementation is:

```
let fhpressed = reader "BEGTAB_LHEADER_fhbutton=u&_bmotor=stop_$
    fhbutton=d&bmotor=stop_$ (fhbutton=u_or_&fhbutton=d)&bmotor/=stop//
    UHEADER_fhpos=0_$fhpos<249&fhpos>0_$fhpos=249_/_fhmotor1=stop_&
    fhpos1=fhpos$fhmotor1=up&fhpos1=fhpos-1$fhmotor1=up&fhpos1=fhpos-1
    //fhmotor1=down&fhpos1=fhpos+1_$fhmotor1=down&fhpos1=fhpos+1_$
    fhmotor1=stop&fhpos1=fhpos_/_fhmotor1=stop&fhpos1=fhpos_$
    fhmotor1=stop&fhpos1=fhpos_$fhmotor1=stop&fhpos1=fhpos_/_ENDTAB"
in let p1 = ST(fhpressed,[reader "fhmotor";reader "fhpos"])
in let b1 = reader "fhpos_>=_0_&_fhpos_<_250"
in let c1 = reader "fhpos_>=_0_&_fhpos_<_250"
in if (tvp b1 p1 c1) then begin
    ...print fhpressed to screen and a latex file...
    ...inform the invariant holds...
end;
```

For HR motor up or down button pressed event and HR tick event triggered by it, we have the table as follows:

| | $hrpos = 0$ | $hrpos < 129 \wedge$ $hrpos > 0$ | $hrpos = 129$ |
|---|---|---|---|
| $hrbutton = u \wedge$ $bmotor = stop \wedge$ $fhmotor = stop$ | $hrmotor1 = stop$ $\wedge hrpos1 = hrpos$ | $hrmotor1 = up \wedge$ $hrpos1 = hrpos -$ 1 | $hrmotor1 = up \wedge$ $hrpos1 = hrpos -$ 1 |
| $hrbutton = d \wedge$ $bmotor = stop \wedge$ $fhmotor = stop$ | $hrmotor1 = down$ $\wedge hrpos1 = hrpos$ $+ 1$ | $hrmotor1 = down$ $\wedge hrpos1 = hrpos$ $+ 1$ | $hrmotor1 = stop$ $\wedge hrpos1 = hrpos$ |
| $(hrbutton = u \vee$ $hrbutton = d) \wedge$ $(bmotor \neq stop$ $\vee fhmotor \neq$ $stop)$ | $hrmotor1 = stop$ $\wedge hrpos1 = hrpos$ | $hrmotor1 = stop$ $\wedge hrpos1 = hrpos$ | $hrmotor1 = stop$ $\wedge hrpos1 = hrpos$ |

Table 6.6: FH Motor Button Pressed *fhpressed*

The invariant of FH motor movement in normal mode is formally presented as:

$$0 \leq hrpos < 130$$

The formal proof of the invariants in terms of implementation is:

```
let hrpressed = reader "BEGTAB_LHEADER_hrbutton=_u_&_bmotor=_stop_&
    fhmotor=_stop_$hrbutton=_d_&_bmotor=_stop_&_fhmotor=_stop_$
    (hrbutton=_u_or_hrbutton=_d)_&_(bmotor/=stop_or_fhmotor/=stop)_//
    UHEADER_hrpos=_0_$hrpos<_129_&_hrpos>_0_$hrpos=_129_//
    hrmotor1=stop_&_hrpos1=hrpos_$hrmotor1=up_&_hrpos1=hrpos-1_$
    hrmotor1=up_&_hrpos1=hrpos-1_//_hrmotor1=down_&_hrpos1=hrpos+1_$
```

```

    hrmotor1=down_&_hrpos1=hrpos+1_$ _hrmotor1=_stop_&_hrpos1=hrpos_//
    hrmotor1=stop&hrpos1=hrpos_$ _hrmotor1=stop&hrpos1=hrpos_$
    hrmotor1=stop&hrpos1=hrpos_//_ENDTAB"
in let p1 = ST(hrpressed,[reader "hrmotor";reader "hrpos"])
in let b1 = reader "hrpos_>=_0_&_hrpos_<_130"
in let c1 = reader "hrpos_>=_0_&_hrpos_<_130"
in if (tvp b1 p1 c1) then begin
    ...print hrpressed to screen and a latex file...
    ...inform the invariant holds...
end;

```

6.4 Memory and Memory Set Mode

6.4.1 Memory Mode and Its Properties

Same as in Normal mode, there are two groups of motor movements in Memory mode. A motor in group 1 and any one motor in group 2 can be running concurrently. But all motors in one group must move in a time order to reach their setting positions. RH is adjusted after LA sets; FH is adjusted after B sets while HR is adjusted after both B and FH sets. We define operation *latoset* over variable *lamotor* to be a single LA movement to its setting position. A predicate representation of *latoset* as a table is shown below:

| | | |
|-----------------|---------------------|--------------------|
| lapos = laset | lapos < laset | lapos > laset |
| lamotor1 = stop | lamotor1 = backward | lamotor1 = forward |

Table 6.7: LA Motor Moving to Its Setting *latoset*

Its definition in the implementation is as follows:

```

let deflatoset = "BEGTAB_UHEADER_lapos=_laset_$ _lapos_<_laset_$

```

```
lapos_>_laset_/_lamotor1=_stop_$lamotor1=_backward_$
lamotor1=_forward_/_ENDTAB" in
let latoset = reader deflatoset in
...print latoset to screen and a latex file...
```

Similarly, *rhtoset*, *btoreset*, *fhtoset* and *hrtoset* are defined as well.

| | | |
|-----------------|-----------------|---------------|
| rhpos = rhset | rhpos < rhset | rhpos > rhset |
| rhmotor1 = stop | rhmotor1 = down | rhmotor1 = up |

Table 6.8: RH Motor Moving to Its Setting *rhtoset*

```
let defrhtoset = "BEGTAB_UHEADER_rhpos=_rhset_$rhpos<_rhset_$
rhpos>_rhset_/_rhmotor1=_stop_$rhmotor1=_down_$
rhmotor1=_up_/_ENDTAB" in
let rhtoset = reader defrhtoset in
...print rhtoset to screen and a latex file...
```

| | | |
|----------------|--------------------|-------------------|
| bpos = bset | bpos < bset | bpos > bset |
| bmotor1 = stop | bmotor1 = backward | bmotor1 = forward |

Table 6.9: B Motor Moving to Its Setting *btoreset*

```
let defbtoreset = "BEGTAB_UHEADER_bpos=_bset_$bpos<_bset_$
bpos>_bset_/_bmotor1=_stop_$bmotor1=_backward_$
bmotor1=_forward_/_ENDTAB" in
let btoreset = reader defbtoreset in
...print btoreset to screen and a latex file...
```

| | | |
|-------------------|-------------------|-----------------|
| $fhpos = fhset$ | $fhpos < fhset$ | $fhpos > fhset$ |
| $fhmotor1 = stop$ | $fhmotor1 = down$ | $fhmotor1 = up$ |

Table 6.10: FH Motor Moving to Its Setting *fhtoset*

```

let deffhtoset = "BEGTAB_UHEADER_fhpos=_fhset_$fhpos<_fhset_$
fhpos>_fhset_/_fhmotor1=_stop_$fhmotor1=_down_$
fhmotor1=_up_/_ENDTAB" in
let fhtoset = reader deffhtoset in

...print fhtoset to screen and a latex file...

```

| | | |
|-------------------|-------------------|-----------------|
| $hrpos = hrset$ | $hrpos < hrset$ | $hrpos > hrset$ |
| $hrmotor1 = stop$ | $hrmotor1 = down$ | $hrmotor1 = up$ |

Table 6.11: HR Motor Moving to Its Setting *hrtoset*

```

let defhrtoset = "BEGTAB_UHEADER_hrpos=_hrset_$hrpos<_hrset_$
hrpos>_hrset_/_hrmotor1=_stop_$hrmotor1=_down_$
hrmotor1=_up_/_ENDTAB" in
let hrtoset = reader defhrtoset in

...print hrtoset to screen and a latex file...

```

The system rule is modeled in first order logic as:

$$\begin{aligned}
& latoset \wedge (laset = lapos \implies rhtoset) \wedge btoset \wedge (bset = bpos \implies fhtoset) \wedge \\
& (bset = bpos \wedge fhset = fhpos \implies hrtoset).
\end{aligned}$$

Difference is made depending on which memory button is pressed. For instance, pressing M1 causes the adjustments stored in memory 1 to be retrieved as: $laset = lam1 \wedge rhset = rhm1 \wedge bset = bm1 \wedge fhset = fhm1 \wedge hrset = hrm1$, and this further imply our Memory mode rule. Below is the tabular predicate of Memory mode system

named *memory*.

| | |
|-------------|--|
| button = m1 | $\begin{aligned} & \text{laset} = \text{lam1} \wedge \text{rhset} = \text{rhml} \wedge \text{bset} = \text{bm1} \wedge \text{fhset} = \text{fhm1} \wedge \\ & \text{hrset} = \text{hrm1} \implies \text{latoset} \wedge (\text{laset} = \text{lapos} \implies \text{rhtoset}) \wedge \\ & \text{btoset} \wedge (\text{bset} = \text{bpos} \implies \text{fhtoset}) \wedge (\text{bset} = \text{bpos} \wedge \text{fhset} \\ & \quad = \text{fhpos} \implies \text{hrtoset}) \end{aligned}$ |
| button = m2 | $\begin{aligned} & \text{laset} = \text{lam2} \wedge \text{rhset} = \text{rhml} \wedge \text{bset} = \text{bm2} \wedge \text{fhset} = \text{fhm2} \wedge \\ & \text{hrset} = \text{hrm2} \implies \text{latoset} \wedge (\text{laset} = \text{lapos} \implies \text{rhtoset}) \wedge \\ & \text{btoset} \wedge (\text{bset} = \text{bpos} \implies \text{fhtoset}) \wedge (\text{bset} = \text{bpos} \wedge \text{fhset} \\ & \quad = \text{fhpos} \implies \text{hrtoset}) \end{aligned}$ |

Table 6.12: Memory Mode Movement *memory*

One obvious property of *memory* is that when all motors reach their setting positions, they are stopped. Actually, if we state "all motors are stopped" *allstop* :

$$\text{lamotor} = \text{stop} \wedge \text{rhmotor} = \text{stop} \wedge \text{bmotor} = \text{stop} \wedge \text{fhmotor} = \text{stop}$$

$$\wedge \text{hrmotor} = \text{stop}$$

as the postcondition of *memory*, the weakest precondition will be "all motors reach their setting positions" *alltoset* :

$$\text{laset} = \text{lapos} \wedge \text{rhset} = \text{rhpos} \wedge \text{bset} = \text{bpos} \wedge \text{fhset} = \text{fhpos} \wedge \text{hrset} = \text{hrpos}.$$

The proof of our weakest precondition of program *memory* to establish our postcondition takes following three steps:

1. Make axioms according to system requirement of *memory* mode

(a) Variable button is of enumeration type and should be specified.

$$(\text{DISTINCT } m1 \ m2) \ (\text{OR } (\text{EQ } \text{button } m1) \ (\text{EQ } \text{button } m2))$$

- (b) Since all cases in the sub-operations of memory mode are considered, tables (*latoset*, *rhotoset*, *btoreset*, *fhtoset* and *hrtoset*) should be total.

```
(OR (AND (EQ lamotor1 backward) (> laset lapos))
    (AND (EQ lamotor1 forward) (< laset lapos))
    (AND (EQ lamotor1 stop) (EQ laset lapos)))
(OR (AND (EQ rhmotor1 down) (> rhset rhpos))
    (AND (EQ rhmotor1 up) (< rhset rhpos))
    (AND (EQ rhmotor1 stop) (EQ rhset rhpos)))
(OR (AND (EQ bmotor1 backward) (> bset bpos))
    (AND (EQ bmotor1 forward) (< bset bpos))
    (AND (EQ bmotor1 stop) (EQ bset bpos)))
(OR (AND (EQ fhmotor1 down) (> fhset fhpos))
    (AND (EQ fhmotor1 up) (< fhset fhpos))
    (AND (EQ fhmotor1 stop) (EQ fhset fhpos)))
(OR (AND (EQ hrmotor1 down) (> hrset hrpos))
    (AND (EQ hrmotor1 up) (< hrset hrpos))
    (AND (EQ hrmotor1 stop) (EQ hrset hrpos)))
```

2. Input operation *memory* based on several sub operations which already exists by using type expression OP.

OP takes the name and the predicate representation of an operation as its parameters. Denoting the nesting tables with it makes the tabular structure more explicit. The trick is that we leave the detailed specification of sub operation out when we care only about its function in an overall operation. For $OP(n, f)$, inputting $(OP\ n\ f)$, our program will print n to standard outputs, but f to a file which will be verified by Simplify.

3. Call function *wpt b p c* with parameter *b = allstop*, *c = alltaset*, and *p = memory*.

The code of the proof implemented in OCaml is as follows:

```
let memory = reader ("BEGTAB\LHEADER\button=m1_$\button=m2_//
  laset=lam1_\&\rhset=rhm1_\&\bset=bm1_\&\fhset=fhm1_\&\hrset=hrm1_\>_
  (OP_latoset_\&^ deflatoset ^")_\&\(laset=lapos_\>_(OP_rhtoset_\&^
  defrhtoset ^"))_\&\(OP_btoset_\&^ defbtoset ^")_\&\(bset=bpos_\>
  (OP_fhtoset_\&^ deffhtoset ^"))_\&\(bset=bpos_\&\fhset=fhpos_\>
  (OP_hrtoset_\&^ defhrtoset ^"))_//_laset=lam2_\&\rhset=rhm2_\&
  bset=bm2_\&\fhset=fhm2_\&\hrset=hrm2_\>_(OP_latoset_\&^ deflatoset ^
  ")_\&\(laset=lapos_\>_(OP_rhtoset_\&^ defrhtoset ^"))_\&\(OP_btoset_\&^
  defbtoset ^")_\&\(bset=bpos_\>_(OP_fhtoset_\&^ deffhtoset ^"))_\&\(bset
  =bpos_\&\fhset=fhpos_\>_(OP_hrtoset_\&^ defhrtoset ^"))_//_ENDTAB")
in let p = ST(memory,[reader "lamotor";reader "rhmotor";
  reader "bmotor";reader "fhmotor";reader "hrmotor"])
in let b = reader "laset=_\lapos_\&\rhset=_\rhpos_\&\bset=_\bpos_\&
  fhset=_\fhpos_\&\hrset=hrpos"
in let c = reader "lamotor=stop_\&\rhmotor=stop_\&\bmotor=stop_\&
  fhmotor=stop_\&\hrmotor=stop"
in if (wpt b p c) then begin
  ...print memory to screen and a latex file...
  ...inform successful verification of memory weakest precondition...
end
```

6.4.2 Memory Set Mode

Initially, the positions stored in memory 1 and 2 are both set to motor home positions—0. *Memory Set* mode is functioned by a set of assignment statements. To set memory 1 or 2 to the current positions of motors, the states of all motors should be stop. This guard condition is in the upper header of Memory Set table. There are

no properties deriving from this table. Variables in this mode are the same as those in memory mode.

| | | |
|-------------|--|---|
| | $\text{lamotor} = \text{stop} \wedge \text{rhmotor} =$ $\text{stop} \wedge \text{bmotor} = \text{stop} \wedge$ $\text{fhmotor} = \text{stop} \wedge \text{hrmotor} =$ stop | $\text{lamotor} \neq \text{stop} \vee \text{rhmotor} \neq$ $\text{stop} \vee \text{bmotor} \neq \text{stop} \vee$ $\text{fhmotor} \neq \text{stop} \vee \text{hrmotor} \neq$ stop |
| button = m1 | $\text{lam11} = \text{lapos} \wedge \text{rhm11} =$ $\text{rhpos} \wedge \text{bm11} = \text{bpos} \wedge \text{fhm11}$ $= \text{fhpos} \wedge \text{hrm11} = \text{hrpos}$ | $\text{lam11} = \text{lam1} \wedge \text{rhm11} = \text{rhm1}$ $\wedge \text{bm11} = \text{bm1} \wedge \text{fhm11} = \text{fhm1}$ $\wedge \text{hrm11} = \text{hrm1}$ |
| button = m2 | $\text{lam21} = \text{lapos} \wedge \text{rhm21} =$ $\text{rhpos} \wedge \text{bm21} = \text{bpos} \wedge \text{fhm21}$ $= \text{fhpos} \wedge \text{hrm21} = \text{hrpos}$ | $\text{lam21} = \text{lam2} \wedge \text{rhm21} = \text{rhm2}$ $\wedge \text{bm21} = \text{bm2} \wedge \text{fhm21} = \text{fhm2}$ $\wedge \text{hrm21} = \text{hrm2}$ |

Table 6.13: Memory Set Mode

6.5 Calibration Mode and Its Properties

When a seat is calibrated, all motors move in a way similar to that in Memory mode. But LA and B can not move backward; RH, FH, and HR can not move down since the home positions are the front stops for LA and B and the upper stops for RH, FH, and HR. For instance, the single LA movement to its home position *latohome* can be written as follows:

| | |
|---------------------------------|------------------------------------|
| $\text{lapos} = \text{lahome}$ | $\text{lapos} \neq \text{lahome}$ |
| $\text{lamotor1} = \text{stop}$ | $\text{lamotor1} = \text{forward}$ |

Table 6.14: LA Motor Moving to Its Home *latohome*

Its definition in Ocaml programming language is:

```
let deflathome = "BEGTAB_UHEADER_lapos=_lahome$_lapos/=_lahome_//
    lamotor1=_stop$_lamotor1=_forward_//_ENDTAB" in
let lathome = reader deflathome in

...print lathome to screen and a latex file...
```

Other motor movement to their home position *rthome*, *bthome*, *fthome*, and *hthome* are defined similarly.

| | |
|-----------------|---------------------|
| rhpos = rhhome | rhpos \neq rhhome |
| rhmotor1 = stop | rhmotor1 = up |

Table 6.15: RH Motor Moving to Its Home *rthome*

```
let defrthome = "BEGTAB_UHEADER_rhpos=_rhhome$_rhpos/=_rhhome_//
    rhmotor1=_stop$_rhmotor1=_up_//_ENDTAB" in
let rthome = reader defrthome in

...print rthome to screen and a latex file...
```

| | |
|----------------|-------------------|
| bpos = bhome | bpos \neq bhome |
| bmotor1 = stop | bmotor1 = forward |

Table 6.16: B Motor Moving to Its Home *bthome*

```
let defbthome = "BEGTAB_UHEADER_bpos=_bhome$_bpos/=_bhome_//
    bmotor1=_stop$_bmotor1=_forward_//_ENDTAB" in
let bthome = reader defbthome in

...print bthome to screen and a latex file...
```

| | |
|-------------------|---------------------|
| $fhpos = fhhome$ | $fhpos \neq fhhome$ |
| $fhmotor1 = stop$ | $fhmotor1 = up$ |

Table 6.17: FH Motor Moving to Its Home *fthome*

```
let defffthome = "BEGTAB_UHEADER_fhpos=_fhhome_$fhpos/=fhhome//
    fhmotor1=_stop_$fhmotor1=_up_//_ENDTAB" in
let fthome = reader defffthome in
...print fthome to screen and a latex file...
```

| | |
|-------------------|---------------------|
| $hrpos = hrhome$ | $hrpos \neq hrhome$ |
| $hrmotor1 = stop$ | $hrmotor1 = up$ |

Table 6.18: HR Motor Moving to Its Home *hrthome*

```
let defhrthome = "BEGTAB_UHEADER_hrpos=_hrhome_$hrpos/=hrhome//
    hrmotor1=_stop_$hrmotor1=_up_//_ENDTAB" in
let hrthome = reader defhrthome in
...print hrthome to screen and a latex file...
```

The calibration consists of five sub operations *lathome*, *rthome*, *bthome*, *fthome*, and *hrthome*. The relation among them named *calibrate* embody the priority of motor movements:

$$lathome \wedge (lahome = lapos \implies rthome) \wedge bthome \wedge (bhome = bpos \implies fthome) \wedge (bhome = bpos \wedge fhhome = fhpos \implies hrthome).$$

We observe that during calibration, all motors are stopped if they are in their respective home positions. We state it formally as *pre* :

$$lahome = lapos \wedge rhhome = rhpos \wedge bhome = bpos \wedge fhhome = fhpos \wedge$$

$hrhome = hrpos$.

pre is the weakest precondition of program *calibrate* to establish postcondition po :

$lamotor = stop \wedge rhmotor = stop \wedge bmotor = stop \wedge fhmotor = stop \wedge$

$hrmotor = stop$.

We assume all cases in the sub-operations of calibration mode are considered for proving our weakest precondition of program *calibrate* to establish our postcondition. Tables (*latohome*, *rthohome*, *bthohome*, *fthohome* and *hrthohome*) are supposed to be total, which is set as axioms:

```
(OR (AND (EQ lahome lapos) (EQ lamotor1 stop))
    (AND (NEQ lahome lapos) (EQ lamotor1 forward)))
(OR (AND (EQ rhhome rhpos) (EQ rhmotor1 stop))
    (AND (NEQ rhhome rhpos) (EQ rhmotor1 up)))
(OR (AND (EQ bhome bpos) (EQ bmotor1 stop))
    (AND (NEQ bhome bpos) (EQ bmotor1 forward)))
(OR (AND (EQ fhhome fhpos) (EQ fhmotor1 stop))
    (AND (NEQ fhhome fhpos) (EQ fhmotor1 up)))
(OR (AND (EQ hrhome hrpos) (EQ hrmotor1 stop))
    (AND (NEQ hrhome hrpos) (EQ hrmotor1 up)))
```

Then, we call function $wp\ b\ p\ c$ with $b = pre$, $c = po$, and $p = calibrate$ to justify that b is the weakest precondition of operation p in form of plain predicate to establish postcondition po .

```
let calibrate = reader ("(OP_⊔latohome_⊔" ^ deflatohome ^ ")_⊔&_⊔
    (lahome=lapos_⊔=>_⊔(OP_⊔rthohome_⊔" ^ defrthohome ^ ")_⊔&_⊔(OP_⊔bthohome
    " ^ defbthohome ^ ")_⊔&_⊔(bhome=bpos_⊔=>_⊔(OP_⊔fthohome_⊔" ^ defffthohome ^ ")_⊔
    &_⊔(bhome=bpos_⊔&_⊔fhhome=fhpos_⊔=>_⊔(OP_⊔hrthohome_⊔" ^ defhrthohome ^ ")_⊔)")
in let p = ST(calibrate,[reader "lamotor";reader "rhmotor";
```

```

    reader "bmotor";reader "fhmotor";reader "hrmotor"]])
in let b = reader "lahome_=_lapos_&_rhhome=rhpos_&
    bhome=bpos_&_fhome=fhpos_&_hrhome=hrpos"
in let c = reader "lamotor=stop_&_rhmotor=stop_&
    bmotor=stop_&_fhmotor=stop_&_hrmotor=stop"
in if (wp b p c) then begin
    ...print calibrate to screen and a latex file...
    ...inform successful verification of calibrate weakest precondition...

```

6.6 Summary

The result of proving properties for the first three modes are shown in Figure 6.2. All (sub-) operations and their related motors are listed in columns "(Sub) Operation" and "Related Motors" respectively. In column "Proof Condition", we list the number of proof obligations generated for each operations by applying the theorem given in column "Theorems Applied". The length of proof predicates are listed in column "Size"; when a proof predicate relate to tabular operations, the structure of the tables are also given. In column "Time/Tabular", we give the user times that Simplify needs for proofs of properties of specifications in tabular form; in column "Time/Plain", we give the user times that Simplify needs for proofs of properties of the same specifications as plain predicates. Note that user time of proof related to *memory*, which is the largest specification in this example, is less than the user time of proof related to *memory* as plain predicate.

Our car seat example is suited for both statecharts and tables. We formalize the example, come up with a specification in tabular form and proved properties of the specification. Weakest preconditions are determined for single operations. System invariants are checked by showing that the initialization establishes them and that

| Mode | (Sub) Operation | Related Motors | Proof Condition | Size | Time(msec) | | Theorems Applied |
|-------------|-----------------|-------------------|-----------------|---------------------|------------|-------|---|
| | | | | | Tabular | Plain | |
| Calibration | (latohome) | LA | | 2rows,2cols | | | Weakest precondition |
| | (rthome) | RH | | 2rows,2cols | | | |
| | (btohome) | B | | 2rows,2cols | | | |
| | (fthtohome) | FH | | 2rows,2cols | | | |
| | (hrthtohome) | HR | | 2rows,2cols | | | |
| | calibrate | LA,RH,B ,FH,HR | 1 | 221c | | 54 | |
| Normal | lapressed | LA | 6 | 3rows,4cols 253c | 7 | 2 | Tabular Verification with Predicates |
| | rhpressed | RH | 9 | 4rows,4cols 375c | 9 | 4 | |
| | bpressed | B | 6 | 3rows,4cols 231c | 9 | 2 | |
| | fhpressed | FH | 9 | 4rows,4cols 372c | 10 | 5 | |
| | hrpressed | HR | 9 | 4rows,4cols 411c | 10 | 5 | |
| Memory | (latoset) | LA | | 2rows,3cols | | | Weakest Precondition with Predicates |
| | (rhtoset) | RH | | 2rows,3cols | | | |
| | (btoset) | B | | 2rows,3cols | | | |
| | (fhtoset) | FH | | 2rows,3cols | | | |
| | (hrtoset) | HR | | 2rows,3cols | | | |
| | memory | LA,RH,B ,FH,HR | 2 | 2rows,2cols 422c | 86 | 131 | |

Figure 6.2: Performance of Proof in Car Seat Control

all operations preserve them. Nesting operation structure is expressed by replacing internal operation predicates with their names. Most priorities are considered within tables except priorities of movements among different modes (e.g. Motor movements in Memory mode have higher priorities than those in Normal mode). A table is illustrated for each type of operation. We only introduce in detail group one movements for each mode. Tabular specifications for group two movements are derived in a similar way.

Chapter 7

Modeling a Visitor Information System

7.1 Example Introduction

A real life example is a visitor information system for managing a conference site [34]. Visitors come to a site to attend meetings. Each meeting is required to take place in a designated conference room, at a certain day. A meeting may require the use of a dining room for lunch. Booking a dining room requires lunch information, including the number of places needed. Several constraints have to be observed:

1. A conference room can host only one meeting.
2. A meeting may need more than one conference rooms.
3. All participants of a meeting take lunch in the same dining room.
4. Participants from several meetings can occupy the same dining room.
5. A visitor can attend only one meeting.

6. A meeting may involve several visitors.

At a first step in modeling, we have to find the different kinds of objects of concern.

Usually they appear as nouns in natural language descriptions.

1. Visitor
2. DiningRoom
3. ConferenceRoom
4. Meeting

We assume that each visitor, dining room, conference room, and meeting has a unique name, say given as a string. There are a different set of meetings taking place and a different set of visitors attending those. Hence we define two variables.

| | |
|--|----------------------------------|
| <i>visitors</i> : set of <i>Visitor</i> | the set of registered visitors |
| <i>meetings</i> : set of <i>Meeting</i> | the set of meetings taking place |

As the next step, we state the relationships between the various objects. Each registered visitor attends a meeting and only one meeting. Hence we define *attends* as a mapping from registered visitors to the meetings taking place, a mapping is a set $F \subseteq I \times O$ such that if $(x, y), (x, y') \in F$, then $y = y'$:

attends: **map** *Visitor* to *Meeting*
dom *attends* = *visitors*
ran *attends* \subseteq *meetings*

The last two lines express that every visitor must register to a meeting, but there can be a meeting without any visitors(yet).

Each meeting occupies at least one conference room, but no conference rooms can be shared between meetings. We model this as an injective relation between meetings and conference rooms:

convenes: **rel** *Meetings* **to** *ConferenceRoom*

dom *convenes* = *meetings*

injective(*convenes*)

The last line expresses that each meeting must take place in one conference room.

Each meeting takes lunch in a dining room, if it requires lunch at all. However, several meetings can share a dining room. We express this as a mapping between meetings taking place and dining rooms:

eats: **map** *Meeting* **to** *DiningRoom*

dom *eats* \subseteq *meetings*

If we use $(attends \circ eats)^{-1} = eats^{-1} \circ attends^{-1}$ to relate each dining room to all the visitors eating in there, we may consider the implicit requirement that the total number of visitors eating in a dining room must not exceed capacity of the dining room. Let

capacity: **map** *DiningRoom* **to** integer

total(*capacity*)

be a function returning the maximal capacity of each dining room, where $\forall dr \cdot capacity(dr) > 0$. Then we may add:

$$\forall dr \in \mathbf{ran} \text{ eats} \cdot \#((attends \circ eats)^{-1}\{dr\}) \leq capacity(dr)$$

Since in Simplify the *universal quantifier* and the *existential quantifier* are assumed to have the range of integers, in our implementation we can reformulate above predicate by limiting *dr* to type **ran eats**

$$dr \in \mathbf{ran} \text{ eats} \implies \#((attends \circ eats)^{-1}\{dr\}) \leq capacity(dr).$$

Based on the above data structure, the system has to support following operations [31]:

1. *createMeeting*: create a new meeting

2. *cancelMeeting*: delete meeting, provided no dining room, conference rooms, and visitors are associated with that meeting.
3. *cancelMeetingArrangement*: delete meeting with all associated rooms and visitors.
4. *enterVisitor*: create a new visitor entry.
5. *removeVisitor*: remove visitor from the system.
6. *addVisitorToMeeting,removeVisitorFromMeeting*: as the name says.
7. *bookDiningRoom,cancelDiningRoom*: for a particular meeting
8. *bookConferenceRoom,cancelConferenceRoom*: for a particular meeting.

7.2 Specification of Visitor Information System

Invariants have to hold throughout the operations to maintain the data structure of function for variable *attends* and *eats*, and injective relation for variable meetings. Besides invariants, we solve the *Completeness* issue for each procedure: Does the specification cover all possible cases, or did we forget some cases? Completeness often implies (some sort of) definedness which is expressed by preconditions. Assertion statements are served as precondition of the procedure. For a program segment with alternative statements, precondition of it is the coverage of the condition. We arrive at this functional specification from a holistic point of view, considering the whole system.

module *VisitorInformationSystem*

var *visitors*: **set of** *String* (* **set of** Visitor *)

```

var meetings: set of String          (* set of Meeting *)
var attends: map String to String    (* map Visitor to Meeting *)
var convenes: rel String to String  (* rel Meeting to ConferenceRoom *)
var eats: map String to String      (* map Meeting to DiningRoom *)

{invariant:(dom attends  $\subseteq$  visitors)  $\wedge$  (ran attends  $\subseteq$  meetings)  $\wedge$ 
(dom convenes  $\subseteq$  meetings)  $\wedge$  map(attends)  $\wedge$  injective(convenes)  $\wedge$ 
map(eats)  $\wedge$  (dom eats  $\subseteq$  meetings)  $\wedge$ 
( $\forall dr \in$  ran eats  $\cdot \#((attends \circ eats)^{-1}\{dr\}) \leq capacity(dr))$ }

public procedure createMeeting(val m : String)
    m  $\neq$  nil  $\wedge$  m  $\neq$  ""  $\wedge$  m  $\notin$  meetings  $\wedge$  meetings1 = meetings  $\cup$  {m}

public procedure cancelMeeting(val m : String)
    m  $\in$  meetings  $\wedge$  m  $\notin$  ran(attends)  $\wedge$  m  $\notin$  dom(convenes)  $\wedge$  m  $\notin$  dom(eats)  $\wedge$ 
    meetings1 = meetings – {m}

public procedure cancelMeetingArrangement(val m : String)
    m  $\in$  meetings  $\wedge$  meetings1 = meetings – {m}  $\wedge$ 
    attends1 = attends – revrelate(m, attends)  $\wedge$ 
    convenes1 = convenes – relate(m, convenes)  $\wedge$ 
    eats1 = eats – relate(m, eats)

public procedure enterVisitor(val v : String)
    v  $\neq$  nil  $\wedge$  v  $\neq$  ""  $\wedge$  v  $\notin$  visitors  $\wedge$  visitors1 = visitors  $\cup$  {v}

public procedure removeVisitor(val v : String)
    v  $\in$  visitors  $\wedge$  v  $\notin$  dom(attends)  $\wedge$  visitors1 = visitors – {v}

public procedure addVisitorToMeeting(val v : String, val m : String)
    v  $\in$  visitors  $\wedge$  m  $\in$  meetings  $\wedge$  v  $\notin$  dom(attends)  $\wedge$  attends1 = attends  $\cup$ 

```

$$\{(v, m)\} \wedge (\neg(dr \in \text{ran}(\text{eats}) \implies \text{card}(\text{revrelate}(dr, \text{attends1} \circ \text{eats})) \leq \text{capacity}(dr)) \wedge \text{eats1} = \text{eats} - \text{relate}(m, \text{eats}) \vee (dr \in \text{ran}(\text{eats}) \implies \text{card}(\text{revrelate}(dr, \text{attends1} \circ \text{eats})) \leq \text{capacity}(dr)) \wedge \text{eats1} = \text{eats})$$

public procedure *removeVisitorFromMeeting*(**val** *v* : *String*)

$$v \in \text{dom}(\text{attends}) \wedge \text{attends1} = \text{attends} - \text{relate}(v, \text{attends})$$

public procedure *visitorInfo*(**val** *v* : *String*, **res** *mt* : *String*)

| | | |
|--|--|----------------------------|
| $v \in \text{visitors} \wedge v \in \text{dom}(\text{attends})$ | $v \in \text{visitors} \wedge v \notin \text{dom}(\text{attends})$ | $v \notin \text{visitors}$ |
| $\text{mt1} = \text{concat}(\text{ran}(\text{relate}(v, \text{attends})))$ | $\text{mt1} = ""$ | $\text{mt1} = \text{nil}$ |

Table 7.1: Checking Meeting Attended *visitorInfo*

public procedure *bookDiningRoom*(**val** *m* : *String*, **val** *d* : *String*)

$$m \in \text{meetings} \wedge d \neq \text{nil} \wedge m \notin \text{dom}(\text{eats}) \wedge (dr \in \text{ran}(\text{eats} \cup \{(m, d)\}) \implies \text{card}(\text{revrelate}(dr, \text{compose}(\text{attends}, \text{eats} \cup \{(m, d)\}))) \leq \text{capacity}(dr)) \wedge \text{eats1} = \text{eats} \cup \{(m, d)\}$$

public procedure *cancelDiningRoom*(**val** *m* : *String*)

$$m \in \text{dom}(\text{eats}) \wedge \text{eats1} = \text{eats} - \text{relate}(m, \text{eats})$$

public procedure *bookConferenceRoom*(**val** *m* : *String*, **val** *c* : *String*)

$$m \in \text{meetings} \wedge c \neq \text{nil} \wedge c \notin \text{ran}(\text{convenes}) \wedge \text{convenes1} = \text{convenes} \cup \{(m, c)\}$$

public procedure *cancelConferenceRoom*(**val** *c* : *String*)

$$c \in \text{ran}(\text{convenes}) \wedge \text{convenes1} = \text{convenes} - \text{revrelate}(c, \text{convenes})$$

public procedure *conferenceRooms*(**val** *m* : *String*, **res** *cr* : *String*)

| $m \in \text{meetings} \wedge m \in \text{dom}(\text{convenes})$ | $m \in \text{meetings} \wedge m \notin \text{dom}(\text{convenes})$ | $m \notin \text{meetings}$ |
|---|---|----------------------------|
| <code>cr1 = concat(<i>ran</i>(<i>relate</i>(<i>m</i>, <i>convenes</i>)))</code> | <code>cr1 = ""</code> | <code>cr1 = nil</code> |

Table 7.2: Conference Room *conferenceRooms*

public procedure *diningRooms*(**val** *m* : *String*, **res** *dr* : *String*)

| $m \in \text{meetings} \wedge m \in \text{dom}(\text{eats})$ | $m \in \text{meetings} \wedge m \notin \text{dom}(\text{eats})$ | $m \notin \text{meetings}$ |
|--|---|----------------------------|
| <code>d1 = concat(<i>ran</i>(<i>relate</i>(<i>m</i>, <i>eats</i>)))</code> | <code>d1 = ""</code> | <code>d1 = nil</code> |

Table 7.3: Dining Room *diningRooms*

begin *visitors*, *meetings*, *attends*, *convenes*, *eats* := ϕ , ϕ , ϕ , ϕ , ϕ
end

7.3 Axiom

Since Simplify only accepts a sequence of first order formulae as input, the properties and operations of set, relation and function are not acceptable directly. But Simplify proves its formulae assuming some set of axioms, we can define properties and operations in terms of functions which are called axioms and customize the axiom set in Simplify before validating each predicate. There are a large amount of axioms for set, relation and function properties and operations. A few of them are applied in this example to prove our stated invariants. Axioms are presented in a style which Simplify can recognize. Their explanations in the combination of alternative statement (to limit the definedness of an axiom), conventional symbol and first order logic are also provided following each of them.

1. MEMBER 1

```
(FORALL (x y s)
  (PATS (MEMBER x (INSERT s y)))
  (IMPLIES (NEQ x y) (EQ (MEMBER x (INSERT s y)) (MEMBER x s)))))
```

if $x \neq y$ then $(x \in s \cup \{y\} \Leftrightarrow x \in s) \wedge (x \notin s \cup \{y\} \Leftrightarrow x \notin s)$

2. MEMBER 2

```
(FORALL (x xs ys)
  (PATS (MEMBER x (DELETE xs ys)))
  (IMPLIES (NEQ (MEMBER x xs) true)
    (NEQ (MEMBER x (DELETE xs ys)) true))))
```

if $x \notin xs$ then $x \notin xs - ys$

3. MEMBER 3

```
(FORALL (x xs ys)
  (PATS (MEMBER x (dom (DELETE xs ys))))
  (IMPLIES (NEQ (MEMBER x (dom xs)) true)
    (NEQ (MEMBER x (dom (DELETE xs ys))) true)))
```

if $x \notin \text{dom } xs$ then $x \notin \text{dom } (xs - ys)$

4. MEMBER 4

```
(FORALL (x xs ys)
  (PATS (MEMBER x (ran (DELETE xs ys))))
  (IMPLIES (NEQ (MEMBER x (ran xs)) true)
    (NEQ (MEMBER x (ran (DELETE xs ys))) true)))
```

if $x \notin \text{ran } xs$ then $x \notin \text{ran } (xs - ys)$

5. SUBSET 1

```
(FORALL (y xs ys)
  (PATS (SUBSET xs (INSERT ys y)))
  (IMPLIES (EQ (SUBSET xs ys) true)
    (EQ (SUBSET xs (INSERT ys y)) true)))
```

if $xs \subseteq ys$ then $xs \subseteq ys \cup \{y\}$

6. SUBSET 2

```
(FORALL (x xs ys)
  (PATS (SUBSET xs (DELETE ys (INSERT EMPTY x)))))
```


(IMPLIES (AND (NEQ (MEMBER x xs) true) (EQ (SUBSET xs ys) true))
(EQ (SUBSET xs (DELETE ys (INSERT EMPTY x))) true)))

if $x \notin xs \wedge xs \subseteq ys$ then $xs \subseteq ys - \{x\}$

7. SUBSET 3

(FORALL (xs ys zs)
(PATS (SUBSET (dom (DELETE xs zs)) ys))
(IMPLIES (EQ (SUBSET (dom xs) ys) true)
(EQ (SUBSET (dom (DELETE xs zs)) ys) true)))

if dom $xs \subseteq ys$ then dom $(xs - zs) \subseteq ys$

8. SUBSET 4

(FORALL (xs ys zs)
(PATS (SUBSET (ran (DELETE xs zs)) ys))
(IMPLIES (EQ (SUBSET (ran xs) ys) true)
(EQ (SUBSET (ran (DELETE xs zs)) ys) true)))

if ran $xs \subseteq ys$ then ran $(xs - zs) \subseteq ys$

9. SUBSET 5

(FORALL (x y xs ys)
(PATS (SUBSET (dom (INSERT ys (PAIR x y))) xs))
(IMPLIES (AND (EQ (MEMBER x xs) true)
(EQ (SUBSET (dom ys) xs) true))
(EQ (SUBSET (dom (INSERT ys (PAIR x y))) xs) true)))

if $x \in xs \wedge \text{dom } ys \subseteq xs$ **then** $\text{dom } (ys \cup (x, y)) \subseteq xs$

10. SUBSET 6

```
(FORALL (x y xs ys)
  (PATS (SUBSET (ran (INSERT ys (PAIR x y))) xs))
  (IMPLIES (AND (EQ (MEMBER y xs) true)
    (EQ (SUBSET (ran ys) xs) true))
    (EQ (SUBSET (ran (INSERT ys (PAIR x y))) xs) true)))
```

if $y \in xs \wedge \text{ran } ys \subseteq xs$ **then** $\text{ran } (ys \cup (x, y)) \subseteq xs$

11. UNION 1

```
(FORALL (xs)
  (PATS (UNION xs EMPTY))
  (EQ (UNION xs EMPTY) xs))
```

$$xs \cup \phi = xs$$

12. UNION 2

```
(FORALL (xs ys y)
  (PATS (UNION xs (INSERT ys y)))
  (EQ (UNION xs (INSERT ys y)) (INSERT (UNION xs ys) y)))
```

$$xs \cup (ys \cup \{y\}) = (xs \cup ys) \cup \{y\}$$

13. DELETE 1

```
(FORALL (x xs)
  (NEQ (MEMBER x (dom (DELETE xs (relate x xs)))) true))
```

$$x \notin \mathbf{dom} (xs - |\{x\}|)$$

14. DELETE 2

```
(FORALL (x xs)
  (NEQ (MEMBER x (ran (DELETE xs (revrelate x xs)))) true))
```

$$x \notin \mathbf{ran} (xs - |\{x\}|^{-1})$$

15. DOMAIN 1

```
(FORALL (xs x1 x2)
  (PATS (dom (INSERT xs (PAIR x1 x2))))
  (EQ (dom (INSERT xs (PAIR x1 x2))) (INSERT (dom xs) x1)))
```

$$\mathbf{dom} (xs \cup (x_1, x_2)) = \mathbf{dom} xs \cup \{x_1\}$$

16. RANGE 1

```
(FORALL (xs x1 x2)
  (PATS (ran (INSERT xs (PAIR x1 x2))))
  (EQ (ran (INSERT xs (PAIR x1 x2))) (INSERT (ran xs) x2)))
```

$$\mathbf{ran} (xs \cup (x_1, x_2)) = \mathbf{ran} xs \cup \{x_2\}$$

17. INJECTIVE 1

```
(FORALL (xs ys)
  (PATS (injective (DELETE xs ys)))
  (IMPLIES (EQ (injective xs) true)
    (EQ (injective (DELETE xs ys)) true)))
```

if injective xs then injective $(xs - ys)$

18. INJECTIVE 2

```
(FORALL (x x1 xs)
  (PATS (injective (INSERT xs (PAIR x x1))))
  (IMPLIES (AND (EQ (injective xs) true)
    (NEQ (MEMBER x1 (ran xs)) true))
    (EQ (injective (INSERT xs (PAIR x x1))) true)))
```

if (injective xs) \wedge ($x_1 \in \text{ran } xs$) then injective $(xs \cup \{(x, x_1)\})$

19. MAP 1

```
(FORALL (xs ys)
  (PATS (map (DELETE xs ys)))
  (IMPLIES (EQ (map xs) true) (EQ (map (DELETE xs ys)) true)))
```

if map xs then map $(xs - ys)$

20. MAP 2

```
(FORALL (x x1 xs)
  (PATS (map (INSERT xs (PAIR x1 x))))
  (IMPLIES (AND (EQ (map xs) true)
    (NEQ (MEMBER x1 (dom xs)) true))
    (EQ (map (INSERT xs (PAIR x1 x))) true)))
```

if (map xs) \wedge ($x_1 \notin \text{dom } xs$) then map $(xs \cup (x_1, x))$

21. CAPACITY 1

```

(FORALL (y xs ys m v)
  (IMPLIES (EQ (MEMBER y (ran (DELETE ys (relate m ys)))) true)
    (<= (card (revrelate y
      (compose (UNION xs (INSERT EMPTY (PAIR v m)))
        (DELETE ys (relate m ys)))))
      (card (revrelate y (compose xs ys)))))
  
$$\forall y \in \mathbf{ran} \, ys \cdot \#(((xs \cup (v, m)) \circ (ys - |\{m\}|))^{-1}[\{y\}]) \leq \#((xs \circ ys)^{-1}[\{y\}])$$


```

22. CAPACITY 2

```

(FORALL (y xs ys zs)
  (IMPLIES (EQ (MEMBER y (ran ys)) true)
    (<= (card (revrelate y (compose (DELETE xs zs) ys))
      (card (revrelate y (compose xs ys)))))
  
$$\forall y \in \mathbf{ran} \, ys \cdot \#(((xs - zs) \circ ys)^{-1}[\{y\}]) \leq \#((xs \circ ys)^{-1}[\{y\}])$$


```

23. CAPACITY 3

```

(FORALL (y xs ys zs)
  (IMPLIES (EQ (MEMBER y (ran ys)) true)
    (<= (card (revrelate y (compose xs (DELETE ys zs)))
      (card (revrelate y (compose xs ys)))))
  
$$\forall y \in \mathbf{ran} \, ys \cdot \#((xs \circ (ys - zs))^{-1}[\{y\}]) \leq \#((xs \circ ys)^{-1}[\{y\}])$$


```

7.4 Proving Invariants and Preconditions

Like the formal proof in terms of implementation in our CarSeat example, we prove preconditions and the system invariant preserved for each procedure in this

information system based on predicate. Invariant and preconditions are already predicate except that set and relation properties and operations should be transformed to self-defined functions.

The invariant vi of the system can be represented in predicate by our input notation:

```
dom(attendes)<:visitors & ran(attendes)<:meetings &
dom(convenes)<:meetings & map(attendes)=true & map(eates)=true &
injective(convenes)=true & dom(eates) <:meetings &
(dr:ran(eates)=>card(revrelate(dr,compose(attendes,eates)))<=capacity(dr))
```

The proof of invariant for each procedure is calling function tpv (or vp) $b p c$ where procedure $p = ST(op, vl)$ and op is a predicate in tabular (plain) form, $b = c = vi$. Preconditions are validated by calling function $pret$ (or pre) $b p$ where parameters b and p are the same as those of tpv (or vp). We include the source code of the proof in the following subsections.

```
public procedure createMeeting(val m : String)

let createMeeting = reader "m/=nil_&_m/=EMPSTR_&_m/:_meetings_&
    meetings1=meetings_&_m}"
in let p = ST(createMeeting, [reader "meetings"])
in let b = reader "m/=nil_&_m/=EMPSTR_&_m/:_meetings"
in if (pre b p) ^ (vp bi p ci) then begin
    ...print createMeeting to screen and a latex file...
    ...print b as precondition of createMeeting...
    ...print bi as system invariant preserved by createMeeting...
end

public procedure cancelMeeting(val m : String)

let cancelMeeting = reader "m:meetings_&_m/:ran(attendes)_&
```

```

    m/:dom(convenes)⊔&m/:dom(eats)⊔meetings1=meetings--{m}"
in let p = ST(cancelMeeting,[reader "meetings"])
in let b = reader "m:meetings⊔&m/:ran(attendes)⊔&
m/:dom(convenes)⊔&m/:dom(eats)"
in if (pre b p) ∧ (vp bi p ci) then begin
    ...print cancelMeeting to screen and a latex file...
    ...print b as precondition of cancelMeeting...
    ...print bi as system invariant preserved by cancelMeeting...
end

public procedure cancelMeetingArrangement(val m : String)

let cancelMeetingArrangement = reader "m:meetings⊔&
meetings1=meetings--{m}⊔&attendes1=attendes--revrelate(m,attendes)⊔&
convenes1=convenes--relate(m,convenes)&eats1=eats--relate(m,eats)"
in let p = ST(cancelMeetingArrangement,
[reader "meetings";reader "attendes";reader "convenes";reader "eats"])
in let b = reader "m:meetings"
in if (pre b p) ∧ (vp bi p ci) then begin
    ...print createMeetingArrangement to screen and a latex file...
    ...print b as precondition of createMeetingArrangement...
    ...print bi as system invariant preserved by createMeetingArrangement...
end

public procedure enterVisitor(val v : String)

let enterVisitor = reader
    "v/=nil⊔&v/=EMPSTR⊔&v/:⊔visitors⊔&visitors1=⊔visitors⊔U⊔{v}"
in let p = ST(enterVisitor,[reader "visitors"])
in let b = reader "v/=nil⊔&v/=EMPSTR⊔&v/:visitors"
in if (pre b p) ∧ (vp bi p ci) then begin
    ...print enterVisitor to screen and a latex file...
    ...print b as precondition of enterVisitor...

```

```

...print bi as system invariant preserved by enterVisitor...

end

public procedure removeVisitor(val v : String)

let removeVisitor = reader
  "v:visitors1&v/:dom(attends)&visitors1=visitors--{v}" in
in let p = ST(removeVisitor, [reader "visitors"])
in let b = reader "v:visitors1&v/:dom(attends)"
in if (pre b p) ∧ (vp bi p ci) then begin
  ...print removeVisitor to screen and a latex file...
  ...print b as precondition of removeVisitor...
  ...print bi as system invariant preserved by removeVisitor...
end

public procedure addVisitorToMeeting(val v : String, val m : String)

let addVisitorToMeeting = reader "v:visitors&m:meetings&v/:dom(attends)
  &attends1=attendsU{PAIR(v,m)}&(not(dr:ran(eats)=>
  card(revrelate(dr,compose(attends1,eats)))<=capacity(dr))
  &eats1=eats--relate(m,eats))or((dr:ran(eats)=>card(revrelate
  (dr,compose(attends1,eats)))<=capacity(dr))&eats1=eats))"
in let p = ST(addVisitorToMeeting, [reader "attends";reader "eats"])
in let b = reader "v:visitors1&m:meetings1&v/:dom(attends)"
in if (pre b p) ∧ (vp bi p ci) then begin
  ...print addVisitorToMeeting to screen and a latex file...
  ...print b as precondition of addVisitorToMeeting...
  ...print bi as system invariant preserved by addVisitorToMeeting...
end

public procedure removeVisitorFromMeeting(val v : String)

```



```

let removeVisitorFromMeeting = reader
  "v:dom(attendings)&attendings1=attendings--relate(v,attendings)"
in let p = ST(removeVisitorFromMeeting,[reader "attendings"])
in let b = reader "v:dom(attendings)"
in if (pre b p) ∧ (vp bi p ci) then begin
  ...print removeVisitorFromMeeting to screen and a latex file...
  ...print b as precondition of removeVisitorFromMeeting...
  ...print bi as system invariant preserved by removeVisitorFromMeeting...
end

```

```

public procedure visitorInfo(val v : String, res mt : String)

let visitorInfo = reader "BEGTAB_UHEADER_v:visitors_&v:dom(attendings)_$
  v:visitors_&v/:dom(attendings)_&v/:visitors_//_mt1_=
  concate(ran(relate(v,attendings)))_&_mt1=EMPSTR_&_mt1=nil//_ENDTAB"
let p = ST(visitorInfo,[reader "mt"])
in let b = reader "TRUE"
in if (pret b p) ∧ (tvp bi p ci) then begin
  ...print visitorInfo to screen and a latex file...
  ...print b as precondition of visitorInfo...
  ...print bi as system invariant preserved by visitorInfo...
end

```

```

public procedure bookDiningRoom(val m : String, val d : String)

let bookDiningRoom = reader "m:meetings_&d/=nil_&m/:dom(eats)&
  (dr:ran(eats_U_{PAIR(m,d)}))=>card(revrelate(dr,compose(attendings,
  eats_U_{PAIR(m,d)})))<=capacity(dr))&eats1=eats_U_{PAIR(m,d)}"
in let p = ST(bookDiningRoom,[reader "eats"])
in let b = reader "m:meetings_&d/=nil_&m/:dom(eats)_&
(dr:ran(eats_U_{PAIR(m,d)}))=>card(revrelate(dr,compose(attendings,

```

```

eats ⊔ U {PAIR(m,d)})) <= capacity(dr))"
in if (pre b p) ∧ (vp bi p ci) then begin
  ...print bookDiningRoom to screen and a latex file...
  ...print b as precondition of bookDiningRoom...
  ...print bi as system invariant preserved by bookDiningRoom...
end

public procedure cancelDiningRoom(val m : String)

let cancelDiningRoom = reader
  "m:dom(eats) ⊔ & eats1 ⊔ = ⊔ eats ⊔ -- ⊔ relate(m,eats)"
in let p = ST(cancelDiningRoom, [reader "eats"])
in let b = reader "m:dom(eats)"
in if (pre b p) ∧ (vp bi p ci) then begin
  ...print cancelDiningRoom to screen and a latex file...
  ...print b as precondition of cancelDiningRoom...
  ...print bi as system invariant preserved by cancelDiningRoom...
end

public procedure bookConferenceRoom(val m : String, val c : String)

let bookConferenceRoom = reader "m:meetings ⊔ & c/=nil ⊔ &
  c ⊔ /:ran(convenes) ⊔ & convenes1 ⊔ = ⊔ convenes ⊔ U {PAIR(m,c)}"
in let p = ST(bookConferenceRoom, [reader "convenes"])
in let b = reader "m:meetings ⊔ & c/=nil ⊔ & c/:ran(convenes)"
in if (pre b p) ∧ (vp bi p ci) then begin
  ...print bookConferenceRoom to screen and a latex file...
  ...print b as precondition of bookConferenceRoom...
  ...print bi as system invariant preserved by bookConferenceRoom...
end

public procedure cancelConferenceRoom(val c : String)

let cancelConferenceRoom = reader "c:ran(convenes) ⊔ &

```

```

    convenes1_ = _convenes_ -- _revrelate(c,convenes)"
in let p = ST(cancelConferenceRoom,[reader "convenes"])
in let b = reader "c:ran(convenes)"
in if (pre b p) ∧ (vp bi p ci) then begin
    ...print cancelConferenceRoom to screen and a latex file...
    ...print b as precondition of cancelConferenceRoom...
    ...print bi as system invariant preserved by cancelConferenceRoom...
end

public procedure conferenceRooms(val m : String, res cr : String)

let conferenceRooms = reader "BEGTAB_UHEADER_m:meetings_&
    m:dom(convenes)_$_m:meetings&m/:dom(convenes)_$_m/:meetings_//
    cr1=concat(ran(related(m,convenes)))$cr1=EMPSTR$cr1=nil//ENDTAB"
in let p = ST(conferenceRooms,[reader "cr"])
in let b = reader "TRUE"
in if (pret b p) ∧ (vp bi p ci) then begin
    ...print conferenceRooms to screen and a latex file...
    ...print b as precondition of conferenceRooms...
    ...print bi as system invariant preserved by conferenceRooms...
end

public procedure diningRooms(val m : String, res dr : String)

let diningRooms = reader "BEGTAB_UHEADER_m_m:meetings_&
    m:dom(eats)_$_m:meetings_&_m/:dom(eats)_$_m/:meetings_//
    d1=concat(ran(related(m,eats)))_$_d1=EMPSTR$_$_d1=nil//_ENDTAB"
in let p = ST(diningRooms,[reader "d"])
in let b = reader "TRUE"
in if (pret b p) ∧ (vp bi p ci) then begin
    ...print diningRooms to screen and a latex file...
    ...print b as precondition of diningRooms...
    ...print bi as system invariant preserved by diningRooms...

```

end

7.5 Conclusion

The result of our implementation is listed in Figure 7.1. We choose two operations, *createMeeting* and *visitorInfo* to demonstrate how to read our result. We use the theory of **precondition** to prove the completeness of *createMeeting*, which generate 1 proof obligation. Our proof predicate is 74 characters long. We use the theorem of **verification with predicates** to prove our invariant holds in the executing of *createMeeting*, which generate 1 proof obligation. The proof predicate of it contains 719 characters. The proofs related to operation *createMeeting* takes 9 millisecond. We use the theory of **precondition with tabular predicates** to prove the completeness of *visitorInfo*, which generate 3 obligations. We use the theorem of **tabular verification with predicates** to prove our invariant holds in the executing of *visitorInfo*, which generate 3 obligations. The proof predicates for the precondition and the invariant related to *visitorInfo* contain 117 and 783 characters respectively. The proofs related to tabular specification *visitorInfo* takes 7 millisecond. The proofs related to *visitorInfo* as plain predicate takes 4 millisecond.

The specification of the visitor information system is straightforward according to requirement except the procedure *addVisitorToMeeting*. It state that after a visitor is added to a meeting, if a dining room has been booked for the meeting and the total number of visitors eating in the dining room exceeds the capacity of the dining room, the booking should be canceled. Although Simplify supports relations, it only includes the ordering relations on integers. Simplify also include function definition, but in our case, domain and range can not be modeled by functions defined in Simplify.

| Operation | Proof Condition | Size | Time(msec) | | Theorems Applied |
|--------------------------|-----------------|--------------------------|------------|-------|---|
| | | | Tabular | Plain | |
| createMeeting | 1;1 | 74c;719c | | 9 | Precondition;Verification with Predicates |
| cancelMeeting | 1;1 | 134c;751c | | 10 | Precondition;Verification with Predicates |
| cancelMeetingArrangement | 1;1 | 150c;810c | | 11 | Precondition;Verification with Predicates |
| enterVisitor | 1;1 | 74c;719c | | 8 | Precondition;Verification with Predicates |
| removeVisitor | 1;1 | 78c;723c | | 51 | Precondition;Verification with Predicates |
| addVisitorToMeeting | 1;1 | 266c;900c | | 18 | Precondition;Verification with Predicates |
| removeVisitorFromMeeting | 1;1 | 68c;724c | | 9 | Precondition;Verification with Predicates |
| visitorInfo | 3;3 | 2rows,3cols 117c;783c | 7 | 4 | Precondition with Tabular Predicates; Tabular Verification with Predicates |
| bookDiningRoom | 1;1 | 260c;808c | | 16 | Precondition;Verification with Predicates |
| cancelDiningRoom | 1;1 | 53c;712c | | 9 | Precondition;Verification with Predicates |
| bookConferenceRoom | 1;1 | 96c;734c | | 8 | Precondition;Verification with Predicates |
| cancelConferenceRoom | 1;1 | 76c;731c | | 4 | Precondition;Verification with Predicates |
| conferenceRooms | 3;3 | 2rows,3cols 120c;786c | 7 | 4 | Precondition with Tabular Predicates; Tabular Verification with Predicates |
| diningRooms | 3;3 | 2rows,3cols 105c;771c | 8 | 4 | Precondition with Tabular Predicates; Tabular Verification with Predicates |

Figure 7.1: Performance of Proof in Visitor Information System

Therefore, we model our binary relations as sets of pairs and our function as a special binary relation. When modeling sets, relations and functions, different axioms must be built for different procedures. We pick exactly those axioms by generate and test method, check if each axiom is helpful in proving our stated properties.

Chapter 8

Elevator Control Refinement

8.1 Controlling Elevators

A typical passenger elevator will have [1]:

- General controls
 - Pressing call buttons to choose a floor.
 - Pressing door open and door close buttons to instruct the elevator to close immediately or remain open longer.
 - Controlling an alarm switch to signal that passengers have been trapped in the elevator.
 - Controlling the lights and ventilation fans switches in the elevator.
- Floor numbering, the numbering scheme used for a building's floors.
- Elevator scheduling
- Special operating modes

In this chapter, call button pressed operation is abstracted from a concrete specification, elevator scheduling is stepwise refined to execute the elevator algorithm.

8.2 Call Button Pressed Abstraction

8.2.1 Case Introduction

First, we illustrate an example on abstracting elevator call button pressed operation [32]. Integer variable *floor* stands for the current floor; variable *reqs* is a set of integers, for the floors to which requests exist; variable *mode* would take values *up*, *down*, *waiting* as the current direction of the elevator. Table 8.1 specifies the operation *buttonPressed* of requesting the elevator at floor *f*.

| | $\text{mode} = \text{waiting}$ | $\text{mode} \neq \text{waiting}$ |
|--------------------|---|--|
| $f > \text{floor}$ | $\text{reqs1} = \{f\} \wedge \text{mode1} = \text{up} \wedge$ $\text{floor1} = \text{floor}$ | $\text{reqs1} = \text{reqs} \cup \{f\} \wedge \text{mode1} =$ $\text{mode} \wedge \text{floor1} = \text{floor}$ |
| $f = \text{floor}$ | $\text{reqs1} = \{\} \wedge \text{mode1} = \text{waiting}$ $\wedge \text{floor1} = \text{floor}$ | $\text{reqs1} = \text{reqs} \wedge \text{mode1} = \text{mode}$ $\wedge \text{floor1} = \text{floor}$ |
| $f < \text{floor}$ | $\text{reqs1} = \{f\} \wedge \text{mode1} = \text{down} \wedge$ $\text{floor1} = \text{floor}$ | $\text{reqs1} = \text{reqs} \cup \{f\} \wedge \text{mode1} =$ $\text{mode} \wedge \text{floor1} = \text{floor}$ |

Consider applying decoding to the relation *buttonPressed* over variables *mode* and *reqs* as defined above, our intention is to abstract variable *reqs* with a Boolean variable *r* that only reflects if *reqs* is empty and to abstract variable *mode* with a Boolean variable *w* that only reflects whether *mode* is *waiting* or not. Thus this abstraction reduces the state space to two Boolean variables. A typical use of such an abstraction is to allow (automated) proofs about the abstraction, for example the

property that if there are no requests then the mode must be *waiting*. Formally our decoding relation is

$$RW(reqs, mode)(r, w) \equiv (r \equiv reqs \neq \{\}) \wedge (w \equiv mode = waiting)$$

8.2.2 Axioms

As stated before, any program variable belongs to an enumeration type or boolean type should be specified about its type. In this example, three program variable—*mode, r, w* have their types specified:

```
(DISTINCT up down waiting)
(OR (EQ mode waiting) (EQ mode down) (EQ mode up))
(OR (EQ mode1 waiting) (EQ mode1 down) (EQ mode1 up))
(DISTINCT true false)
(OR (EQ r true) (EQ r false))
(OR (EQ r1 true) (EQ r1 false))
(OR (EQ w true) (EQ w false))
(OR (EQ w1 true) (EQ w1 false))
```

Other properties of set used in this example are followed by their explanation:

- EMPTY

```
(FORALL (xs x)
  (PATS (INSERT xs x))
  (NEQ (INSERT xs x) EMPTY))
```

$$xs \cup \{x\} \neq \phi$$

- MEMBER


```
(FORALL (x s)
  (PATS (MEMBER x (INSERT s x)))
  (EQ (MEMBER x (INSERT s x)) true)))
```

$$x \in s \cup \{x\}$$

- DELETE

```
(FORALL (xs ys x)
  (PATS (MEMBER x (DELETE xs ys)))
  (IMPLIES (EQ (MEMBER x ys) true)
    (NEQ (MEMBER x (DELETE xs ys)) true))))
```

if $x \in ys$ then $x \notin xs - ys$

8.2.3 Result and Further Simplification

Call button pressed operation combines algorithmic refining to table (Theorem 7.1 b) with data refinement with predicates (Theorem 7.4 b) for data abstraction. The result after further simplifications is the tabular predicate used for defining the relation *abButtonPressed*:

| | w = true | w = false |
|-----------|--|--|
| f > floor | r1 = true ∧ w1 = false ∧ floor1 = floor | r1 = true ∧ w1 = w ∧ floor1 = floor |
| f = floor | r1 = false ∧ w1 = true ∧ floor1 = floor | r1 = r ∧ w1 = w ∧ floor1 = floor |
| f < floor | r1 = true ∧ w1 = false ∧ floor1 = floor | r1 = true ∧ w1 = w ∧ floor1 = floor |

The formal proof of above abstraction by our decoding relation $RW(reqs, mode)(r, w)$ is implemented by calling function *drpb* passing two parameters with $p = p$ and $r = r$:

let *buttonPressed* = *reader*

```

"BEGTAB_LHEADER_f>_floor_$f=_floor_$f<_floor_//
UHEADER_mode=_waiting_$mode/=waiting_//
reqs1={f}_&_mode1=up_&_floor1=floor_$
reqs1=reqsU{f}_&_mode1=mode_&_floor1=floor//
reqs1={}_&_mode1=waiting_&_floor1=floor_$
reqs1=reqs_&_mode1=mode_&_floor1=floor//
reqs1={f}_&_mode1=down_&_floor1=floor_$
reqs1=reqsU{f}_&_mode1=mode_&_floor1=floor//ENDTAB"
in let abButtonPressed = reader
"BEGTAB_LHEADER_f>_floor_$f=floor_$f<_floor_//
UHEADER_w=true_$w=false_//
r1=true_&_w1=false_&_floor1=floor_$r1=true_&_w1=w_&_floor1=floor//
r1=false_&_w1=true_&_floor1=floor_$r1=r_&_w1=w_&_floor1=floor//
r1=true_&_w1=false_&_floor1=floor_$r1=true_&_w1=w_&_floor1=floor//
ENDTAB"
in let p = RE(ST(buttonPressed,[reader "mode";reader "reqs"]),
ST(abButtonPressed,[reader "w";reader "r"]))
in let r = reader "(r=true_<=>_reqs/={})_&_(w=true_<=>_mode=waiting)"
in if (drpb p r) then begin
...print buttonPressed, abButtonPressed and r to standard output...
...inform result of data abstraction from buttonPressed by decoding
relation r is algorithmically abstracted to abButtonPressed...
end

```

We now use Theorem 2.9 (with transposition) to join the first and last row.

| | w = true | w = false |
|----------------|--|--|
| f \neq floor | r1 = true \wedge w1 = false \wedge floor1 = floor | r1 = true \wedge w1 = w \wedge floor1 = floor |
| f = floor | r1 = false \wedge w1 = true \wedge floor1 = floor | r1 = r \wedge w1 = w \wedge floor1 = floor |

It is verified by calling function *sjrc* passing four parameters with *ifrow* = *true*, *from1* = 0, *from2* = 2, and (*TABLE*(*t1*), *TABLE*(*t2*)) = (*joinButtonPressed*, *abButtonPressed*).

```
let joinButtonPressed = reader "BEGTAB\LHEADER\f/=floor_\$f=floor_\//
  UHEADER\w=true_\$w=false_\//
  r1=true_\&w1=false_\&floor1=floor\$r1=true_\&w1=w_\&floor1=floor//
  r1=false_\&w1=true_\&floor1=floor\$r1=r_\&w1=w_\&floor1=floor//
  ENDTAB"
in if (sjrc true 0 2 (joinButtonPressed, abButtonPressed)) then begin
  ...inform joining the first and last row of abButtonPressed is joinButtonPressed...
  ...print joinButtonPressed to screen and a latex file...
end
```

The final result of applying Theorem 2.8(a) to simplify the rightmost column is the table below.

| | w = true | w = false |
|----------------|--|--|
| f \neq floor | r1 = true \wedge w1 = false \wedge floor1 = floor | r1 = true \wedge w1 = false \wedge floor1 = floor |
| f = floor | r1 = false \wedge w1 = true \wedge floor1 = floor | r1 = r \wedge w1 = false \wedge floor1 = floor |

The formal proof is implemented by calling function *rtea* passing one parameter with $(TABLE(t1), TABLE(t2)) = (joinButtonPressed, simpButtonPressed)$.

```
let simpButtonPressed = reader "BEGTAB_LHEADER_f/=floor_$_f=floor_//
    UHEADER_w=true$_w=false_//
    r1=true_&_w1=false_&_floor1=floor$_r1=true_&_w1=false_&_floor1=floor
    //_r1=false_&_w1=true_&_floor1=floor$_r1=r_&_w1=false_&_floor1=floor
    //_ENDTAB"
in if (rtea (joinButtonPressed, simpButtonPressed)) then begin
    ...inform using Theorem 2.8(a) to simplify the rightmost column of joinButtonPressed
        is simpButtonPressed...
    ...print simpButtonPressed to screen and a latex file...
end
```

8.3 Elevator Scheduling Refinement

8.3.1 Case Introduction

In contrast to the abstraction of *buttonPressed*, we will present a refinement of *scheduling* in which we increase the state space. When a *traveling* elevator reaches a specific floor, which is captured by a sensor, the system will determine the membership of current floor from the set of requested floors and perform different sequences of operations.

1. Current floor is not in requested floors, formally represented by $rs = 0$.
 - (a) Motor keeps moving, formally represented by $s = false$.
 - (b) Door is still closed, formally represented by $c = true$.
 - (c) Mode remains busy, formally represented by $w = false$.

2. Current floor is in one-element set of requested floors, formally represented by $rs = 1$.

- (a) Motor stops, formally represented by $s = true$.
- (b) Door opens, formally represented by $c = false$.
- (c) Mode becomes idle, formally represented by $w = true$.

3. Current floor is in multiple-element set of requested floors, formally represented by $rs = 2$.

- (a) Motor stops.
- (b) Door opens.
- (c) Mode remains busy.

Based on above requirement, we define our abstract specification as:

$$w = false \wedge floor1 = floor \wedge (rs = 0 \wedge c1 = true \wedge s1 = false \wedge w1 = w \wedge rs1 = rs \vee c1 = false \wedge s1 = true \wedge rs1 = 0 \wedge (rs = 2 \wedge w1 = false \vee rs = 1 \wedge w1 = true))$$

In the case of *scheduling* abstract specification we make following observations:

- The abstract boolean variable s , c and w could be replaced by enumeration variable *motor*, *door* and *mode* respectively to indicate their concrete states.
- We could consider more detailed relations between current floor and the set of requested floors by introducing set *reqs*.
- The refinement through relation $SCWRS(s, c, w, rs)(motor, door, mode, reqs)$ can be broken into two steps by a composition relation $SCW(s, c, w)(motor, door, mode) \circ RS(rs)(reqs)$. The reason is that $SCW(s, c, w)(motor, door, mode)$ is

represented as a plain predicate while $RS(rs)(mode)$ is represented as a vector predicate table and the refinements on these two encoding relations apply different theorems.

8.3.2 Stepwise Data Refinement

The refinement on encoding relation $SCW(s, c, w)(motor, door, mode)$:

$$(c = true \Leftrightarrow door = closed) \wedge (s = true \Leftrightarrow motor = stop) \wedge (w = true \Leftrightarrow mode = waiting)$$

can be represented in plain predicate *stepRefScheduling* as:

$$mode \neq waiting \wedge floor1 = floor \wedge (rs = 0 \wedge door1 = closed \wedge motor1 \neq stop \wedge model = mode \wedge rs1 = rs \vee door1 = open \wedge motor1 = stop \wedge rs1 = 0 \wedge (rs = 2 \wedge model \neq waiting \vee rs = 1 \wedge model = waiting))$$

The formal proof of above refinement is implemented by calling function *drpp* passing two parameters with $p = p$ and $r = r$:

```
let scheduling = reader "w=false⊔⊔floor1=floor&
  ((rs=0⊔⊔c1=true⊔⊔s1=false⊔⊔w1=w⊔⊔rs1=rs)⊔or⊔(c1=false&s1=true&
  rs1=0⊔⊔((rs=2⊔⊔w1=false)⊔or⊔(rs=1⊔⊔w1=true))))"
in let r = reader "(c=true⊔=>⊔door=closed)⊔&
  (s=true⊔=>⊔motor=⊔stop)⊔⊔(w=true⊔=>⊔mode=waiting)"
in let stepRefScheduling = reader "mode/=waiting&floor1=floor
  &((rs=0⊔⊔door1=closed⊔⊔motor1/=stop⊔⊔model=mode⊔⊔rs1=rs)⊔or
  (door1=open⊔⊔motor1=stop⊔⊔rs1=0⊔⊔((rs=2⊔⊔model/=waiting)
  or⊔(rs=1⊔⊔model=waiting))))"
in let p = RE(ST(scheduling,[reader "c";reader "s";reader "w"]),
  ST(stepRefScheduling,[reader "door";reader "motor";reader "mode"]))
in if drpp p r 0 then begin
  ...print scheduling, stepRefScheduling and r to standard output...
  ...inform result of data abstraction from scheduling by
```

encoding relation r is `stepRefScheduling...`

end

For the relation of current floor and the set of requested floors, we can divide it into five categories:

1. Current floor is not in requested floors, represented as: $floor \notin reqs$.
2. Current floor is in requested floors and there are no requests in both up and down directions, represented as:
$$floor \in reqs \wedge \neg(\exists f \cdot f \in reqs \wedge f > floor) \wedge \neg(\exists f \cdot f \in reqs \wedge f < floor).$$
3. Current floor is in requested floors and there are only requests in down direction, represented as:
$$floor \in reqs \wedge \neg(\exists f \cdot f \in reqs \wedge f > floor) \wedge (\exists f \cdot f \in reqs \wedge f < floor).$$
4. Current floor is in requested floors and there are only requests in up direction, represented as:
$$floor \in reqs \wedge (\exists f \cdot f \in reqs \wedge f > floor) \wedge \neg(\exists f \cdot f \in reqs \wedge f < floor).$$
5. Current floor is in requested floors and there are requests in both directions, represented as:
$$floor \in reqs \wedge (\exists f \cdot f \in reqs \wedge f > floor) \wedge (\exists f \cdot f \in reqs \wedge f < floor).$$

We reflect above relation to our abstract relation by a vector table:

| | | | | | |
|-----|-----------------------------------|--|--|--|--|
| | $\text{floor} \notin \text{reqs}$ | $\text{floor} \in \text{reqs} \wedge \neg(\exists f. f \in \text{reqs} \wedge f > \text{floor}) \wedge \neg(\exists f. f \in \text{reqs} \wedge f < \text{floor})$ | $\text{floor} \in \text{reqs} \wedge \neg(\exists f. f \in \text{reqs} \wedge f > \text{floor}) \wedge (\exists f. f \in \text{reqs} \wedge f < \text{floor})$ | $\text{floor} \in \text{reqs} \wedge (\exists f. f \in \text{reqs} \wedge f > \text{floor}) \wedge \neg(\exists f. f \in \text{reqs} \wedge f < \text{floor})$ | $\text{floor} \in \text{reqs} \wedge (\exists f. f \in \text{reqs} \wedge f > \text{floor}) \wedge (\exists f. f \in \text{reqs} \wedge f < \text{floor})$ |
| rs= | 0 | 1 | 2 | 2 | 2 |

Table 8.5: Encoding Relation $R(rs)(reqs)$

We apply theorem 7.5 (Data Refinement with Vector Table) to get the refinement of intermediate specification *StepRefScheduling* on encoding relation $RS(rs)(reqs)$.

It can be represented in standard table as:

| | | | | | |
|---|---|---|---|---|---|
| | $\text{floor} \notin \text{reqs1}$ $\neg(\exists f. f \in \text{reqs1} \wedge f > \text{floor}) \wedge \neg(\exists f. f \in \text{reqs1} \wedge f < \text{floor})$ | $\text{floor} \in \text{reqs1}$ $\neg(\exists f. f \in \text{reqs1} \wedge f > \text{floor}) \wedge (\exists f. f \in \text{reqs1} \wedge f < \text{floor})$ | $\text{floor} \in \text{reqs1}$ $\neg(\exists f. f \in \text{reqs1} \wedge f > \text{floor}) \wedge (\exists f. f \in \text{reqs1} \wedge f < \text{floor})$ | $\text{floor} \in \text{reqs1}$ $(\exists f. f \in \text{reqs1} \wedge f > \text{floor}) \wedge \neg(\exists f. f \in \text{reqs1} \wedge f < \text{floor})$ | $\text{floor} \in \text{reqs1}$ $(\exists f. f \in \text{reqs1} \wedge f > \text{floor}) \wedge (\exists f. f \in \text{reqs1} \wedge f < \text{floor})$ |
| $\text{floor} \notin \text{reqs}$ | $\text{mode} \neq \text{waiting} \wedge \text{door1} = \text{closed} \wedge \text{motor1} \neq \text{stop} \wedge \text{mode1} = \text{mode} \wedge \text{floor1} = \text{floor}$ | FALSE | FALSE | FALSE | FALSE |
| $\text{floor} \in \text{reqs} \wedge \neg(\exists f. f \in \text{reqs}$ | $\text{mode} \neq \text{waiting}$ | FALSE | FALSE | FALSE | FALSE |

| | | | | | |
|--|---|-------|-------|-------|-------|
| $\wedge f > \text{floor}) \wedge$ $\neg(\exists f. f \in \text{reqs}$ $\wedge f < \text{floor})$ | $\wedge \text{door1}$ $= \text{open} \wedge$ $\text{motor1} =$ $\text{stop} \wedge$ $\text{model} =$ waiting $\wedge \text{floor1}$ $= \text{floor}$ | | | | |
| $\text{floor} \in \text{reqs} \wedge$ $\neg(\exists f. f \in \text{reqs}$ $\wedge f > \text{floor}) \wedge$ $(\exists f. f \in \text{reqs}$ $\wedge f < \text{floor})$ | $\text{mode} \neq$ waiting $\wedge \text{door1}$ $= \text{open} \wedge$ $\text{motor1} =$ $\text{stop} \wedge$ $\text{model} \neq$ waiting $\wedge \text{floor1}$ $= \text{floor}$ | FALSE | FALSE | FALSE | FALSE |
| $\text{floor} \in \text{reqs} \wedge$ $(\exists f. f \in \text{reqs}$ $\wedge f > \text{floor}) \wedge$ $\neg(\exists f. f \in \text{reqs}$ $\wedge f < \text{floor})$ | $\text{mode} \neq$ waiting $\wedge \text{door1}$ $= \text{open} \wedge$ $\text{motor1} =$ $\text{stop} \wedge$ $\text{model} \neq$ | FALSE | FALSE | FALSE | FALSE |

| | | | | | |
|--|---|-------|-------|-------|-------|
| | waiting \wedge floor1 $=$ floor | | | | |
| floor \in reqs \wedge $(\exists f. f \in$ reqs $\wedge f > \text{floor}) \wedge$ $(\exists f. f \in$ reqs $\wedge f < \text{floor})$ | mode \neq waiting \wedge door1 $=$ open \wedge motor1 = stop \wedge mode1 \neq waiting \wedge floor1 $=$ floor | FALSE | FALSE | FALSE | FALSE |

It is implemented by calling function *drv* with parameter $p = p$ and $r = r$.

```

let r = reader "BEGTAB_LHEADER_rs=//
    UHEADER_floor/:reqs_$floor:reqs_&not_(&f|f:reqs_&f>floor)
    &not_(&f|f:reqs_&f<floor)_$floor:reqs_&not_(&f|f:reqs_&
    f>floor)_&(&f|f:reqs_&f<floor)_$floor:reqs_&(&f|f:reqs
    &f>floor)_&not_(&f|f:reqs_&f<floor)_$floor:reqs_&
    (&f|f:reqs_&f>floor)_&(&f|f:reqs_&f<floor)_//
    0_$1_$2_$2_//_ENDTAB"
in let refScheduling = reader "BEGTAB_LHEADER_
    floor/:reqs_$floor:reqs_&not_(&f|f:reqs_&f>floor)_&
    not_(&f|f:reqs_&f<floor)_$floor:reqs_&not_(&f|f:reqs_&
    f>floor)_&(&f|f:reqs_&f<floor)_$floor:reqs_&(&f|f:reqs
    &f>floor)_&not_(&f|f:reqs_&f<floor)_$floor:reqs_&
    (&f|f:reqs_&f>floor)_&(&f|f:reqs_&f<floor)_//_UHEADER
    floor/:reqs1_$floor:reqs1_&not_(&f|f:reqs1_&f>floor)
    
```

```

&_not_(_#f|f:reqs1_&_f<floor)_$_floor:reqs1_&_not_
(_#f|f:reqs1_&_f>floor)_&_(_#f|f:reqs1_&_f<floor)_$_
floor:reqs1_&_(_#f|f:reqs1_&_f>floor)_&_not_(_#f|f:reqs1_&
f<floor)_$_floor:reqs1_&_(_#f|f:reqs1_&_f>floor)_&
(_#f|f:reqs1_&_f<floor)_$_//_mode/=waiting_&_door1=closed_&
motor1/=stop_&_mode1=mode_&_floor1=floor_$_FALSE_$_FALSE_$_FALSE_$_
FALSE//mode/=waiting_&_door1=open_&_motor1=stop_&_mode1=waiting
&_floor1=floor_$_FALSE_$_FALSE_$_FALSE_$_FALSE//_mode/=waiting_&
door1=open_&_motor1=stop_&_mode1/=waiting_&_floor1=floor_$_FALSE_$_
FALSE_$_FALSE_$_FALSE//_mode/=waiting_&_door1=open_&_motor1=stop_&
mode1/=waiting_&_floor1=floor_$_FALSE_$_FALSE_$_FALSE_$_FALSE//
mode/=waiting&door1=open&motor1=stop&mode1/=waiting&floor1=floor_$_
FALSE_$_FALSE_$_FALSE_$_FALSE//_ENDTAB"
in let p = RE(ST(stepRefScheduling,[reader "rs"]),
  ST(refScheduling,[reader "reqs"]))
in if (drv p r) then begin
  ...print stepRefScheduling, refScheduling and r to standard output...
  ...inform data refinement of operation stepRefScheduling by encoding
    relation r is refScheduling...
end

```

We make following adjustment to clarify table structure.

1. Push upper header to each cell of table body.
2. Remove columns which equal to FALSE.
3. Lift predicate $\text{mode} \neq \text{waiting}$ to upper header.
4. Splitting table by copying one column to another with $\text{mode} = \text{up}$ and $\text{mode} = \text{down}$ as their upper header corresponding to each column.

After reorganization, we have following specification in tabular form:

| | mode = up | mode = down |
|--|---|---|
| $\text{floor} \notin \text{reqs}$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{closed} \wedge \text{motor1} \neq \text{stop} \wedge \text{mode1} = \text{mode} \wedge \text{floor1} = \text{floor}$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{closed} \wedge \text{motor1} \neq \text{stop} \wedge \text{mode1} = \text{mode} \wedge \text{floor1} = \text{floor}$ |
| $\text{floor} \in \text{reqs} \wedge \neg(\exists f. f \in \text{reqs} \wedge f > \text{floor}) \wedge \neg(\exists f. f \in \text{reqs} \wedge f < \text{floor})$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{open} \wedge \text{motor1} = \text{stop} \wedge \text{mode1} = \text{waiting} \wedge \text{floor1} = \text{floor}$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{open} \wedge \text{motor1} = \text{stop} \wedge \text{mode1} = \text{waiting} \wedge \text{floor1} = \text{floor}$ |
| $\text{floor} \in \text{reqs} \wedge \neg(\exists f. f \in \text{reqs} \wedge f > \text{floor}) \wedge (\exists f. f \in \text{reqs} \wedge f < \text{floor})$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{open} \wedge \text{motor1} = \text{stop} \wedge \text{mode1} \neq \text{waiting} \wedge \text{floor1} = \text{floor}$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{open} \wedge \text{motor1} = \text{stop} \wedge \text{mode1} \neq \text{waiting} \wedge \text{floor1} = \text{floor}$ |
| $\text{floor} \in \text{reqs} \wedge (\exists f. f \in \text{reqs} \wedge f > \text{floor}) \wedge \neg(\exists f. f \in \text{reqs} \wedge f < \text{floor})$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{open} \wedge \text{motor1} = \text{stop} \wedge \text{mode1} \neq \text{waiting} \wedge \text{floor1} = \text{floor}$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{open} \wedge \text{motor1} = \text{stop} \wedge \text{mode1} \neq \text{waiting} \wedge \text{floor1} = \text{floor}$ |
| $\text{floor} \in \text{reqs} \wedge (\exists f. f \in \text{reqs} \wedge f > \text{floor}) \wedge (\exists f. f \in \text{reqs} \wedge f < \text{floor})$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{open} \wedge \text{motor1} = \text{stop} \wedge \text{mode1} \neq \text{waiting} \wedge \text{floor1} = \text{floor}$ | $\text{floor} \notin \text{reqs1} \wedge \text{door1} = \text{open} \wedge \text{motor1} = \text{stop} \wedge \text{mode1} \neq \text{waiting} \wedge \text{floor1} = \text{floor}$ |

| | | |
|----------------------------|--|--|
| $\wedge f < \text{floor})$ | | |
|----------------------------|--|--|

By above specification, a single elevator can decide where to stop. It is still not refined enough to decide the direction for a busy elevator. There are several algorithm to decide which request to service next such as *elevator algorithm* and *heuristic algorithm* [1]. We will adopt the elevator algorithm here to demonstrate an algorithmic refinement. The elevator algorithm is executed:

1. Continue traveling in the same direction while there are remaining requests in that same direction.
2. If there are no further requests in that direction, then become idle, or change direction if there are requests in the opposite direction.

Besides above algorithm, our algorithm refinement also specify the state changes of set *reqs* when a specific floor is reached by an assignment statement instead of set properties. The final refinement table according to theses modification is:

| | mode = up | mode = down |
|--|--|--|
| $\text{floor} \notin \text{reqs}$ | $\text{reqs1} = \text{reqs} \wedge \text{door1} =$ $\text{closed} \wedge \text{motor1} = \text{up} \wedge$ $\text{mode1} = \text{up} \wedge \text{floor1} =$ floor | $\text{reqs1} = \text{reqs} \wedge \text{door1} =$ $\text{closed} \wedge \text{motor1} = \text{down} \wedge$ $\text{mode1} = \text{down} \wedge \text{floor1} =$ floor |
| $\text{floor} \in \text{reqs} \wedge$ $\neg(\exists f. f \in \text{reqs}$ $\wedge f > \text{floor}) \wedge$ $\neg(\exists f. f \in \text{reqs}$ $\wedge f < \text{floor})$ | $\text{reqs1} = \text{reqs} - \{\text{floor}\} \wedge$ $\text{door1} = \text{open} \wedge \text{motor1} =$ $\text{stop} \wedge \text{mode1} = \text{waiting} \wedge$ $\text{floor1} = \text{floor}$ | $\text{reqs1} = \text{reqs} - \{\text{floor}\} \wedge$ $\text{door1} = \text{open} \wedge \text{motor1} =$ $\text{stop} \wedge \text{mode1} = \text{waiting} \wedge$ $\text{floor1} = \text{floor}$ |

| | | |
|--|---|---|
| $\text{floor} \in \text{reqs} \wedge$ $\neg(\exists f. f \in \text{reqs}$ $\wedge f > \text{floor}) \wedge$ $(\exists f. f \in \text{reqs}$ $\wedge f < \text{floor})$ | $\text{reqs1} = \text{reqs} - \{\text{floor}\} \wedge$ $\text{door1} = \text{open} \wedge \text{motor1} =$ $\text{stop} \wedge \text{mode1} = \text{down} \wedge$ $\text{floor1} = \text{floor}$ | $\text{reqs1} = \text{reqs} - \{\text{floor}\} \wedge$ $\text{door1} = \text{open} \wedge \text{motor1} =$ $\text{stop} \wedge \text{mode1} = \text{down} \wedge$ $\text{floor1} = \text{floor}$ |
| $\text{floor} \in \text{reqs} \wedge$ $(\exists f. f \in \text{reqs}$ $\wedge f > \text{floor}) \wedge$ $\neg(\exists f. f \in \text{reqs}$ $\wedge f < \text{floor})$ | $\text{reqs1} = \text{reqs} - \{\text{floor}\} \wedge$ $\text{door1} = \text{open} \wedge \text{motor1} =$ $\text{stop} \wedge \text{mode1} = \text{up} \wedge$ $\text{floor1} = \text{floor}$ | $\text{reqs1} = \text{reqs} - \{\text{floor}\} \wedge$ $\text{door1} = \text{open} \wedge \text{motor1} =$ $\text{stop} \wedge \text{mode1} = \text{up} \wedge$ $\text{floor1} = \text{floor}$ |
| $\text{floor} \in \text{reqs} \wedge$ $(\exists f. f \in \text{reqs}$ $\wedge f > \text{floor}) \wedge$ $(\exists f. f \in \text{reqs}$ $\wedge f < \text{floor})$ | $\text{reqs1} = \text{reqs} - \{\text{floor}\} \wedge$ $\text{door1} = \text{open} \wedge \text{motor1} =$ $\text{stop} \wedge \text{mode1} = \text{up} \wedge$ $\text{floor1} = \text{floor}$ | $\text{reqs1} = \text{reqs} - \{\text{floor}\} \wedge$ $\text{door1} = \text{open} \wedge \text{motor1} =$ $\text{stop} \wedge \text{mode1} = \text{down} \wedge$ $\text{floor1} = \text{floor}$ |

The formal proof of applying Theorem 7.1(b) is implemented by calling function *rtt* passing one parameter with $p = p$:

```
let adjScheduling = reader "BEGTAB_LHEADER_floor/:reqs_$
  floor:reqs_&_not(#f|f:reqs&f>floor)_&_not(#f|f:reqs&f<floor)
  $_floor:reqs_&_not_(_#f|f:reqs&f>floor)_&_(_#f|f:reqs&f<floor)
  $_floor:reqs_&_(_#f|f:reqs&f>floor)_&_not_(_#f|f:reqs&f<floor)
  $_floor:reqs_&_(_#f|f:reqs&f>floor)_&_(_#f|f:reqs&f<floor)_//
  UHEADER_mode=up_$mode=down_//
  floor/:reqs1_&door1=closed_&motor1/=stop&mode1=mode&floor1=floor$
  floor/:reqs1_&door1=closed_&motor1/=stop&mode1=mode&floor1=floor//
  floor/:reqs1_&door1=open&motor1=stop&mode1=waiting&floor1=floor_$
  floor/:reqs1_&door1=open&motor1=stop&mode1=waiting&floor1=floor_//
```

```

floor/:reqs1_&door1=open&motor1=stop&mode1/=waiting&floor1=floor_&
floor/:reqs1_&door1=open&motor1=stop&mode1/=waiting&floor1=floor//
floor/:reqs1_&door1=open&motor1=stop&mode1/=waiting&floor1=floor_&
floor/:reqs1_&door1=open&motor1=stop&mode1/=waiting&floor1=floor//
floor/:reqs1_&door1=open&motor1=stop&mode1/=waiting&floor1=floor_&
floor/:reqs1&door1=open&motor1=stop&mode1/=waiting&floor1=floor//
ENDTAB"

in let algScheduling = reader "BEGTAB_LHEADER_&floor/:reqs_&
floor:reqs_&_not(#f|f:reqs&f>floor)_&_not(#f|f:reqs&f<floor)
_&floor:reqs_&_not_(&f|f:reqs&f>floor)_&_(&f|f:reqs&f<floor)
_&floor:reqs_&_(&f|f:reqs&f>floor)_&_not_(&f|f:reqs&f<floor)
_&floor:reqs_&_(&f|f:reqs&f>floor)_&_(&f|f:reqs&f<floor)_//
UHEADER_mode=up_&mode=down_//
reqs1=reqs&door1=closed&motor1=up&mode1=up&floor1=floor_&
reqs1=reqs&door1=closed&motor1=down&mode1=down&floor1=floor_//
reqs1=reqs--{floor}&door1=open&motor1=stop&mode1=waiting&floor1=
floor_&reqs1=reqs--{floor}&door1=open&motor1=stop&mode1=waiting_&
floor1=floor//reqs1=reqs--{floor}&door1=open&motor1=stop&mode1=down
&floor1=floor_&reqs1=reqs--{floor}&door1=open&motor1=stop&mode1=
down&floor1=floor//reqs1=reqs--{floor}&door1=open&motor1=stop&
mode1=up&floor1=floor_&reqs1=reqs--{floor}&door1=open&motor1=stop&
mode1=up&floor1=floor_//reqs1=reqs--{floor}&door1=open&motor1=stop&
mode1=up&floor1=floor_&reqs1=reqs--{floor}&door1=open&motor1=stop&
mode1=down&floor1=floor_//ENDTAB"

in let p = RE(ST(algScheduling, [reader "reqs";reader "door";
reader "motor";reader "mode"]), ST(adjScheduling,
[reader "reqs";reader "door";reader "motor";reader "mode"]))
in if (rtt p) then begin
...print adjScheduling, algScheduling and r to standard output...
...inform algorithm refinement of operation adjScheduling by
encoding relation r is algScheduling...
end

```


8.4 Summary

The result of our implementation is listed in Figure 8.1. We applied the theorems of **refining to table** and **data refining with predicates** for an abstraction. The result of such a proof is a table with the same structure as the concrete table. The abstraction reduces the size of our specification from 176 characters to 115 characters. The proof related to tabular abstraction takes 17 millisecond. The proof related to abstraction in terms of plain predicate specification takes 14 millisecond. We apply the theorem of **splitting and joining rows and columns** to join the first and last row, and get a table with 3 rows and 3 columns. Finally, the theorem of **replacing table elements** is used to simplify the rightmost column. The elevator scheduling is stepwise refined to execute the elevator algorithm. The theorem of **soundness of encoding** is applied in order to prove the first step of our refinement, the proof of which takes the longest running time (29,118 millisecond) in our applications. We apply the theorem of **data refinement with vector table** to prove the second step of our refinement in 70 millisecond; while the encoding relation is represented by plain predicate, Simplify fails to prove the valid of it. We apply the theorem of **refining to table** to refine our tabular specification in the same state space, the proof of which takes 30 millisecond. The proof of such a refinement in terms of plain predicate specification takes 32 millisecond.

From the elevator example, we observe that data refinement (abstraction) and algorithmic refinement (abstraction) are normally applied together to refine (abstract) a specification. Data refining can be carried out in a stepwise manner and different elements of a composite relation could be represented in different forms. Tabular refinement are successfully implemented in elevator system.

| Operation | Proof Condition | Coding Size | Abstract Size | Refined Size | Time(msec) | | Theorems Applied |
|----------------|-----------------|-------------------------|--------------------------|--------------------------|------------|--------|---|
| | | | | | Tabular | Plain | |
| Button Pressed | 6 | 28c | 4 rows 3 cols 115c | 4 rows 3 cols 176c | 17 | 14 | Refining to Table Data Refining with redicates |
| | | | 3 rows 3 cols 80c | | 11 | | Splitting and Joining Rows and Columns |
| | | | 3 rows 3 cols 88c | | 11 | | Replacing Table Elements |
| Scheduling | 1 | 62c | 103c | 139c | | 29,118 | Soundness of Encoding |
| | 25 | 2 row 6 cols 222c | 139c | 6 rows 6 cols 785c | 70 | fail | Data Refinement with Vector Table |
| | 25 | | 6 rows 3 cols 708c | 6 rows 3 cols 734c | 30 | 32 | Refining to Table |

Figure 8.1: Performance of Proof in Elevator Control

Chapter 9

Conclusion and Future Work

In this thesis, we presented an implementation of a new table tool which includes support for both specifications and refinements. A parser is developed based on the recursive descent parsing technique such that our formulae can be inputted through either an export function *reader* or an expression of data type *form* directly. A printing file include the functions to print a single formula to screen and a \LaTeX file respectively. A number of theorems are applied in our validation functions. Our source code has three files defining the compilation units, *table.ml*, *print.ml*, and *parse.ml*. When their compiled files are linked together with a Unix library to produce a executable file (e.g. *theprogram*), the command is as follows:

```
ocamlc -o theprogram Unix.cma parse.cmo print.cmo table.cmo
```

In such an order, the definitions and declarations contained in *table.ml* can refer to definition in *print.ml*, *parse.ml*, and *Unix.cma*; *print* can refer to *parse* and *Unix*; *parse* can refer to *Unix*.

We did several experiments to call our validation functions. They are examples of control system and information management system. Axioms for each example are set to specify types and properties. It can be seen that it is much easier and more

convenient to specify and refine a program in its tabular predicate form. Large manipulations are broken into several small parts. Concerns are divided so that designer and theorem prover can solve them separately. By introducing this implementation, we are able to design simple and complex cases of tabular specification and refinement and generate executable code for them in OCaml programming language.

One plain predicate in car seat example is proved in longer time than its equivalent tabular predicate. All plain predicates in visitor information system example are proved in shorter time than their equivalent tabular predicates. One plain predicate in elevator example is proved in longer time than its equivalent tabular predicate. One valid plain predicate in elevator example is failed to prove its correctness by Simplify. We conclude that large predicates can be proved more efficient if they are in tabular form. By applying theorem of tabular specification and refinement, failures of Simplify are reduced since the inputs of Simplify are smaller decomposed predicates instead of complex predicates.

According to our implementation and previous works, there are some possible improvement in specifying and refining a program, such as:

- The output to screen are now ASCII characters and can be replaced by Unicode.
- Optimization could be made on existing code.
- More theorems are to be developed together with examples applying them.
- Specification and refinement can be manipulated on 10 classes of tables defined by Parnas.
- Algorithmic refinement could be extended to its definition on partial relations.

The theorems and implementation of it could be derived.

Bibliography

- [1] Elevator. Wikimedia Foundation, Inc, September 2006.
- [2] R. Abraham. Evaluating generalized tabular expressions in software documentation. Master's thesis, McMaster University, Hamilton, Ont., February 1997.
- [3] J.M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transaction Software Engineering*, 19(1):24–40, January 1993.
- [4] R-J. Back and J. von Wright. Encoding, decoding, and data refinement. *Formal Aspects of Computing*, 12(5):313–349, 2000.
- [5] Bazaar. Ocaml - ml language implementation with a class-based object system. Debian, 1997-2006. <http://packages.debian.org/oldstable/devel/ocaml>.
- [6] J. Crow and B. Di Vito. Formalizing space shuttle software requirements: four case studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):296–332, July 1998.
- [7] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

- [8] S.R. Faulk. *State Determination in Hard-Embedded Systems*. Phd thesis, University of North Carolina, Chapel Hill, 1989.
- [9] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *Proceedings of the European symposium on programming on ESOP 86*, pages 187–196. Lecture Notes in Computer Science 213, Springer-Verlag, 1986.
- [10] Eric C.R. Hehner. Predicate programming. *Communications of the ACM*, 27(2):144–151, February 1984.
- [11] M.P.E. Heimdahl and N.G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
- [12] C. Heitmeyer, A.Bull, C. Gasarch, and B. Labaw. SCR: A toolset for specifying and analyzing requirements. In *Proceedings of the 10th Annual Conference on Computer Assurance*, pages 109–122, New York, June 1995. IEEE.
- [13] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
- [14] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.
- [15] D.N. Hoover and Z. Chen. Tablewise, a decision table tool. In Md. Gaithersburg, editor, *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS '95)*, pages 97–108, New York, June 1995. IEEE.

- [16] M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl, and B. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.
- [17] R. Janicki and A. Wassying. On tabular expressions. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 92–106, Toronto, Ont., 2003. IBM Centre for Advanced Studies Conference, IBM Press.
- [18] M. Lawford, J. McDougall, P. Froebel, and G. Moum. Practical application of functional and relational methods for the specification and verification of safety critical software. In T. Rus, editor, *Proceedings Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 2000*, volume 1816 of *LNCS*, pages 73–88. Springer, May 2000.
- [19] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994. September.
- [20] S. Nejati. Refinement relations on partial specifications. Master’s thesis, University of Toronto, Department of Computer Science, University of Toronto, 2003.
- [21] G. Nelson and D. Detlefs. *Extended Static Checking for Java*. Compaq Systems Research, Palo Alto, CA, Compaq Computer Corporation edition, 1999,2000.
- [22] S. Owre, J. Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In *Slightly expanded version of a paper presented at the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’97)*, volume 1217 of *LNCS*, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.

- [23] D.L. Parnas. Tabular representation of relations. CRL Report 260, McMaster University, October 1992.
- [24] D.L. Parnas. Inspection of safety-critical software using program-function tables. In *Proceedings of the IFIP World Congress*, volume III, pages 270–277, August 1994.
- [25] D.L. Parnas, G.J.K. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, (32):189–198, 1991.
- [26] D.L. Parnas and J. Madey. Functional documentation for computer systems engineering (version 2). Technical Report CRL 237, Telecommunications Research Institution of Ontario, McMaster University, Hamilton, Ontario, 1991.
- [27] D.L. Parnas, J. Madey, and M. Iglewski. Formal documentation of well-structured programs. CRL Report 259, Communications Research Laboratory, McMaster University, 1992.
- [28] L.C. Paulson. *ML for the Working Programmer*. The Press Syndicate of the University of Cambridge, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, United Kingdom, second edition edition, 1996.
- [29] L.W. Roberts and M. Berims. Using formal methods to assist in the requirements analysis of the space shuttle hac change request (cr 90960e). Technical Report JSC 27599, NASA Johnson Space Center, Houston,TX, September 1996.
- [30] G. Schmidt and T. Strohlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. Springer-Verlag, August 1992.

- [31] E. Sekerinski. *Computer Science 3EA3—Software Design II Assignment 1*, pages 1–3. McMaster University, 1280 Main Street West Hamilton, Ontario, L8S 4K1, October 2002.
- [32] E. Sekerinski. Exploring tabular verification and refinement. *Formal Aspects of Computing*, 15(2-3):215–236, November 2003.
- [33] E. Sekerinski. *Computer Science 4TB3 Compiler Construction*, pages 84–90. McMaster University, 1280 Main Street West Hamilton, Ontario, L8S 4K1, September 2005.
- [34] E. Sekerinski. *Computer Science 703—Software Design*, chapter Lecture 3-Abstract Programs, Lecture 5-Functional Specification, pages 59–92. McMaster University, 1280 Main Street West Hamilton, Ontario, L8S 4K1, January 2006.
- [35] A.J. van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application for monitoring systems. Technical Report TR 90-276, Queen’s University, Kingston, Ontario, 1990.
- [36] A.J. Wilder and J.V. Tucker. System documentation using tables—a short course. CRL Report 306, Communications Research Laboratory, McMaster University, 1995.
- [37] J.I. Zucker. Transformations of normal and inverted function tables. *Formal Aspects of Computing*, 8(6):679–705, May 1996.