



The
University
Of
Sheffield.

Support for Model Checking Z Specifications

By:

Maria Ulfah Siregar

A thesis submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

20 December 2016

Abstract

One of deficiencies in the Z tools is that there is limited support for model checking Z specifications. To build a model checker directly for a Z specification would take considerable effort and time due to the abstraction of the language. Translating inputs of a Z specification into a language that an existing model checker tool accepts is an alternative method. Researchers at the University of Sheffield implemented a translation tool which took a Z specification and translated it into the input for the Symbolic Analysis Laboratory (SAL) tool, a framework for combining different tools for abstraction, program analysis, theorem proving and model checking, which they called Z2SAL. In this paper, support for model checking Z specifications is discussed, in which the ability of the existing Z2SAL is extended. This support includes a translation for generic constant and schema calculus. Instead of translating these aspects of the Z language into the SAL language as Z2SAL does, a Z specification containing these two notations will be pre-processed, in which a generic constant definition will be redefined to its equivalent axiomatic definition, and schema calculus will be expanded to a new schema definition. This paper discusses the implementation of these types of support, and illustration of some working examples. The discussion also includes other several issues related to a new approach in translating Z functions and constants in SAL language, which originates from the type incompatibility obtained during execution by the SAL tool, an approach to a SAL translation of embedded theorems on Z specifications, and a manual experiment on applying an abstraction on Z specifications. Results have been gathered during our experiments with the implemented support. Several of these results could be translated by Z2SAL and be executed by the SAL tool.

Declaration

I declare that the composition of this thesis and the work within are entirely my own except where explicitly stated otherwise in the text. This work has not been put forward for any other degree or professional qualification, except as stated.

Maria Ulfah Siregar

Acknowledgment

First, I thank Allah for the guidance to complete this study.

Second, I wish to express my gratitude to my supervisor, Prof John Derrick, who always supports and gives invaluable advice for me throughout this study. Last but not least, I also wish to thank my second supervisor, Dr Siobhán North, who has shown support and encouragement to me during this study.

I am much obliged to the Ministry of Religion Affairs of the Indonesia Government for offering and funding my study in the United Kingdom and to UIN Sunan Kalijaga for giving me permission and time to pursue this study.

I would like to thank my dear parents and my big family for their love, support and advice. My success in this research is because of their praying.

I am incredibly grateful to my husband and our wonderful daughter without whom I could have finished this study. Thank you for giving me the motivation I need to undertake this study, being my best friends and patient during my study.

I would like to thank my lab mates and DCS staffs and technical support for their assistance throughout my time in this department. Specifically Mathew and Thomas Gyeera who help me in proofreading my thesis, I wish more successful works for you. Also Hanaa and Ermira who encourage me to finish this study, I wish that both of you will finish your study soon. Nina who also supports me to finish this study, I am waiting to hear your happiness when you submit your thesis.

Finally, thank you very much for my friends of awardees of MORA Scholarship 2012 and Indonesian Society in this country, who plays as an extended family, for encouragements and advice. I wish better luck for all of you.

Contents

1	Introduction	12
1.1	Research Motivation	15
1.2	Research Objective and Contribution	15
1.3	Thesis Structure	16
1.4	Publications	18
1.4.1	Peer Reviewed Journal Paper	18
1.4.2	Peer Reviewed Conference Papers	18
1.4.3	Posters	19
2	Background	20
2.1	Formal Methods	20
2.1.1	Z	21
2.1.2	VDM	21
2.1.3	RAISE Specification Language (RSL)	21
2.1.4	CSP	22
2.1.5	Conclusion	22
2.2	The Z Notation	22
2.2.1	Sets	23
2.2.2	The First-Order Predicate Logic	24
2.2.3	Schema Operators	26
	Disjunction	26
	Conjunction	26
	Negation	27
	Implication	28
	Bi-implication	28
	Renaming	29
	Hiding	29
	Schema Composition	30
	Quantification	31
2.2.4	A Z Notation Example	32
2.2.5	The Z Tools	35

2.2.6	Conclusion	38
2.3	Model Checking	39
2.3.1	The Temporal Logic	39
2.3.2	Binary Decision Diagram	44
2.3.3	Introduction to Abstraction	46
2.3.4	Abstraction on Z Specifications	47
2.3.5	Conclusion	48
2.4	The SAL Tools	49
2.4.1	A Glance at SAL	49
2.4.2	SAL Components	52
2.4.3	The SAL Environment	53
	The SAL Model Checker	53
	The SAL Simulator	57
2.4.4	Conclusion	60
2.5	The Z2SAL Translator	61
2.5.1	Introduction	61
2.5.2	The Current Z2SAL Translator	63
	The #1 Aspect: Generic Constant Definition	67
	The #2 Aspect: Schema Calculus Definition	71
2.5.3	Conclusion	73
3	Translation of Embedded Theorems in Z Specifications: A Proposed Method	74
3.1	Adding Theorems in the Generated SAL	74
3.2	Embedded Theorems on Z Specifications	76
3.2.1	Proposed Method in a Translation of Embedded Theorems on Z Specifications	79
3.2.2	Experiment 1: Unique Allocator Specification	80
3.2.3	Experiment 2: Counter 4 Modulo Specification	81
3.2.4	Experiment 3: Cars Park Specification	82
3.2.5	Experiment 4: Birthday Book Specification	83
3.2.6	Experiment 5: Paper Example Specification	83
3.2.7	Experiment 6: Shop Specification	85
3.2.8	Result and Discussion	85
3.3	Conclusion	87
4	Implementing A Z Scanner and Parser	88
4.1	A Z Scanner	89
4.1.1	Introduction	89
4.1.2	A Lexical Specification	90
	User Code	90

	Options and Declarations	90
	Lexical Rules	90
4.1.3	An Implementation of a Z Scanner	92
4.1.4	Conclusion	94
4.2	A Z Parser	96
4.2.1	Introduction	96
4.2.2	A YACC Parser	96
	Shift/Reduce Conflict	98
	Reduce/Reduce Conflict	99
	Precedence, Associativity, and Operator Declarations	99
4.2.3	An Implementation of a Z Parser	100
	The First Conflict	105
	The Second Conflict	106
	The Third Conflict	107
	The Fourth Conflict	108
	The Fifth Conflict	108
	The Sixth Conflict	109
4.2.4	Conclusion	109
5	Redefining Generic Constants	111
5.1	Introduction	111
5.2	A Method for Redefining a Generic Constant	113
5.3	An Implementation of the Redefinition System	113
	5.3.1 Reading a Z specification	113
	5.3.2 Spotting Generic Constant Definition	114
	5.3.3 Spotting Usages of Generic Constants	118
5.4	Important Findings around a Redefinition of Generic Constants	125
5.5	A Proposed Translation of SAL Function	126
5.6	Conclusion	128
6	Expanding Schema Calculus	130
6.1	Introduction	130
6.2	An Implementation of the Expansion System	130
	6.2.1 Expansion Processes in Java Main Program	131
	6.2.2 Expansion Processes in <code>schConstruction</code> function	133
	An Operator with Two Operands	135
	The First Simplification	135
	The Second Simplification	136
	The First Not Regular Simplification	137
	The Second Not Regular Simplification	137
	An Operator with a Right Operand	138

	An Operator with a Left Operand	138
	An Only Operator	139
	An Implication Operator	140
	A Bi-implication Operator	144
	A Negation Operator	147
	Separators in a Schema Calculus Definition	148
	Other Processes	148
6.2.3	Expansion Processes in <code>operate</code> function	150
	Conjunction and Disjunction	152
	Negation	152
	Renaming	153
	Hiding	153
	Composition	153
6.2.4	Expansion Processes in <code>expand</code> function	154
6.2.5	Expansion Processes in <code>normalised</code> function	155
6.2.6	Expansion Processes in <code>collapse</code> function	158
6.2.7	Expansion Processes in <code>negateSch</code> function	161
6.2.8	Expansion Processes in <code>rename</code> function	162
6.2.9	Expansion Processes in <code>hide</code> function	164
6.3	Conclusion	168
7	Integration among the Scanner, the Parser, and Java Programs	171
7.1	Java Main Program	171
7.2	Reading a Z Specification	172
7.3	Establishing a Connection Amongst Systems	177
7.4	Conclusion	177
8	A Generic Constants Redefinition	179
8.1	Setting up Questions for Evaluation	179
8.2	Experiments with the Generic Constant Redefinition	180
8.2.1	Experiment 1: <code>bbook.tex</code>	183
8.2.2	Experiment 2: <code>bbook_map.tex</code>	186
8.2.3	Experiment 3: <code>bbook_uni.tex</code>	186
8.2.4	Experiment 4: <code>bbook_map_uni.tex</code>	189
8.2.5	Experiment 5: <code>fDomRan.tex</code>	190
8.2.6	Experiment 6: <code>fEmpty.tex</code>	191
8.2.7	Experiment 7: <code>fEmptyImpl.tex</code>	192
8.2.8	Experiment 8: <code>fFirst.tex</code>	192
8.2.9	Experiment 9: <code>fHead.tex</code>	194
8.2.10	Experiment 10: <code>fHeadFunc.tex</code>	195

8.2.11	Experiment 11: fMaxComSubSeq_orig.tex	196
8.2.12	Experiment 12: fMaxComSubSeq_1.tex	199
8.2.13	Experiment 13: fMaxComSubSeq.tex	200
8.2.14	Experiment 14: fMonoSeq_1.tex	200
8.2.15	Experiment 15: fMonoSeq.tex	201
8.2.16	Experiment 16: fSwap.tex	202
8.2.17	Experiment 17: fUniqSeq.tex	203
8.2.18	Experiment 18: fUniq1Seq.tex	205
8.2.19	Experiment 19: fUniq2Seq.tex	206
8.2.20	Experiment 20: tn.tex	207
8.2.21	Experiment 21: tnImpl.tex	208
8.2.22	Experiment 22: fFileStorage.tex	208
8.2.23	Experiment 23: fSet.tex	209
8.3	Evaluation of Generic Constants Redefinition	210
8.3.1	Evaluation of the #1 Question	211
8.3.2	Evaluation of the #2 Question	214
8.3.3	Evaluation of the #3 Question	217
8.4	Conclusion	219

9 A Schema Calculus Expansion 221

9.1	Setting up Questions for an Evaluation	221
9.2	Experiments with the Schema Calculus Definitions	222
9.2.1	Experiment 1: expandingschema_1.tex	224
9.2.2	Experiment 2: expandingschema_2.tex	227
9.2.3	Experiment 3: expandingschema_3.tex	228
9.2.4	Experiment 4: expandingschema_4.tex	229
9.2.5	Experiment 5: expandingschema_5.tex	231
9.2.6	Experiment 6: expandingschema_6.tex	232
9.2.7	Experiment 7: expandingschema_7.tex	233
9.2.8	Experiment 8: expandingschema_8.tex	233
9.2.9	Experiment 9: expandingsch2_4.tex	234
9.2.10	Experiment 10: expandingsch3_1.tex	235
9.2.11	Experiment 11: expandingsch3_2.tex	236
9.2.12	Experiment 12: expandingsch3_4.tex	236
9.2.13	Experiment 13: expandingsch4_1.tex	237
9.2.14	Experiment 14: expandingsch4_2.tex	238
9.2.15	Experiment 15: expandingsch5_1.tex	238
9.2.16	Experiment 16: expandingsch5_2.tex	239
9.2.17	Experiment 17: expandingsch6_1.tex	239
9.2.18	Experiment 18: expandingsch6_2.tex	240
9.2.19	Experiment 19: expandingsch7_1.tex	240

9.2.20	Experiment 20: expandingsch8_1.tex	241
9.2.21	Experiment 21: expandingsch8_2.tex	241
9.2.22	Experiment 22: expandingsch8_3.tex	242
9.2.23	Experiment 23: expandingsch8_6.tex	242
9.3	Evaluation of Schema Calculus Expansion	243
9.3.1	Evaluation of the #1 Question	243
	Failed Experiments	244
	expandingsch7_2.tex	244
	expandingsch3_10.tex	245
	Impossible Specifications to be Expanded	246
	expandingsch1_20.tex	246
	expandingsch6_3.tex	247
	expandingsch6_4.tex	247
	expandingsch8_4.tex	247
9.3.2	Evaluation of the #2 Question	248
	Case In-Sensitive in SAL	250
	A Range of Numbers	253
	A Mismatch in the Function Application	254
	Redeclaring State or Global Variables	254
9.3.3	Evaluation of the #3 Question	255
9.4	Conclusion	257
10	Conclusion and Future Work	259
10.1	Thesis Summary	259
10.2	The Main Contribution of Our Research	260
10.3	Relating Research Outcomes to Research Objectives	261
10.4	Future Work	262
10.5	Finally	263
A	The SAL File of Unique Allocator Specification	270
A.1	The SAL File of The Original Specification	270
A.2	The SAL File of The Fifth Abstract Model	271
B	The Counter-Example of The Fourth Abstraction	272
C	Full Z Specifications from Related Chapter	274
C.1	shop.tex	274
C.2	telephonenetwork.tex	276
C.3	hotelspecguestcomps.tex	277
C.4	club_horz.txt	278
C.5	counterMod4.tex	279

C.6	uniqueAllocator.tex	279
C.7	carspark.tex	280
C.8	birthdaybook.tex	280
C.9	paperexample.txt	281
D	The States Animation of telephonenetwork.tex	283
E	JFlex Specification: Lexer.flex	287
F	BYACC/J Specification: Parser.y	305
G	fHead.tex and output_fHead.tex	327
H	expandingschema_2.tex and its expanded schema	328
I	expandingsch2_4.tex and its expanded schema	330
J	expandingsch5_2.tex and its expanded schema	331
K	expandingsch6_1.tex and its expanded schema	332
L	expandingsch7_1.tex and its expanded schema	333

List of Figures

2.1	ϕ holds in the next state on a path	40
2.2	ϕ eventually holds on a path	40
2.3	ϕ always (globally) holds on a path	41
2.4	ϕ holds until ψ holds on a path	41
2.5	The directed graph of an LTS	42
2.6	Abstraction based on equivalent classes	48
2.7	Transition system of the abstract model	57
2.8	Error message from Z2SAL	72
3.1	The Architecture	77
4.1	The JFlex scanner generator	95
6.1	Finalising expansion process	167
6.2	Finalising expansion process (continued)	168
6.3	Finalising expansion process (continued)	169
6.4	Finalising expansion process (continued)	170
7.1	Support for Model Checking Z Specifications	172

List of Tables

2.1	Equivalence of LTL and King's notation	44
2.2	Model checking with verbosity 3 on original specification . . .	55
2.3	Model checking with verbosity 3 on abstracted specification . .	56
3.1	Equivalence of temporal logic notations	80
3.2	Execution Time of Embedded Theorem Experiments	85
3.3	Details of Execution Time of Experiment #6	86
4.1	A list of unspecified Z rules	100
4.2	Precedencies and associativity of Z operators	102
8.1	Several Experiments with the Redefinition System	181
8.2	Summaries of Experiments	182
8.3	Frequent Usages of Generic Constant (GC) Names	213
8.4	Summary of a Usage of Generic Constant (GC) Names	214
8.5	Summaries of Checked Files	216
8.6	Sizes of Z Specifications	218
9.1	Details of Several Experiments with the Expansion System . .	222
9.2	Several Experiments with the Expansion System	243
9.3	Other Experiments with the Expansion System	249
9.4	Other Experiments with the Expansion System (continued) . .	250
9.5	Sizes of Z Specifications	256
9.6	Sizes of Z Specifications (continued)	257

Chapter 1

Introduction

This section introduces this research briefly which relates to the Z language, research motivation, objective and contribution, a structure of our thesis and publications of the research.

Our current life is surrounded by computer applications. Although those applications are used in almost every aspect of our life, how they perform their jobs accurately, particularly one which relates to safety-critical system is a need. This requirement related to formal analysis.

Formal analysis can be staged into three parts [26]: modelling which uses formal specification, specifying which specify properties in formal language and verifying which checks properties formally. A brief description below gives us methods to write specifications in order to model systems.

Previously, natural language and graphics were used to draw systems flowcharts and to write specifications. However, natural language is inadequate for writing specifications due to its imprecision. An alternative, which was the use of a programming language to write a specification, is equally flawed in that it forces one to work at the wrong level of abstraction [56].

A method for writing a specification should be precise enough as well as implementation free. Moreover, if the method is equipped with a proof theory, it can help us to describe properties of specifications easily by conducting 'rigorous arguments' [56].

It raised a need for a certain level of formality and for specifications to be written at a suitably high level of abstraction. Thus, a mathematical notation is used, which is based on the set theory, logic, function, and relation to write those specifications. Notations used to do this are called specification languages or *formal methods*.

Z is one example of formal languages and its use in academia and industry has increased extensively. It is because Z is used successfully to address a large variety of problems and the international standard was also designed

for this language [56].

The set theory, on which Z is based, is adequate to build a more complex data structure, which is necessary in designing a specification [56]. As a formal language, the use of Z could make a specification more formal, precise, and free from ambiguity. In addition, it allows a specification to be analysed mechanically [35].

Designing a specification of a system, which is expected to take considerable time and effort in the software engineering, will ease a verification of the system in the early stage of its development and could avoid high cost in its implementation and test phases, if the specification is designed correctly [56, 73, 75].

Therefore, a specification is crucial for a system, especially the one that relates to the safety of property and life [56]. Indeed, although their use is not widespread in every sphere, formal methods are recommended by many standards bodies concerned with Safety-Critical systems [73].

Following is a discussion on verification. However, there is a lack of tools for this language, especially in model checking Z specifications. Although the Community Z Tools (CZT) project is developing continuously a set of open source tools for Z, its progress is slow [21].

There are many causes of the shortage of Z tools, which mostly are related to its language and semantics, such as its inherent expressiveness, and the difficulty in deciding effectively any theorem about its specifications [21, 35]. Malik et al argue that with less tools support for Z specifications, only a few of them can be used in validating the intended meaning of such Z specifications [48]. Jackson argues that the richness of this language might be the issue in verifying it [35].

It is found that model checking can be used to verify Z specifications as model checking is a verification method that is an automatic, model-based, property-verification approach, intended for use on concurrent, reactive systems, and is stemmed as a post-development methodology [35, 34]. One does not need expertise in mathematical disciplines to model check such specifications [13].

Model checking performs a verification process starting with a model which is described by a user, and discovers whether the hypotheses asserted by the user are valid on that model. This uses an exhaustive searching of the state space of such a system using suitable graph algorithms [13, 54]. If the model checker cannot satisfy those hypotheses, counter-examples consisting of execution traces will be produced. This automatic generation of counter traces is an important tool in the systems design and debugging [13].

Furthermore, if such a model checker were to exist for a Z specification, it has drawbacks. The two principle ones being that it only applies to finite

state systems, and even then these cannot be too large since it can suffer from *state space explosion* problems [13, 54, 51]. Such an explosion is shown to be the most challenging problem in model checking [13, 54, 51].

Smith and Winter [68] report that a Z specification can consist of complex predicates as well as a large number of or even infinite state spaces. As a result, run out of memory could be experienced. Our experiments such as ones which are given in Appendix C.2, C.3 and ones which are discussed in this thesis are examples of that error. As a result, researchers such as Jackson, Smith and Winter [35, 68] proposed an approach of abstraction to a Z specification.

The less supporting tools for the Z language and/or mentioned issues above in mind, makes researchers suggest an alternative method, which is a quick approach to reuse and adapt the existing tools. This quick approach has advantages [10]: users can continue to work with existing tools that they have known and cost to reuse existing tools is less expensive than to build new tools.

Researchers at the University of Sheffield implemented the Z2SAL translator, which uses the SAL model checker to model check Z specifications. A brief introduction to Z2SAL and SAL are given in Section 2.5.1 on page 61 and Section 2.4 on page 49.

On the other hand, there exist several model checking techniques for integrations of the Z language and other languages. For example, model checking CSP-Z by using FDR (Failures-Divergences Refinement) model checker [52]. This language is a semantical integration of both languages in a way that CSP is responsible for the concurrent of a system and Z handles data structure of the system [52]. Mota and Sampaio [52] reuse and link theories and tools to model check CSP-Z specifications. Using the integrated language, they have shown that the language is more expressive power and flexibility than singular approaches [52]. Their work has major limitation on the use of CSP_M data types and channel expansion [52].

Other example is model checking for combination of Z and Statechart [10]. Bussöw uses the existing tool, SMV, to model check specifications. Several translators or adaptors were developed to support Z, Statechart and input language of SMV (Symbolic Model Verifier).

Another example is *Circus* [26] which is a model checking and theorem prover for programs in combination of model based specification language Z, the process algebra CSP (Communicating Sequential Processes) and specification statements of refinement calculi. Freitas built *Circus* from scratch due to the difficulty to find suitable existing verification tools which support both languages. Another reason is Freitas intends to do both model check and theorem proving.

In line with the quick approach, our research relates to Z2SAL which is to support model checking for Z specifications. Based on our experiments with Z2SAL, Z2SAL supports many tags of Z , but not all. Furthermore, several of the generated SAL could not be verified or simulated by the SAL tools.

The following section describes our research motivation.

1.1 Research Motivation

One tag that is not supported by Z2SAL is a generic construct. Z2SAL could not translate specifications that consist of generic constructs and error files were generated instead. Our finding is that Z2SAL cannot recognize a generic constant which is one of generic constructs, though it has been declared in the generic constant definition; Z2SAL treated a generic constant as a new identifier.

Z2SAL has not encountered any generic construct on Z specifications beforehand, so this part of Z has not been implemented yet. Since then, our assumption is that the current Z2SAL does not support a translation of either a generic constant or a generic schema. Although, Z2SAL could implement them some time during our research on this redefinition of a generic constant.

Based on our exploration on the SAL literature itself, a generic form cannot be found either. Thus, another assumption is that Z2SAL does not support a generic constant in order to be consistent with the SAL language.

Other Z tags to be considered are tags which relate to schema calculus. Z2SAL supports a translation of several schema calculus such as a schema inclusion, Δ operator, and Ξ operator, but they must be specified either vertically or horizontally in a schema. However, if a new schema is specified as being constructed from earlier schemas, Z2SAL does not support this schema construction. Thus, it is assumed that Z2SAL does not support schema calculus.

Generic constants and schema calculus are decided to be studied further afterwards to support Z2SAL so this tool can translate both of them.

1.2 Research Objective and Contribution

Being specified using generic parameters, a generic constant is commonly used in formulating mathematical tool-kit operators [2], in which these operators do not depend on the particular type of elements in its construction [69]. Another usage of a generic constant is to specify a general notion which is used frequently in a system.

In a case there is no generic constant, several equivalent functions should be formulated because each function is dedicated to one set of types of parameters; it is a redundant work. Thus, a generic constant is quite beneficial to a Z specification.

Thus, our objective is to implement a tool which will redefine a generic constant definition to an equivalent axiomatic definition based on usages of this generic constant. This redefinition is called an actualization process, in which a generic type of a parameter will be actualised to its actual type of a parameter. Our approach originated from a similar behaviour between a generic constant definition and an axiomatic definition: they declare a global variable inside a Z specification.

Another objective is to implement a tool to construct a new schema by expanding other schemas, in which they are connected by schema operators. The constructed schema is used commonly to define a more complex, modular and a huge specification of a system. Schemas that have been specified can be reused to specify a new schema. It is since every schema has its distinctive operation in a specification, called a 'schema separation' [56].

Both these tools are implemented in a system which is called support for model checking Z specifications. This system is our contribution to broaden the applicability of model checking Z specifications. JFlex [43], BYACC/J [33], and Java language [19] are software which were used to implement our system.

The following section is a thesis structure. It will describe briefly the whole picture of this thesis.

1.3 Thesis Structure

This thesis is divided into three parts. The first part relates to an introduction to this research, backgrounds of this thesis and previous research related to our research. The second one discusses our research, which begins with descriptions of the research and more explanation about it. The last one gives our experiments with the implemented system, evaluations on it, discussions about the results and future works.

The structure of this thesis is as follows:

- Chapter 1 Introduction, which introduces problems related to this research. This section is based on the previously published abstract or paper ([65, 66, 64]). It discusses also our research motivation, objective and contribution and the structure of this thesis. The last discussion on this chapter is publications related to this research.

- Chapter 2 Background consists of five literature sections, beginning with formal method, the Z notation, followed by model checking, the SAL tools, and the Z2SAL translator. Section 2.1 on page 20 offers a brief introduction to formal method and gives several formal languages briefly. Section 2.2 on page 22 describes briefly the Z notation, which begins with the introduction to the Z language, components which Z is based on, operator schemas, an example of a Z specification, some available Z tools and ends with a conclusion. Section 2.3 on page 39 relates to model checking; it contains temporal logics, which are used to formulate a theorem to prove that the related system satisfies this theorem. It also describes binary decision diagram, which is used by a model checker as a search strategy in verifying a theorem. The section before the last discussion on this section is about abstraction which offers a discussion about applying this technique to a Z specification (see 2.3.4 on page 47). Conclusion ends this section. The next section relates to the SAL tools. It begins with a glance at SAL, SAL components, the SAL environment and conclusion. The last section is about the Z2SAL translator which can be grouped as one of the Z tools. This section introduces this translator and gives a brief description about the current Z2SAL and two unsupported Z aspects. Most part of this chapter is based on the previously published abstract or paper ([65, 66, 64]).
- Chapter 3 Translation of Embedded Theorems in Z Specifications: A Proposed Method, which discusses our proposed method in translation of embedded LTL theorems in a Z specification.
- Chapter 4 Implementing A Z Scanner and Parser, it consists of discussions on how our Z scanner and parser were implemented. These two generators are preliminary processes before a redefinition of generic constants or an expansion of schema calculus is performed. Both generators were also included in our paper ([63]).
- Chapter 5 Redefining Generic Constants discusses a design and implementation of a redefinition of generic constants. Several parts of this chapter can be seen in [63].
- Chapter 6 Expanding Schema Calculus discusses a design and implementation of a schema calculus expansion system. Several parts of this chapter can be seen in [63].
- Chapter 7 Integration among the Scanner, the Parser, and Java Programs; describes our method to integrate our separate systems as dis-

cussed in the earlier chapters. For this purpose, another Java program has been built. This Java program which is named `Z_Preprocessing_Tool.java` is our main program.

- Chapter 8 A Generic Constants Redefinition, discusses further our re-definition system. This chapter includes several experiments with this system and an evaluation of this system. Several parts of this chapter can be seen in [63].
- Chapter 9 A Schema Calculus Expansion, discusses another type of our support for model checking Z specification which is support for schema calculus. Several experiments with this system and an evaluation of this system are discussed on this chapter. Several parts of this chapter can be seen in [63].
- Chapter 10 Conclusion and Future Work, which is the last chapter in this thesis, summarises the thesis and defines some future works.

Several appendices which give details of particular chapters are given in appendices sections. It contains several sub-sections from A to L. This thesis was created by using L^AT_EX [74].

1.4 Publications

The following are publications made during this research. They can be either papers or posters.

1.4.1 Peer Reviewed Journal Paper

There is one journal paper which was produced from this research. This paper, which title is "A Pre-processing Tool for Z2SAL to Broaden Support for Model Checking Z Specifications", is under publication in AISC. This paper is an extended version of our conference paper in [63].

1.4.2 Peer Reviewed Conference Papers

Some papers are given as follows:

1. Siregar, M.U. and Derrick, J. Using Abstraction in Model Checking Z Specifications. In The University of Sheffield Engineering Symposium Conference Proceeding Vol. 1. USES 2014 - The University of Sheffield Engineering Symposium, 24 June 2014.

2. Maria Siregar and John Derrick. An Investigation to the Use of Abstraction in Model Checking Z Specifications. In The 9th Annual South East European Doctoral Student Conference Proceeding, pages 330-345, September 2014.
3. M. Siregar, J. Derrick, S. North, and A. Simons. Experiences using Z2SAL. In Advanced Computer Science and Information Systems (ICACISIS) Conference Proceeding, pages 225-231, 2014 International Conference on, October 2014.
4. M. Siregar. Support for Model Checking Z Specifications. In The 4th IEEE International Workshop on Formal Methods Integration (FMi) 2016 which is part of IEEE 17th International Conference on Information Reuse and Integration (IRI), July 2016.

1.4.3 Posters

Several posters were produced as follows:

1. An A1 sized poster whose title is "*Z2SAL*" has been submitted to the Research Retreat of Department of Computer Science, University of Sheffield on May 23rd, 2013.
2. An A1 sized poster whose title is "*An Investigation to the Use of Abstraction in Model Checking Z Specification*" has been submitted to the Research Retreat of Department of Computer Science, University of Sheffield on May 22nd, 2014.
3. This A0 sized poster which is titled "*Using Abstraction in Model Checking Z Specification*" has been accepted to the USE Engineering 2014.
4. An A0 sized poster has been accepted to the York Doctoral Symposium 2015. Its title is "*A Support for Model Checking Z Specifications*".

In the next section, necessary background is described, and it begins with a brief introduction to formal method.

Chapter 2

Background

This chapter describes a brief introduction to formal method, the Z notation, and a shallow description of some related literature to this study. It also gives a glance to model checking, the SAL tools, and the Z2SAL translator. Several examples will be added as they are required. Let us move to formal method which has been introduced briefly in the previous chapter.

2.1 Formal Methods

A formal method (formal technique) is a mathematical foundation based method, which can be explained mathematically. It also means that a user of this method should express his/her descriptions, prescriptions and specifications formally, which at the end the user is able to reason formally on the expression [7].

A formal method requires a formal specification language. A formal specification language has syntax, semantics and proof system which are also formal [7].

Formal specification languages can be classified into three categories [7]:

- **Model-oriented Specification Languages:** Use mathematical constructions such as sets, Cartesians, lists, functions, etc. to specify the specification in a direct way. It results an abstract model specification. The specification includes model type; invariant properties of model; each process consists of name, parameters and return values; and pre- and post- conditions. Languages fall into this category are VDM-SL, Z and B.
- **Property-oriented Specification Languages:** Use logical properties of the specification to express it. It expresses the properties in a declara-

tive way. It results an algebraic specification. The specification consists of two parts: syntax and axioms. For example: OBJ3, CafeOBJ and CASL.

- Mixed of both model-oriented and property-oriented specification languages, such as RAISE specification language (RSL).

Several of those specification languages will be described briefly below. It begins with Z.

2.1.1 Z

Z, which is based on simple set theoretic notions, can handle simple state-oriented sequential problems elegantly and traditionally [7]. It has also been extended such that it is able to express concurrency and objects.

Since Z is one of main topics of this thesis, a further description will be given in a separate section below, though it is not a very extensive discussion. The section will offer several features relate to this formal specification language.

2.1.2 VDM

VDM stands for the Vienna Development Method. It developed by Cliff Jones and Dines Bjørner at IBM's Vienna Laboratory on programming language definition in the 1970s [37]. This specification language might represent the first '*full-fledged*' formal specification language concept [7]. It also comprises an approach to refine specification into code.

This language was standardized by ISO in the 1990s. It has two syntaxes: based on ASCII (most of VDM tools) and based on mathematical symbols.

VDM has been used in the development of electronic trading systems, secure smart cards and the Dutch flower auction systems [37].

2.1.3 RAISE Specification Language (RSL)

RSL has compositionality of object-oriented and concurrency of CSP (communicating sequential processes) [7]. It has been extended also with several features such as timing and duration calculus. RSL is the closest to discrete mathematics to some extent. It is also able to express the imperative specification style and concurrency. It allows for introduction of sorts, postulation of observer function which are shaped by its axioms. RSL structures its specification in a modular fashion the same as Z, B, CafeOBJ and CASL.

2.1.4 CSP

Communicating Sequential Processes is a language for describing patterns of interaction [59]. This language is based on an elegant, mathematical theory, and a set of proof tools. Fortunately, literature on this language is quite extensive.

It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi, based on message passing via channels. Thus, it could be categorized as property-oriented specification language.

A 1977 paper by Tony Hoare described CSP at the first time [32]. It then evolved substantially matured by its practical application in industry as a tool for specifying and verifying the concurrent aspects of a variety of different systems. For example, T9000 Transputer and a secure e-commerce system. Its theory is also being the subject of active research, including work to increase its range of practical applicability.

2.1.5 Conclusion

As described above, formal method offers formal technique in designing a specification of a system which can be reasoned formally later. There are many projects applied to this method. However, formal method is not a panacea for all of our problems. Although it could detect failure in a specification in an early stage of a development of a system and thus could avoid loss of life and money, it should be applied appropriately.

Let us move to the further discussion on Z.

2.2 The Z Notation

Z is a notation to describe computing systems [38]. The terminology of computing systems is used since Z is used to model both hardware and software.

The Z notation, which is based on set theory and first-order predicate logic, is a '*model-based notation*' [38]. A system will be modelled by representing its *state*.

The scope of the set theory is a standard set operator, set comprehensions, Cartesian products, and power sets. There are also other contents such as the Z relation, the Z function, the Z sequence, and the Z bag. Together, they develop one aspect of Z, which is based on set theory and the first-order predicate logic. [75].

The second aspect is how to organize structurally the mathematics in a specification [75]. A particular novelty is the use of what are called *schemas*

which are used to collect mathematical objects of Z and their properties. Specifically, the state of a system and its changes are described in a schema language, as well as its properties and refinement. Z can be differentiated with other formal notations by the existence of schemas [38]. Due to its unique type, an algorithm for type checking every mathematical object can be written, and therefore there are type-checking tools to support the practical use of Z [75].

The third aspect is the usage of natural language [75]. Mathematics objects are related to the real world by means of natural language. Therefore, a Z specification is a mixture of a formal part, which describes precisely the system, and the informal text in natural language, which makes the specification comprehensive and easy to read.

The last aspect is refinement [75], which is the process of moving from an abstract design to a more concrete one. In Z , refinement is supported by constructing another model (in Z), in which particular design decisions have been made and is thus closer to an implementation.

In addition to **Z language** and **mathematical tool-kit**, Z has **schema calculus** which serves as rules for combining individual schema into a complete specification [3].

Z is not a programming language, so it is not an executable notation (although there is an executable subset). In order to allow functionality of Z to work properly, a translation of a specification into an executable form is needed and that process is called '*animation* or *prototyping*' [73].

Z was conceived in the late 1970's, and developed through the 1980's in collaborative projects between Oxford University and industrial partners, such as IBM and Inmos (a semiconductor producer) [38]. It has been applied successfully on several large projects by the above institutions on CICS system, a series projects by Praxis Critical Systems and the security verification of Mondex electronic purse by NatWest Bank. In 1989, the first Z reference manual [69] was widely published. Meanwhile, an international Z standard was completed in 2002.

The main elements of Z , beginning with sets, will be described as follows.

2.2.1 Sets

In mathematics, a collection of similar objects is called a set; For example, the collection of post graduate students who belong to the VT research group, a collection of books about the Z notation, or a collection of nursery rhymes.

There is another method to describe a set, which is by listing all its contents inside curly brackets. For example: {Chris, Sina, Jo, Victor, Maria}, {the way of Z : practical programming with formal methods, an introduction

to formal specification and Z, Z: a beginner's guide, using Z: specification, refinement, and proof}, or {Mary had a little lamb, Twinkle-twinkle little star, Star light star bright}. Each of these examples presents the same set as given in the first method above to define a set. In the second method, the order of members is not important. Thus, they represent the same set though the order of their members is different, as long as their members are the same. In addition, it is impossible to refer to the first, the last, or any order of the contents of set. Moreover, elements in a set are not repeated, thus to have repeated elements in a collection of objects, a sequence or a bag can be used to define it. Both a sequence and a bag are included also in Z.

However, the second method to describe a set is not practical, especially the set who has many members. For such a case, the easier method is simply to give them descriptive names. Those sets can be defined as follows:

- VTGroup - a set of post graduate students on VT lab
- ZBooks - a set of the Z notation books
- NURSERYRHYMES - a set of nursery rhymes

Furthermore, some sets have been assigned specific names, such as "Z" for a set of all whole numbers, "N" for a set of natural numbers, "∅" for a set with no members, and others. Based on these default names, objects are classified into a set according to their types. For example, all integers can be classified into the set called "Z", and a floating-point number cannot be added into that set since the latter number has a different type with the set "Z". Due to this feature, these types can be used to detect errors and inconsistencies in a Z specification [49].

Other aspects of a set will be given in this thesis, and will be described as they appear. The next is a glance of the First-Order Predicate Logic.

2.2.2 The First-Order Predicate Logic

It is assumed that the reader is familiar with the propositional logic. Because of the limitations of the *propositional logic*, most specifications use the *predicate logic* which is also called the *first order logic*.

For example, a proposition in English taken from [34]: "*Every student is younger than some instructors*", will be defined as an atom, say p , in the *propositional logic*. However, 'the finer logical structure' of this English proposition is not visible in this framework. This English proposition will be presented as the same as if it is changed into: "*Every student is younger than every instructor*".

By using the *predicate logic*, both these English propositions can be defined differently and finely. Thus, if there are predicates as follows:

$S(x) : x \text{ is a student}$
 $I(x) : x \text{ is an instructor}$
 $Y(x, y) : x \text{ is younger than } y$

, then the first English proposition will be formulated as:

$$\forall x(S(x) \Rightarrow (\exists y(I(y) \wedge Y(x, y))))$$

On the other hand, the second one is formulated as this:

$$\forall x, y(S(x) \Rightarrow ((I(y) \Rightarrow Y(x, y))))$$

Thus, the predicate logic is more expressive than the propositional logic, which is its ancestor. Due to this expressiveness, propositions in the predicate logic are more complex than those in the propositional logic [34].

As in the propositional logic, there are two properties of an object in the predicate logic. First is denoting the objects particularly. For example, a and p (referring Alan and Patrick), $n(a)$, $h(x, y)$, and others that are considered as objects. These all objects are called *terms*.

Terms constitute variables, constant symbols and functions applied to those [34]. In the Backus Naur Form, terms are:

$$t ::= x \mid c \mid f(t, \dots, t)$$

where x ranges over a set of variables **var**, c over a nullary function symbol in F , and f over those elements of F with arity $n \neq 0$.

The Backus Naur Form (BNF) for the predicate logic is [34]:

$$\phi ::= P(t_1, t_2, \dots, t_n) \mid (\neg \phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\forall x\phi) \mid (\exists x\phi)$$

where $P \in$ predicate symbols of arity ≥ 1 , t_i are terms over function symbols and x is a variable. The binding priorities for those operators are:

- "¬", "∀" and "∃" bind most tightly;
- Then "∧" and "∨";
- Then "→", which is right associative.

Second is denoting truth values or *formulas*. Every term has its truth-value as well as terms combined with at least one operator.

The following sub-section offers brief introductions to several schema operators in Z. All these operators are implemented in our system which will be discussed later.

2.2.3 Schema Operators

The description of schema operators in Z is followed by examples. It begins with the disjunction.

Disjunction

One schema can be combined with another schema by using the disjunction operator " \vee ". This operation can be performed if both schemas have the same types for all variables whose names are the same. This is a precondition for disjunction. A resultant schema will have all declarations from both schemas. Not only is the declaration part obtained, but the resultant schema also gets also predicates from both schemas. However, the predicates are formed from a combination of predicates from the first schema and the second one by a disjunction operator.

An example taken from [57] is given as follows:

$$\begin{array}{c} \overline{S} \\ x, y : \mathbb{N} \\ \hline x + y = 2 \end{array} \qquad \begin{array}{c} \overline{T} \\ y, z : \mathbb{N} \\ \hline y + z = 4 \end{array}$$

If a new schema U is specified as $U \hat{=} S \vee T$, then U has the below form:

$$\overline{U} \\ x, y, z : \mathbb{N} \\ \hline x + y = 2 \vee y + z = 4$$

The above schema is a resultant schema from a combination of the schema S and T by using a disjunction operator.

Conjunction

Another schema operator is conjunction; its behaviour is similar with a schema inclusion operator. By using the same examples as above, a definition of $S \wedge T$ can be used to generate another new schema, V:

$$\overline{V} \\ x, y, z : \mathbb{N} \\ \hline x + y = 2 \wedge y + z = 4$$

Implicitly, each line of a predicate part is separated by an AND operator. This operator has the operation which is similar with previous operator (OR). The difference is on how predicates are combined. In this operator, each predicate part is joined with a " \wedge " operator.

Negation

Applying this operator to a schema yields the same schema but in a negated constraint or predicate. Thus, if there is the \mathbf{S} schema which was taken from [75] shown as follows:

S	_____
$a : A; b : B$	_____
P	_____

The negation of \mathbf{S} , $\neg\mathbf{S}$, is as follows:

a	_____
$: A; b : B$	_____
$\neg P$	_____

However, before a predicate part of a schema can be negated, this operator requires the schema to be in a normalised form. For example: if the above \mathbf{V} schema is negated, normalisation will first occur in this schema.

Normalisation is a method to rewrite a schema so that all constraints are listed in a predicate part. These constraints, named implicit predicates, restrict the declared variables' values that these values are obtained from subsets of the complete types. Furthermore, the declaration part of this schema will be modified to its canonical form [75].

A canonical form specifies a unique representation of every mathematical object. The canonical declaration is a declaration of pure types, known as a signature [2]. Moreover, a predicate part which has been added with implicit predicates is called a property of a schema. A schema which is written in terms of a signature and a property is a normalised schema.

If a normalisation is not performed, several unacceptable results might be obtained when a negation is required to be performed. For example: a declaration of a function from A to B in a schema can be rewritten into its abbreviated form which has the type of the set of the pairs of A and B . Moreover, predicates to express the functionality of this function will be added also into the predicate part of this schema. Then, a negation operation requires us to negate predicates on the predicate part. As mentioned earlier, as a default, each line of a predicate is connected by a conjunction operator. Thus, when these predicates are negated, conjunctions are changed into disjunctions. Inevitably, such a process might produce predicates which do not express the functionality of the previous function.

Several examples taken from [56] showing how normalisation works can be seen as follows:

$$\text{OneToFortyNine} \hat{=} [n : 1..100 \mid n < 50]$$

Its normalised form is:

$$\text{OneToFortyNine} \hat{=} [n : \mathbb{Z} \mid 1 \leq n \wedge n \leq 100 \wedge n < 50]$$

If this schema is negated, its negated schema might be as follows:

$$\text{notOneToFortyNine} \hat{=} [n : \mathbb{Z} \mid 1 > n \vee n > 100 \vee n \leq 50]$$

If normalisation is not performed before a negation operation, the negated schema is:

$$\text{FiftyToHundred} \hat{=} [n : 1..100 \mid n \geq 50]$$

This result is not expected as a negation from the first schema. This simple example makes the previous function problem is clearer.

Implication

This operator is rare to use in a schema calculus definition. Implication on schemas results a merged schema in which the signatures are merged, and the properties are joined by a logical implication [2].

As formulated in propositional logic, an implication operator is equivalent to a disjunction between a negated form of the left hand side proposition and the right hand side one:

$$S \Rightarrow T \equiv \neg S \vee T$$

Thus, there are two options in implementing a schema implication. The first option is to use directly this operator, whereas the second is to use its equivalent form.

If the latter option is used, the merged schema is obtained by negating the first schema and combining it with the second schema by using a disjunction operator. The first schema must be normalised before it is negated.

Bi-implication

This operator merges two schemas by merging their signatures and joining their properties by using a logical equivalence. As with the previous schema operator (implication), there are also two options for its implementation.

The first option is to use this operator to combine two schemas. The second one uses an equivalent proposition of the bi-implication operator. If the second option is used, the equivalent form of $S \Leftrightarrow T$ is $(S \Rightarrow T) \wedge (T \Rightarrow S)$. The operational procedure of the second option is given follows:

- Follow procedure of an equivalence form of each implication form as given in previous operator.
- Combine them by using a conjunction operator.

Renaming

Another operation in a schema calculus is renaming, "/ . By using this operation, variables can be renamed explicitly [57]. The way a user uses this operator is based on this definition: *writing the schema name followed by a sequence of pairs of new name/old name which is put inside a pair of square brackets.*

If there is a definition of $\mathbf{S}[\mathbf{a}/\mathbf{x}]$, which \mathbf{S} is the same as the schema used in the disjunction operator, and this operation is used to define a new schema, let us say \mathbf{S}' , then \mathbf{S}' is:

$$\frac{\frac{S'}{a, y : \mathbb{N}}}{a + y = 2}$$

As can be seen from the above schema, the new schema \mathbf{S}' is obtained from \mathbf{S} [57] by first renaming all \mathbf{x} s in the schema \mathbf{S} into \mathbf{a} and declaring the renamed variable as well as other variables of the schema \mathbf{S} in the new schema \mathbf{S}' . Afterwards, a predicate part of the renamed schema is also put in the new schema, but the renamed variables in this predicate part must be considered.

This operator works by finding schema whose name is used in the renaming operation. Afterwards, all of its variables and predicates will be recalled. Having this information, all occurrences of old variables will be tracked and be changed or renamed to the new specified name of variables in the renaming operator. This action is reflected also in usages of the old variables in the predicate part.

Hiding

Hiding, "\ , is another example of a schema operation. This operation can erase variables that are less relevant to a particular system. A list of variables which is not interesting any more will be declared after a symbol "\ . Declarations of these variables will be erased from the schema and their existences in the predicate part will be quantified existentially there. Thus, another name for schema hiding is schema existential quantification [75].

Using the same schema example as used on the above disjunction, if there is a definition of $\mathbf{S} \setminus (\mathbf{x})$ which specifies the new \mathbf{S}_- ' ' schema, this new schema is as follows:

$$\frac{\frac{S_-''}{y : \mathbb{N}}}{\exists x : \mathbb{N} \bullet x + y = 2}$$

Since \mathbf{x} is a natural number typed variable which its minimum value is 0, the suitable value for \mathbf{y} can be found. Therefore, \mathbf{x} can be erased permanently from the predicate which gives us a compact schema as follows:

$$\frac{\frac{S_-''}{y : \mathbb{N}}}{y > 0 \wedge y \leq 2}$$

For the translation purpose, it seems that this operator is quite difficult to operate, especially for finding the right value for the quantified variable. It requires user guidance to perform the task.

Schema Composition

A schema composition of two schemas, $S \text{ ; } T$, is obtained by passing the output of S as an input to T . This operator consists of several processes as follows:

- All matched variables that are primed in S and non-primed in T are renamed to common names.
- Join these two renamed schemas by using a schema conjunction operator.
- Hide all common variables
- Simplify the schema

By using the same \mathbf{S} schema as used for disjunction on the earlier subsection, let us specify the new \mathbf{W} schema as follows:

$$\frac{\frac{W}{x', z : \mathbb{N}}}{x' + z = 6}$$

If both schemas are operated by a schema composition operator as $\mathbf{W} \text{ ; } \mathbf{S}$, the new \mathbf{X} schema as a result of this operation is as follows:

X
$y, z : \mathbb{N}$
$z - y = 4$

The above schema is obtained after simplification performed on a predicate part of this schema.

Quantification

By using this operator, a user can quantify over several variables of a schema while other variables are unchanged. There are two types of quantification operators. The first operator is a universal quantification " \forall ", whereas the second one is an existential quantification " \exists ".

If the same **S** schema as the one used for disjunction is used here as an example and it is copied as shown below:

S
$x, y : \mathbb{N}$
$x + y = 2$

then, a universal quantification on the **x** variable is specified as the new schema, **C**, as follows:

C
$y : \mathbb{N}$
$\forall x : \mathbb{N} \bullet x + y = 2$

and an existential quantification on the same variable as above is specified as the new **D** schema as follows:

D
$y : \mathbb{N}$
$\exists x : \mathbb{N} \bullet x + y = 2$

The above **C** schema has a predicate that always returns false. It is since there is no **y** that makes the addition of it to every natural number get 2. On the other hand, the **D** schema, which looks like the **S**' schema in the hiding operator, can be simplified. The simplified schema is similar to the schema given for the hiding operator. Indeed, an existential quantification is also called hiding [75]; to hide variables from a declaration part of a schema.

2.2.4 A Z Notation Example

To know Z better, one simple example of our experiments, **club.tex**, was taken. This specification describes a club that somebody can join as a member and then can leave it.

In a similar fashion to programming languages, every variable or constant is defined as having a type. Moreover, suppose at the time of writing the specification the precise value for such a variable is not necessary or in other words focus is on its essential, that variable could be defined as a **basic type**, which is written within the square brackets in the Z notation [57, 49]. This feature is called *abstraction* [49] to abstract the details that are not essential for a specification.

In the given specification there is only one basic type which is PERSON. No further details are known about PERSON as for this specification, they are not relevant. Furthermore, this type can be used throughout the specification to declare other variables with that type. Thus, the scope of a basic type or a given type is global in that system. The declaration of this type is made as follows:

[PERSON]

In this specification, a **free type**, MESSAGE, with only one value, OK, is defined. By declaring a variable as a free typed variable, it can ease a user to describe recursive structures, such as list or trees [69]. Another usage of a free typed variable is to enforce that variable as a global constant or variable [73]. In our example, there was a declaration as follows:

MESSAGE ::= OK

Variables and predicates will be discussed later on, and these are combined into a structure called a **schema**. In this thesis, a schema is drawn as an open box and is given a name. It has two partitions separated by a horizontal line. Written as such, a schema is defined vertically, and there is an equivalent horizontal definition, which is written with no box. This specification but in horizontal definition can be seen in Appendix C.4.

In the **state schema** of our example, **members** variable was declared as a set of PERSON, which size must be less than or equal to 3. Since then, one specific PERSON can be referred at some time as needed [49].

Based on these declarations, each declaration has two parts. *First* is the word right after the colon, let us say *rhs*, which contains a type or the name of a set. *Second* is the left hand side word, say, *lhs*, which is a name of the set, called **variable**. Thus, a declaration has a form as follows:

lhs : *rhs*

It is needed to define a maximum number of elements set can hold in order to have every set still is in the finite state. The scope of one variable is only inside the schema where it is declared, exception is for state variables. Variables of a state schema are imported into other schemas so they can be used in the specification globally. Variables can change their values which means that they can change system states [57]. A state can be imagined as the contents of a system's memory.

The relationship between variables are defined by writing **predicates** and they can act as system invariants [57] if they are defined in the state schema. In our running example, the system invariant is 'the size of members must be less than or equal to 3'. In this predicate, the cardinal operator, "#", was used to count how many elements a set has. As these explanations, our first schema, the state schema, which was given a name, *Club*, is as follows:

<i>Club</i>
<i>members</i> : \mathbb{P} <i>PERSON</i>
$\#members \leq 3$

Everything which are declared above a horizontal line are **state variables** or **objects** and below this line are **state predicates** [57]. One possible state for our state variable is:

members == {*person1*, *person2*, *person3*}

As a comparison, if that state schema is defined horizontally, it can be defined as:

$Club \hat{=} [members : \mathbb{P} PERSON \mid \#members \leq 3]$

Variables within a state schema must be initialized in an initial schema to assert that system invariants are fulfilled [57]. In our specification above the *Club* specification, **members** will be initialized to empty set. Furthermore, this initial schema includes a declaration *Club*' which is known as *schema inclusion*. Schema inclusion is one of decoration in Z. Doing so will benefit one to [49]:

- Concentrate on just a few things at a time
- Keep schemas small and simple
- Re-use pre-written schemas in different contexts or situations

Let us go back to our specification, the above declaration means that this schema has all variables and predicates, which are declared previously

in the *Club* state schema. Moreover, that variable was ended with *'*, which means variable after operation. Thus, it means **members** after this schema is operated will have a value of an empty set.

Thus, our initialisation was given as follows:

<i>Init</i>
<i>Club'</i>
<i>members' = ∅</i>

In addition to state and initialisation schemas, a *Z* specification has also operational schemas. By using these schemas, it enables us to report on a system's state and describe changes in the state schema [49].

Our specification had two operational schemas. They will be described one by one in the following paragraphs.

First, the **JoinOK** schema was operated to allow someone to join the club. This schema might change the values of state schema variables, which is indicated with a Δ symbol preceding *Club* in the declaration part of this schema. The Δ symbol is other decoration in *Z*. Another decoration but does not exist in this example is Ξ symbol which will not change states of a system.

There was also an input variable, **name?**, whose type is **PERSON** which is a person who wishes to join the club, an output variable, **sumC!**, of type natural number, which will record how many members this club has, and another output variable, **reply!**, with type **MESSAGE**. It is a convention on *Z* that every variable which is ended with symbol **?** is an input variable and which is ended with symbol **!** is an output variable.

In the predicate part of this schema, the person who wants to join this club should not be a member of that club. This is presented as a not membership operator, " \notin ". Furthermore, the number of persons in the club should not exceed the maximal number. If both of these conditions are met, then that person is added to that club. The set union operator, " \cup ", is used to do this addition, which will unite one set and another set in a bigger set. This operation will change the value of **members** and this happened after **JoinOk** operation is conducted. It is also necessary to update the **members** information so it contains all the person being added. The size of this club or how many person on this club is recorded on **sumC!**. Finally, **OK** message is generated, which indicates a process of joining the club is successful. These predicates are combined with a conjunction or **AND** or an operator " \wedge ". Logically **AND** will give true if all the predicates are true. Since this operator is a default operator in the predicate part of a schema, a " \wedge " can be used to present a " \wedge ". However, a precaution is not to use " \wedge " to present a " \wedge " inside a quantifier block.

Thus, our operational schema, `JoinOk`, is as follows:

<i>JoinOk</i>
$\Delta Club; name? : PERSON; sumC! : \mathbb{N}; reply! : MESSAGE$
$name? \notin members$ $\#members < 3$ $members' = members \cup \{name?\}$ $sumC! = \#members'$ $reply! = OK$

Second, an operation for a member who wants to leave this club was formulated in the `LeaveClub` schema. Based on its name, this schema might change the state schema variables values. In this schema an input variable, `name?`, whose type is `PERSON` was declared. This variable was assigned with the name of club members, who wishes to leave that club. There is also an output variable, `sumC!`, whose type is a natural number and contains the number of members of the club after leaving process is performed. A person can leave that club if that person is a member of that club. That operation is to test whether such a person belongs to this club. It is achieved by using a membership operator, " \in ". If it is the case, that member will be deleted from the club. Operator used here is the " \setminus " operator, which is called the **Set Difference**. Remember that this operator works on a set, so `name?` should be enclosed by curly brackets. Finally, as part of this operation, the sum of club members will be calculated. All of these give us the following:

<i>LeaveClub</i>
$\Delta Club; name? : PERSON; sumC! : \mathbb{N}$
$name? \in members$ $members' = members \setminus \{name?\}$ $sumC! = \#members$

This is a very brief introduction to `Z` and its major components. For further definitions and examples, the reader can refer to `Z` literature. In the next section, survey of the available tools for the `Z` specification notation is offered.

2.2.5 The Z Tools

To date various tools were developed and introduced to support the `Z` language. They are: *troff* [41], *CADiZ* [40], *fuZZ* [70], \LaTeX [44], *Z/Eves* [60] and [61], the Community `Z` Tools (CZT), *Isabelle/HOL-Z* [9], *ProofPower* [1], *The Jaza Animator* [71], *Z Word Tools* [29], and *Fastest* [14].

Descriptions of these tools are as follows:

troff

This is a programmable text formatter and due to its flexible fundamental tools it supports a large varieties of formatting tasks. *troff* also

supports phototypesetting on a Graphic Systems phototypesetter and user-designed document styles [41].

CADiZ

This is a UNIX based suite of tools designed to check and typeset Z specifications, as well as preview and investigate interactively of Z properties. For checking, it will check the syntactic, scope and type correctness. Since *troff* is the base for tools on processing documents in UNIX, *CADiZ* is integrated with such a tool. The specifications, which are correct syntactically, might be previewed on a bit-map screen. Furthermore, a mouse is used to investigate interactively their properties [40]. To the date, it has abilities to do also the domain checking, the schema expansion with a redundant term elimination, and the interactive theorem-proving using heuristics and proofs entered by the user and it has evolved towards the ISO-Z standard [21].

fuZZ

This consists of formatting and printing tools for Z specifications, also a checker of Z types and scopes. The package consists of two parts, the first part consists of a style option, which is used to type-setting \LaTeX systems, the second one consists of a program which can analyse and check specifications [70].

\LaTeX

Having hundreds of citations or over a thousand authors in the bibliography is not a problem if \LaTeX is used for preparing documents [44]. It is now used widely on preparing documents in which mathematical formulas are used extensively.

Z/Eves

By using this tool, a Z specification can be analysed in a variety of ways [60]. This tool has abilities such as the domain checking, the interactive theorem proving using heuristic and proof entered by the user [21], and the schema expansion with redundant term elimination.

The Community Z Tools(CZT)

The Community Z Tools project is an open source project providing an integrated tool set to support Z, with some support for Z extensions such as Object-Z, Circus and TCOZ. Recently, the CZT with their projects has developed a parser and a type-checker for Z, an Java AST package for use in third-party modules, and a number of other proposed modules, this includes cross-language translators and model-checkers [21]. It is planned also to have a Z animator.

Isabelle/HOL-Z

HOL-Z is a proof environment for Z built as plug-in of the generic theorem prover Isabelle/HOL. It allows importing Z specifications written in L^AT_EX and type-checked by the Java-based ZeTa-System.

ProofPower

ProofPower is a suite of tools supporting specifications and proofs in Higher Order Logic (HOL) and in the Z notation. It has been under ongoing development since 1989. It was originally designed and implemented by International Computers Ltd. and now was undertaken by Lemma 1 Ltd.

Z Word Tools

These tools are used to enable the usage of Z specifications within a document written using Microsoft Word processor. It consists of several functions such as editing and checking a Z specification. It needs other programs so its functionalities work better, such as Java runtime for working with the standard Z, *Graphviz* for generating diagrams and *fuZZ* for using the stand-alone converter.

The Jaza Animator

It is an animator for the Z formal specification language. It can be used to validate Z specifications by evaluating Z expressions, testing Z schemas against example data values, and executing some of Z specifications.

Fastest

Fastest is a model-based testing tool. It receives a Z specification and generates test cases in almost automatic way. Currently, it provides only limited functionality for a test case refinement into C and Java.

However, there are certain drawbacks, especially in verifying Z specification, associated with these tools, which were developed from scratch. Malik et al. argue that with these support, only few of them can be used in validating the intended meaning of such Z specifications [48]. Therefore, Martin argues that the issue of tool support is the main obstacle in using widely Z specification [50]. Plagge and Leuschel argue that Z is a 'high-level formalism specification' and this might be the cause why Z has limited industrial tools [55]. Derrick et al argue that to build a model checker directly for a Z specification can take considerable effort due to the abstraction of the language [22]. Jackson argues that the richness of this language might be the issue in verifying it [35].

With these issues in mind, it suggests an alternative method for supporting applications of Z, which is to adapt existing tools such as automated checking tools [21]. In most recent studies, there is *ProZ* [48] which was developed from *ProB* [55], data refinement verification [8] which uses Alloy SAT-solver based counter-example, and Z2SAL [20] which is a translator from a Z language specification into a SAL language specification [18].

The translation of Z to an input of Alloy Analyser allows Alloy Analyser generates an instance of Alloy specifications and model checks that specification. By this translation, it enables 'immediate visual feedback' of such a specification for its designer and this quick response provides a worthwhile verification of Z specifications.

Not only could Alloy Analyser model check the Alloy version of Z specifications and other input language specifications, but also it could be used to model check temporal properties specified in Alloy specifications directly to some extent. It is based on research of Vakili and Day in [72]. Their idea is to use the transitive closure operator to specify conducive conditions for the set of states in order to satisfy a property [72].

Z2SAL is intended to use SAL as a model checker and refinement tools for Z specifications, which are translated into the SAL input language. This paper works on this translator and model-checking tools.

One of our works relates also to a proposal of specifying temporal properties directly in Z specifications. It is because several operators to specify temporal properties in LTL and CTL have been provided for Z and Object Z languages [42]. Thus, a user could specify a Z specification which also includes temporal properties.

As mentioned in Chapter 1, several model checkers which can model check combinations of Z and other languages have also been built. They are *Circus* [26], model checker of Z and Statechart [10] and model checker for CSP-Z [52].

2.2.6 Conclusion

Z is a language that can be used for writing a formal specification of a system, and at its heart, it is based on mathematical notation and logic. Using this language, specifications can be clear and concise, thus can avoid ambiguities that can occur when notations that more informal are used.

There is a range of tool support available for the language, which is briefly reviewed above. Based on that description, the availability of more support for model checking Z specifications is a need.

Now let us move on to the specifics of model checking, which begins with the introduction and leading up to abstraction on Z specifications. It will be

followed by the section relates to the SAL model checker, which is a tool-set that can be used to model check specifications written in the SAL language.

2.3 Model Checking

Model checking which is discussed in this thesis is a classical model checking which is based on temporal logic, rather than a refinement model checking which uses automata theoretic methods. These classifications of model checking are mentioned in [26].

The model checking method is an appropriate choice when compared with methods relying upon simulation, testing and deductive reasoning [13]. Expertise in mathematical disciplines is not needed to model check such specifications [13].

Although it has advantages, there are also drawbacks, the two principle ones are it only applies to finite state systems, and even then these cannot be so large since it can suffer from *state space explosion* problems [13, 54, 51]. These are due to the search strategy, which uses an exhaustive searching of the state space of a system using suitable graph algorithms [13, 54], which is adopted in its verification.

The next section gives a brief description to the temporal logic. By using this logic, properties which need to be verified are formulated.

2.3.1 The Temporal Logic

The temporal logic is useful for specifying concurrent systems, since they can describe events in ordered time, regardless of time explicitly [13]. The properties used in model checking are based on the temporal logic [34] and this was introduced by Clarke and Emerson in their algorithm of temporal logic model checking. In that algorithm, a formula can be true in some states and false in other states dynamically. This is quite different from the propositional logic and predicate logic, a formula is always true or false in a model instead.

According to the particular view of time, the temporal logic can be divided into the linear time logic and the branching time logic. In linear time logic or *Linear-time Temporal Logic* (LTL), a time is a set of paths, and a path is a sequence of time instances. However, in the branching time logic or *Computation Tree Logic* (CTL), a time is represented as a tree, rooted at the present moment and branching out into the future. Both these logics are supported by the SAL model checker; both of them are introduced here, beginning with the LTL.

Then LTL has the following syntax given in the Backus Naur form:

$$\phi ::= \top \mid \perp \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (X\phi) \mid (F\phi) \mid (G\phi) \mid (\phi U \psi) \mid (\phi W \psi) \mid (\phi R \psi)$$

where p is any propositional atom from some set of atoms. The connectives X , F , G , U , W and R are called *temporal connectives*. X means *neXt* state, F means some *Future* state, and G means all future states (*Globally*). The next three symbols, U , W and R are called *Until*, *Weak-until* and *Release* respectively.

That grammar means as follows:

- \top (true), \perp (false), \neg (not), \wedge (and), \vee (or), \rightarrow (implies) are boolean connectives.
- Connectives \Leftrightarrow (equivalent) can be defined by using combination of above connectives.
- $X \phi$ means that ϕ holds in the next state on a path. Assumed that ϕ is initially on the first state, that formula is shown in Fig. 2.1 on page 40:

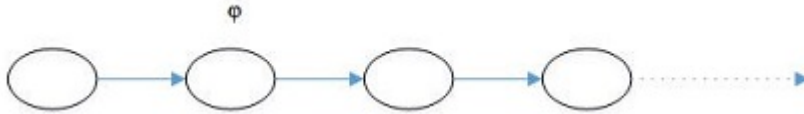


Figure 2.1: ϕ holds in the next state on a path, adopted from [34]

- $F \phi$ means that ϕ eventually holds on a path, it is shown in Fig. 2.2 on page 40:

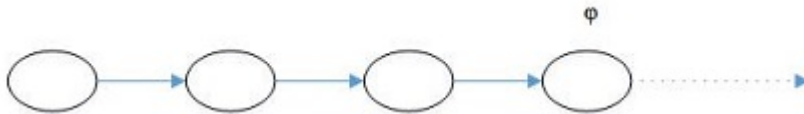


Figure 2.2: ϕ eventually holds on a path, adopted from [34]

- $G \phi$ means that ϕ always (globally) holds on a path, it is shown in Fig. 2.3 on page 41:
- $\phi U \psi$ means that ϕ holds until ψ holds on a path, it is shown in Fig. 2.4 on page 41:

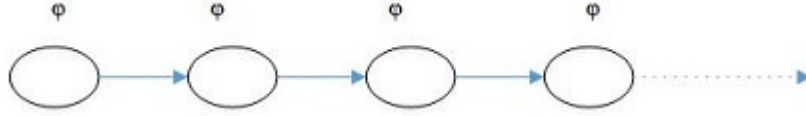


Figure 2.3: ϕ always (globally) holds on a path, adopted from [34]

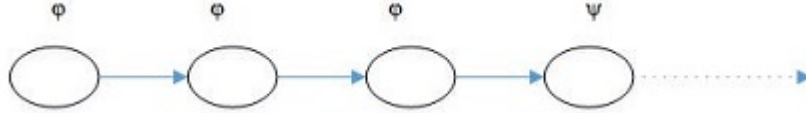


Figure 2.4: ϕ holds until ψ holds on a path, adopted from [34]

- $\phi W \psi$ is a weak variant of $\phi U \psi$

As in the propositional logic and predicate logic, to avoid ambiguities and abundant brackets, there are binding priorities. They are \neg , X , F , G , U , R , W , \wedge and \vee , \rightarrow with decreasing order.

Therefore, previous examples can be written without any loss of ambiguity as:

- $F p \wedge G q \rightarrow p W r$
- $F (p \rightarrow G r) \vee \neg q U p$
- $p W (q W r)$
- $G F p \rightarrow F (q \vee s)$

LTL formulas are interpreted over labelled transition systems which gives the semantics of a specification language of a system [54]. The LTS is a structure (S, \rightarrow, L) where [34]:

- S is a (finite) set of states
- \rightarrow is a binary relation on S , the transition relation
- $L : S \rightarrow 2^P$ is a labelling function

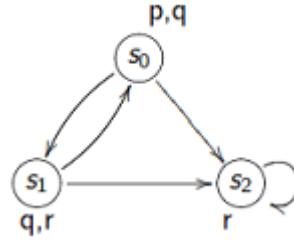


Figure 2.5: The directed graph of an LTS, adopted from [34]

LTSs can be drawn as a directed graph. An example of an LTS is shown in Fig. 2.5 on page 42, which was taken from [34] as follows:

It represents these properties:

$$\begin{aligned}
 S &= s_0, s_1, s_2, \\
 \rightarrow &= \{(s_0, s_1), (s_0, s_2), (s_1, s_0), \\
 &\quad (s_1, s_2), (s_2, s_2)\}, \\
 L(s_0) &= p, q, \\
 L(s_1) &= q, r, \\
 L(s_2) &= r
 \end{aligned}$$

LTL theorems which correspond to previous LTS are as follows:

- $s_0 \models p \wedge q$ and $s_0 \models \neg r$
- $s_0 \models Xr$ and $s_0 \not\models Xq$
- $s_0 \models G\neg(p \wedge r)$ since p and r never hold at the same state
- $s_2 \models Gr$
- $s_i \models F\neg q \rightarrow FG r$
- $s_0 \models GFp \rightarrow GF r$, but $s_0 \not\models GF r \rightarrow GF p$

LTL formulas are evaluated on paths. A state of a system satisfies an LTL formula if all paths from the given state satisfy it. Therefore, LTL implicitly and universally quantifies over paths. Meanwhile, the properties which assert the existence of a path can also be represented using the LTL in the negated form of the property. However, the better way to represent an existence is by using the CTL.

In addition to the temporal operators of the LTL, there are also quantifiers A and E in the CTL, which express all paths and a path that exists, respectively. The CTL formulas using the BNF are:

$$\phi ::= \top \mid \perp \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (AX\phi) \mid (EX\phi) \mid (AF\phi) \mid (EF\phi) \mid (AG\phi) \\ \mid (EG\phi) \mid (A(\phi U\psi)) \mid (E(\phi U\psi))$$

The propositional connectives have the same meaning as defined in the LTL grammar. Other connectives, which are combined with those quantifiers, represent meaning as follows:

- $AX\phi$ means that ϕ must hold for all children in a tree
- $EX\phi$ means that ϕ must hold for some children
- $AF\phi$ means that ϕ must eventually hold on all paths in a tree
- $EF\phi$ means that ϕ must eventually hold on some paths in a tree
- $AG\phi$ means that ϕ must always hold on all paths in a tree
- $EG\phi$ means that ϕ must always hold on some paths in a tree
- $A(\phi U\psi)$ means that, along all paths in a tree, ϕ holds until ψ does
- $E(\phi U\psi)$ means that, along some paths in a tree, ϕ holds until ψ does

As per the same directed graph as shown previously (see Fig. 2.5 on page 42), CTL theorems for that LTS are as follows [34]:

- $s_0 \models EX (q \wedge r)$ and $s_0 \not\models AX (q \wedge r)$
- $s_0 \models \neg EF(p \wedge r)$ since p and r never hold in the same state
- $s_2 \models EG r$
- $s_0 \models AF r$
- $s_0 \models E((p \wedge q) U r)$
- $s_0 \models A(p U r)$
- $s_0 \models AG(p \vee q \vee r \rightarrow EFEG r)$

Table 2.1: Equivalence of LTL and King's notation

<i>Temporal Logic</i>	<i>Style of King</i>
G	□
X	○
F	◇

Following is a table shows equivalences of LTL and King's notation [42]. Table 2.3.1 only shows equivalences for the three most frequent symbols of temporal logic used by a user.

To our **club** specification (see Section 2.2.4 on page 32) can be added some LTL theorems such as:

- $\models G \neg (\# \text{ members} = 3)$

To check that the club never gets full

- $\models G \neg (\text{members} = \emptyset)$

To check that the club never gets empty

It could be applied also to CTL theorems as follows:

- $\models G \exists (m, n: \text{PERSON}): m \neq n$

To check that there is a member who is different

- $\models G \neg (\exists (m, n: \text{PERSON}): m \neq n)$

To check that it is not the case that there is a member who is different

- $\models G \forall (m, n: \text{PERSON}): m \neq n$

To check that every member is different

- $\models G \neg (\forall (m, n: \text{PERSON}): m \neq n)$

To check that it is not the case that every member is different

The following section describes briefly about binary decision diagram.

2.3.2 Binary Decision Diagram

There are two strategies in designing model checking [51], either global or local. If the former is used, the structure of the property will be re-cursed and each of its sub-formula will be evaluated on all paths of that system.

On the other hand, in the latter one, only some of the paths of that system will be re-cursed. CTL and other branching-time logics are in the former strategy, whereas LTL is in the latter one.

CTL and LTL are introduced earlier (see Section 2.3.1 on page 39). Other model checking algorithm that re-curses a system globally is branching-time fixed point logic, $\mu\mathbf{TL}$. Another algorithm in this category is *Ordered Binary Decision Diagrams* (OBDDs) and is called symbolic model checking.

A set of states, which satisfies the properties, are presented symbolically in symbolic model checking [34]. Every state as well as relation will be assigned a Boolean value, and they are encoded by using binary values. However, since a relation is a subset of a state to other state, that encoding needs two copies of Boolean vectors. The SAL model checker is a model checker that uses symbolic model checking (see Section 2.4.1 on page 49).

BDD is an alternative for presenting a Boolean function [34]. The diagram is a binary tree on its simpler form. All of its non-terminal nodes are labelled with Boolean variables, and its terminal nodes or its leaves are labelled with Boolean values, "T" for true or "F" for false, sometimes "1" is used instead of "T" and "0" is used instead of "F". It has two edges at maximum, in which one edge is drawn as a dashed line which represents a variable whose value is 0, another edge is drawn as a solid line which represents a variable's value of 1.

As far as their sizes are concerned, binary decision trees are quite similar with truth tables of the same Boolean functions [34]. A binary decision tree can have at least $2^{n+1}-1$ nodes, which is almost the same size as the size of a truth table 2^n . Thus, a binary decision tree is not more efficient as a truth table. However, on such truth tables, lines that have the same behaviour, in other words have the same values, could be combined and this will minimize the truth table's size. This can be adapted to a decision tree and this will yield a decision diagram for such a decision tree.

A procedure to obtain a decision diagram is as follows [34]:

- Removing duplicate terminals. At the end of this procedure, the decision diagram has just two terminals. To preserve the same meanings, all edges which point to "0" are redirected to point to only one terminal "0". The same process as above is applied also to "1".
- Removing redundant tests. If a node n has more than one outgoing edges and they point to the same node m , delete this n and redirect its incoming edges to point to m directly.
- Removing duplicate non terminals. If there are two nodes that have the same structures of sub-BDDs rooted on each of both nodes, one of

them can be eliminated and all incoming edges of a deleted node are redirected to point to another node.

If none of that procedure's item can be applied to BDD then it is truly the reduced BDD [34]. BDD satisfies function if a 1-terminal node is reachable from the root along a consistent path in that BDD. A path is consistent if and only if from a node there is only a dashed line or solid line leaving such a node, not both of them.

Given BDDs B_f and B_g , an operation \cdot can be performed by taking BDD f and replacing all its 1-terminals by B_g . For the same BDDs, operation $+$ can be obtained by replacing all 0-terminals of B_f by B_g . Moreover, the complementation operation \neg can be obtained by replacing all 0-terminals in one BDD by 1-terminals and vice versa.

OBDDs are BDDs which have an ordering on its variables, which can interfere with the size of OBDD. OBDDs have a *canonical form* which means that for one boolean function, it will give us a unique reduced OBDD [34]. Further information about OBDD can be obtained from literature.

The next section gives a brief description about abstraction. This technique is then applied to Z specifications as per [68].

2.3.3 Introduction to Abstraction

It is suggested that a state explosion in model checking will limit the applicability of such model checking, especially in software verification [13]. Such a state explosion can occur in the usage of memory and time [54].

Pelanek in [54] identifies several basic approaches for avoiding or preventing a state space explosion problem, such as:

- **Minimizing the size of explored states.** This can be obtained by using abstraction, reductions based on equivalences (for example the cone of influence, symmetry or partial order), or compositional methods.
- **Minimizing the size of memory needed for storing the explored states.** Several techniques are storage size reduction, or symbolic representation (such as using binary diagram decision).
- **Enlarging the availability of memory, for example distributed environment or magnetic disk.**
- **Moderating completeness and exploring only part of state space instead, such as heuristics or randomization.**

Previous studies report that abstraction is the most important technique for reducing the state explosion problem in model checking [12, 53, 30, 15, 28, 16, 13, 27, 11, 68].

The abstraction technique is used widely in many researches, such as Clarke et al in their work of an approximation of the concrete system to reduce the complexity of a temporal-logic model checker [12]. Their abstract model can be used to verify the original system and this model can be used to represent a system with a large number of states, which are over 10^{1300} states.

The abstraction technique is an approach that is well applied either for finite state systems [12, 47, 15, 46] or infinite ones ([5, 28, 23, 27, 53, 30]. Furthermore, if model checking is combined with abstraction and/or induction principles; this technique can be applied for infinite systems [13]. Most of those studies related to creating a direct abstract model, which is not begun from the concrete system.

By using abstraction techniques, a system will be modelled by a small system. Thus, if some properties hold for the abstract model, these will hold in the concrete or original model [15].

The abstraction technique was then proposed to be applied to a Z specification by Smith and Winter in [68] and Jackson in [35]. The next section describes about this.

2.3.4 Abstraction on Z Specifications

More recently, literature has emerged that Z specifications can be complex enough to be verified thus abstraction is offered onto that specification [68]. However, a major problem with that research is the lack of practicality, though Smith and Winter started it in [68]. Thus, one of our aims is to investigate the use of abstraction in model checking Z specifications.

Abstraction which refers to [68] groups states in the concrete model into equivalent classes, then mapping these classes by an abstraction function to states in an abstract model, as shown in Fig. 2.6 on page 48. The abstraction function is described in many papers, such as mapping or equivalence relation in [12] and others. However, one that offers a systematic method is [28], Smith and Winter referred to that method.

In order to prove LTL theorems in a Z specification, such a specification must be restricted to have totality operations [68]. This means every operation must define a post-state which in some cases it is an error state.

Smith and Winter proposed this work as per [68]: create monomials from atomic predicates of properties to be proven, form equivalent classes of the concrete model, define an abstraction function and relate those classes to a

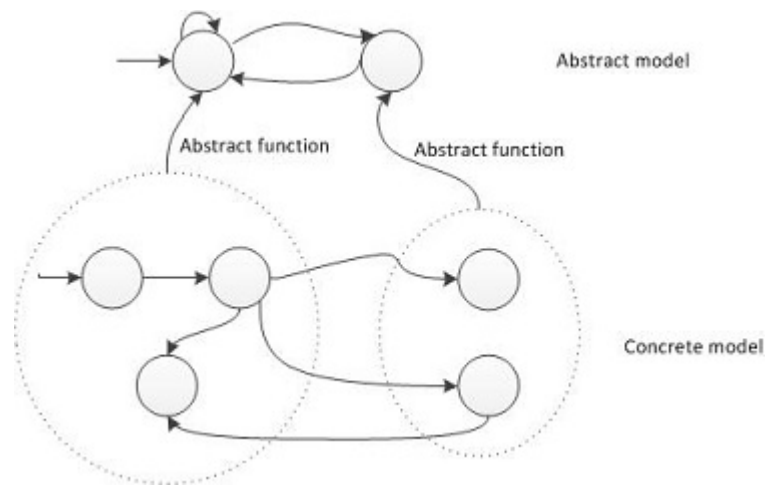


Figure 2.6: Abstraction based on equivalent classes, adopted from [68]

single abstract state. An example in how to apply their method in other Z specification has been shown in our extended abstract [65].

2.3.5 Conclusion

Model checking is found to be useful since it provides a full verification of a finite state system without the user having sophisticated knowledge. Originally applied in hardware systems, it is now commonly available for applications in software systems.

One of the drawbacks of model checking is that it applies to finite state systems, since it works by performing a complete state space exploration. However, the size of the systems that model checkers can now cope with has increased rapidly over the years, especially when other techniques are applied onto it, such as abstraction.

An example of applying an abstraction technique on a Z specification has been performed by us in another paper. This example shows us that the original Z specification can be modelled based on the property that needs to be satisfied. The next chapter will discuss further benefits of this abstract model. This manual work can be put as a future work as how to automate it.

Following is the discussion regarding to the SAL tools. The same example that has been used on the abstraction technique will be used also in the next section.

2.4 The SAL Tools

SAL is a framework that is used to change perceptions and implementations of model checkers and theorem provers, which at first was based on verification to a calculation of properties or symbolic analysis such as abstraction, slicing and composition [5]. It also combines some different tools such as abstraction, program analysis, theorem proving and model checking towards a symbolic analysis of transition systems [18].

The SAL language can be used as a specification language, a target language for some translators or a common source of some analysis tools. It originated from a collaboration of two researchers, David Dill from Stanford University and Thomas Henzinger from the University of California at Berkeley. It evolved and was included Verimag in this collaboration then it is developed at SRI. The current version is SAL 3.3.

A discussion on the SAL model checker will be given in the following section, begins with a glance at SAL.

2.4.1 A Glance at SAL

The brief description below is about expression language on SAL. Since SAL is intended to accommodate users with various translations of sources languages, its language needs to be liberal [18]. Therefore, it includes a large number of operators.

The SAL's grammar is not case-sensitive the same as its user defined variables. This situation enforces users not to use any of SAL's reserved words as variables, even in different case of characters.

As other computer languages, users can specify a comment in their SAL file. Comments are preceded by a symbol % and terminated by an end-of-line.

The SAL language supports the built-in basic types for Boolean, natural numbers, integers, and integers without zero, real and real without zero. For new basic types, it might be introduced using un-interpreted type declarations. Types in SAL are modelled as a set and they are not necessary to be finite.

An expression in the SAL language consists of constants, variables, applications of Boolean, arithmetic, bit-vector and array operations, function, tuples, and records selection and update. The expression without next variables is called the current expression.

The SAL expression contains two kinds of variables: logical variables and state variables. State variables can be the current variable or next ones. The next variables are almost identical with the current one, but the next ones are differentiated from the current one by an " " in the end.

Let us move on to a brief description of a transition language on SAL. The transition system module consists of a state type, an invariants definition, an initialization condition, and a binary transition relation of a specific form. All of these are defined on a state type, which is defined by four pairwise disjoint sets of input, output, global and local variables, which define the state of the module. Input and global variables consist of observed variables of a module. On the other hand, output, global, and local ones are controlled variables of a module.

Rules on the transition play roles as constraints for the current and next states of the transition. Transitions can be written in the form of definitions or guarded commands.

Definitional transitions are used to specify the values taken on by these controlled variables, which transitions can be independently specified in a simple form conveniently. However, for variables that have similar case structures in its definitions, this transition form is less efficient. This is due to the repeated case structure in each of definitions [18]. For these controlled variables, it is more efficient to write an initialization and a transition using guarded commands.

A definition has a form:

$$Lhs = Rhs$$

, in which the *Lhs* can be an identifier with or without the next variable symbol, ', followed by zero or more access. In contrast, the *Rhs* can be expression or the *IN* expression. Thus, the definition is used to construct an invariant, an initialization, and a transition.

Regarding the guarded command, each of this command consists of a guard formula and an assignment part. A guarded formula is a Boolean expression in the current controlled variables, and in the current and next state of state input variables. On the other hand, an assignment part is a list of equalities, in which its left-hand side is the next state variable and its right-hand side is expression in current and next state variables. The format for a guarded command is:

$$label : guard \dashv\dashv > assignments$$

[18], in which label is a name of the operational schema and this so to aid readability [21].

Variable that is defined in the *Lhs* of a definition cannot be assigned either in a guarded initialization or transition. Another constraint is the initialization cannot contain any next state variable, whereas the assignment part of a transition must have next state variables on its left-hand side and might have next state variables on its right-hand side.

A module in SAL works as follows: each time it is activated, one guarded transition is chosen such that a guarded formula holds in the current (and possibly the next input) state. A transition is a conjunction of associated guarded transition with all definitions of transition sections. If no guard is chosen, it might be a deadlock. Thus, the **ELSE** clause is added to the transition section.

The SAL module is a self-contained specification of a transition system in SAL. Properties can be analysed independently in modules. Moreover, modules can be composed synchronously or asynchronously from base modules to yield a new module.

A base module defines a state type of a set of input, output, global and local variables. It can be divided into some sections. These sections will be described as follows:

The first section is the **DEFINITION** section. It serves as invariants to the system and it is used to define controlled variables, which depend on the inputs.

The second section is the **INITIALIZATION** one. In this section, constraints on the possible initial values for local, global, and output declarations are defined, but inputs might not be initialized. This section determines also a state predicate, which sets the initial state of the related base module. Both methods on writing a transition language can be used in defining this section, but they cannot contain any next variable.

The last section is the **TRANSITION** section. This section constraints the possible next states for local, global, and output declarations. It determines a state relation to the previous state of the module.

The SAL language allows us to manipulate state variables. It has some constructs that can be used to achieve that. For example, to make an output and global variable into local, the construct **LOCAL** is used. Another example is to make a global variable into an output, the construct **OUTPUT** is used. To rename a variable in order to avoid name clashes, the construct **RENAME** can be used and the new variable name is introduced by using construct **WITH**. This renaming can be applied to a declaration of **INPUT**, **OUTPUT** and **GLOBAL**.

Some modules can be combined using module composition which will combine modules either synchronous by using "**||**" or asynchronous by using "**[]**".

Several modules will be collected in a context. More on the SAL's grammar, reserved words, and others, can be found in [18]. The following section is about components of SAL.

2.4.2 SAL Components

As an intermediate language, which serves as a medium for presenting the state transition semantics of systems with their own source languages, SAL was integrated with several loosely coupled back-end components. These components relate to each other by using well-defined interfaces [4].

Interfaces such as PVS, SMV, InVeSt [6] and others, make a SAL specification can be analysed in various basic method given as follows [4]:

Validation.

Related to this method, several adjustments to the SAL specification are taken place first to provide a suitable input for each of these tools, such as:

- SAL contexts are translated into PVS theories to produce a semantic of a SAL transition system. PVS is used to perform a theorem proving.
- SAL modules are mapped to SMV modules by expanding its types and constant definitions into a SMV specification, translating output and local variables of SAL into variables, input variables into parameters, and adding two extra variables to cope with a non-deterministic assignment of SMV to fire a relevant SAL transition. SMV model checks a translated specification of a SAL specification.
- For animation of SAL specifications, not all features of such specifications will be supported by Java compiler. By a translation of a SAL specification into a Java program, this specification can be animated.

Invariant generation.

An invariant is an assertion specified in the predicate part of a state, which will hold of every reachable state of the related transition system. This generation will break up an infinitely states space of a SAL specification into finitely many disjoint control states.

Slicing.

A SAL specification, which has large states space, will benefit from this technique of program analysis. This technique deletes all irrelevant codes of a system to a property under consideration.

Abstraction.

Given a concrete system and an abstract function, InVeSt can generate automatically and compositionally an abstract model of a system [5].

The details about PVS, SMV and InVeSt can be found in the related sources, hence they are not described here. The next section gives us a description about an environment on model checking of this tool in the first part of this section. Another environment on this tool relating to a simulation will be given in the second part of the next section.

2.4.3 The SAL Environment

In this section, only two environments of SAL will be discussed, though SAL can be used for other functionalities. The first one is the SAL model checker. Another one is the SAL simulator. Let us move to the first discussion.

The SAL Model Checker

Regarding model checking, SALenv contains symbolic model checker called *SAL-smc* (*simple model checker*). Users can specify properties in the LTL and the CTL in their specification, which needs to be verified. In addition to *SAL-smc*, *SALenv* also contains *SAL-bmc* (*bounded model checker*) which only supports LTL formulas. By conducting bounded model checking, SAL can search on a state space on a given depth. When a property is invalid, a counter-example will be produced, otherwise, it will be proven.

Typical LTL operators are [17]:

- $G(p)$: always p , which means p is always true
- $F(p)$: eventually p , which means p will be eventually true
- $U(p, q)$: p until q , which means p holds until a state is reached, in which q holds
- $X(p)$: next p , which means p holds in the next state

Some examples of LTL theorems are given as follows [17]:

- $G(p \Rightarrow Fq)$, which means whenever p holds, eventually q holds
- $G(F(p))$, which means p will hold infinitely often
- $G(\neg p)$, which means p never holds

On the other hand, CTL operators in SAL can be in one of these forms [17]:

- $AG(p)$: p holds globally in all paths, which means p is globally true

- EG(p): p holds globally in some paths, which means there is a path, in which p holds continuously
- AF(p): p holds eventually in all paths, which means for all paths, p eventually holds
- EF(p): p holds eventually in some paths, which means there is a path, in which p eventually holds
- AU(p,q): p holds for all paths until q holds, which means in all paths p holds until a state is reached, in which q holds
- EU(p, q): p holds in some paths until q holds, which means there is a path, in which p holds until a state is reached, in which q holds
- AX(p): p holds in the next state for all paths, which means p holds in all successor states
- EX(p): p holds in the next state in some paths, which means there is a successor state, in which p holds

SAL enables a user to describe transition system models and to add properties of this system to it. These properties are written using the temporal logic. Having this specification, it could be verified or checked to know whether this system satisfies those properties. In this case, the SAL model checker can produce two results. The first result is the properties that are proven or in other words they are satisfied. Examples of this are many in this thesis. The second one is counter-examples, which means that these properties cannot be proven or they are violated by this system. An example of counter-example can be seen in Appendix B.

The transition system is a collection of \mathbf{S} states, in which there is a set of initial states \mathbf{I} , and a relation by using " \rightarrow " among these states. Thus, it can be modelled as $\tau = (\mathbf{S}, \mathbf{I}, \rightarrow)$. A state is a valuation for the variables with their types.

In this section, the transition system of the examples used for the abstraction on \mathbf{Z} specifications will be discussed. The first example used is the original specification which can be seen in [68, 64]. The " τ " transition system of that example is defined as follows:

$$\begin{aligned} \tau = (&S = \{used = \emptyset, alloc = \emptyset, sentNum = 0\}, \{used = \{2\}, alloc = \{2\}, sentNum = 0\}, \\ &\{used = \{1\}, alloc = \{1\}, sentNum = 0\}, \{used = \{2\}, alloc = \emptyset, sentNum = 2\}, \\ &\{used = \{2\}, alloc = \emptyset, sentNum = 0\}, \{used = \{1, 2\}, alloc = \{1\}, sentNum = 0\}, \\ &\{used = \{1, 2\}, alloc = \emptyset, sentNum = 1\}, \{used = \{1, 2\}, alloc = \emptyset, sentNum = 0\}, \\ &\{used = \{1\}, alloc = \emptyset, sentNum = 1\}, \{used = \{1\}, alloc = \emptyset, sentNum = 0\}, \\ &\{used = \{1, 2\}, alloc = \{2\}, sentNum = 0\}, \{used = \{1, 2\}, alloc = \emptyset, sentNum = 2\}, \\ &I = \{\{used = \emptyset, alloc = \emptyset, sentNum = 0\}\}, \\ &\rightarrow = \{()\}) \end{aligned}$$

Table 2.2: Model checking with verbosity 3 on original specification

Process	Time (secs)
ast generation for count2	0.0
ast generation for set	0.0
ast generation for uniqueAllocator_mod	0.0
type-checker for count2	0.0
type-checker for set	0.0
type-checker for uniqueAllocator_mod	0.015
module flattening	0.0
ast simplification	0.0
function application expansion	0.0
unfolding quantifiers	0.016
flat module – common subexpression elimination	0.0
flat module → boolean flat module conversion	0.0
monitor generation	0.016
static order	0.0
cluster compression	0.0
flat module → BDD conversion	0.078
verification	0.031
Total	0.156

The relation between states, I , leaves blank since this set has a huge number of pairs of states. As a consideration, the number of states of this transition system is quite big which is 12.

The SAL specification of the original Z specification will be model checked by setting the verbosity which is greater than the default values, 0. This SAL file is a modification version of the one generated by Z2SAL from the Z specification found in [68]. The related SAL specification can be seen in Appendix A.1. Regarding modification, it can be read in [64]. This option lets the SAL model checker to present also the way it performs the verification including the time taken.

```
$ sal-smc -v= 3 uniqueAllocator_mod
```

However, all the information produced from this command are not shown here since it takes 115 lines. A table containing the usage of time for each process is presented in Table 2.2 on page 55.

Thus, from Table 2.2 on page 55, this model checking needs 0.156 second,

Table 2.3: Model checking with verbosity 3 on abstracted specification

Process	Time (secs)
ast generation for uniqueAllocatorAbs_simpl5	0.015
type-checker for uniqueAllocator_mod	0.0
module flattening	0.0
ast simplification	0.0
function application expansion	0.0
flat module – common subexpression elimination	0.0
flat module → boolean flat module conversion	0.016
monitor generation	0.0
static order	0.0
cluster compression	0.0
flat module → BDD conversion	0.078
verification	0.016
Total	0.125

whereas the abstracted model is model checked on 0.125 second. The similar table, but the verification is performed on the abstract model ([68, 64]) with the same level of verbosity is shown by Table 2.3 on page 56. The SAL specification of the abstract model can be seen in Appendix A.2.

```
$ sal-smc -v= 3 uniqueAllocatorAbs_simpl5
```

These experiments were conducted on machine with Intel(R) Core (TM) i5-2320 CPU 3.00 GHz.

As said previously, there are 12 states on the original specification, whereas there are only 4 states on the abstracted model. Since the number of states is quite small, the relation of states can be presented as can be seen in the transition system of the abstract model as follows:

$$\begin{aligned}
 \tau = & (S = \{2 \notin \text{used}, \text{alloc} \neq \{2\}, \text{sentNum} \neq 2\}, \{ \text{alloc} = \{2\}, \text{sentNum} \neq 2\}, \\
 & \{2 \in \text{used}, \text{alloc} \neq \{2\}, \text{sentNum} = 2\}, \{2 \in \text{used}, \text{alloc} \neq \emptyset, \text{sentNum} \neq 2\}), \\
 I = & \{ \{2 \notin \text{used}, \text{alloc} \neq \{2\}, \text{sentNum} \neq 2\} \}, \\
 \rightarrow = & \{ (\{2 \notin \text{used}, \text{alloc} \neq \{2\}, \text{sentNum} \neq 2\}, \{2 \notin \text{used}, \text{alloc} \neq \{2\}, \text{sentNum} \neq 2\}), \\
 & (\{2 \notin \text{used}, \text{alloc} \neq \{2\}, \text{sentNum} \neq 2\}, \{ \text{alloc} = \{2\}, \text{sentNum} \neq 2\}), \\
 & (\{ \text{alloc} = \{2\}, \text{sentNum} \neq 2\}, \{ \text{alloc} = \{2\}, \text{sentNum} \neq 2\}), \\
 & (\{ \text{alloc} = \{2\}, \text{sentNum} \neq 2\}, \{2 \in \text{used}, \text{alloc} \neq \{2\}, \text{sentNum} = 2\}), \\
 & (\{2 \in \text{used}, \text{alloc} \neq \{2\}, \text{sentNum} = 2\}, \{2 \in \text{used}, \text{alloc} \neq \emptyset, \text{sentNum} \neq 2\}), \\
 & (\{2 \in \text{used}, \text{alloc} \neq \emptyset, \text{sentNum} \neq 2\}, \{2 \in \text{used}, \text{alloc} \neq \emptyset, \text{sentNum} \neq 2\}) \}
 \end{aligned}$$

It can be seen also in Fig. 2.7 on page 57.

The next section describes the SAL simulator. The same example will be applied to the SAL simulator.

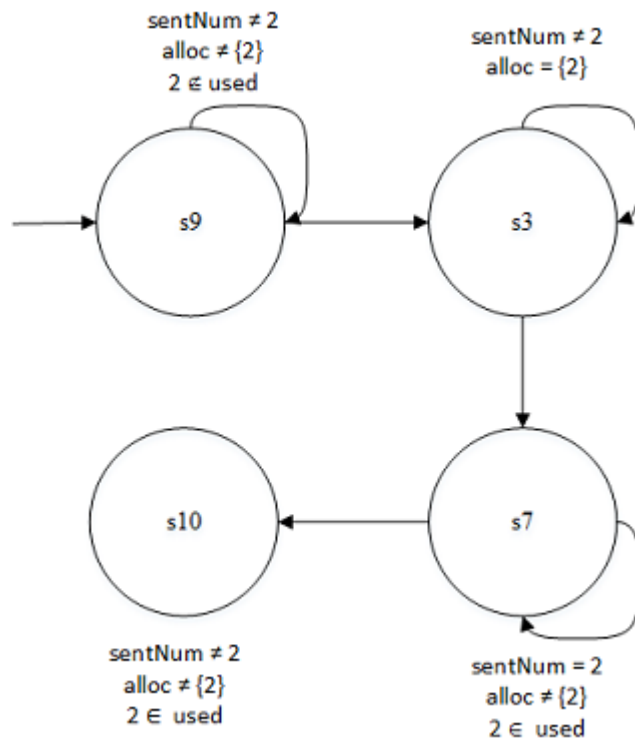


Figure 2.7: Transition system of the abstract model

The SAL Simulator

The SAL environment, (*Salenv*), contains a simulator for finite states specifications based on BDDs, which allows users to explore different execution paths of a SAL specification [17]. By doing such an exploration, users will be more confident of their model before verification is performed on such a model.

The SAL simulator enables a user to use Scheme Programming language to implement a new simulation of repetitive tasks. This simulator is started by the `sal-sim` command. After this command is executed, the system prompt will be changed to the `sal` prompt.

```

$ sal-sim
SAL Simulator (Version 3.3). Copyright (c) 2003-2011 SRI International.
Build date: Mon Mar 18 16:40:07 PDT 2013
Type '(exit)' to exit.
Type '(help)' for help.
  
```

```
sal >
```

A command to import the context, `import!`, that needs to be simulated is entered. As described above, the same example, which is the `uniqueAllocator_mod` specification, will be used here. Thus the import statement is shown as follows:

```
sal > (import! "uniqueAllocator_mod")
uniqueAllocator_mod
```

```
sal >
```

To start a simulation, the below command is used. This time, the module's name is entered.

```
sal > (start-simulation! "State")
```

```
sal >
```

This command initializes the simulation, which composes of a current trace, a current finite state machine, and a set of already visited states. The current trace could be a set of traces since a trace is a list of states.

The `(display-curr-trace)` command will display one of the traces in the current trace, which is initially is a set of initial states. It is shown as follows:

```
sal > (display-curr-trace)
Step 0:
--- System Variables (assignments) ---
used(1) = false
used(2) = false
alloc(1) = false
alloc(2) = false
sentNum = 0
invariant_ = true

sal >
```

To display the set of current states, which is to print at most 10 states as a default, the `(display-curr-states)` command is used. It is shown as follows:

```

sal > (display-curr-states)
State 1
--- System Variables (assignments) ---
used(1) = false
used(2) = false
alloc(1) = false
alloc(2) = false
sentNum = 0
invariant__ = true
-----

sal >

```

However, if it is needed to print at most *num* number of states, the previous command could be defined as `(display-curr-states [num])`.

As can be seen above, there is an index of positive number, which extends a state. This index *idx* can be used to select the intended state by using the `(select-state! [idx])` command.

To perform a simulation step, which will append the successors of the set of current states in the current trace, the `(step!)` command is used. This command updates also the set of current states.

```

sal > (step!)

sal > (display-curr-trace)
Step 0:
--- System Variables (assignments) ---
used(1) = false
used(2) = false
alloc(1) = false
alloc(2) = false
sentNum = 0
invariant__ = true
-----
Transition Information:
(module instance at [Context:  scratch, line(1), column(1)]
(label Request
transition at [Context:  uniqueAllocator_mod, line(26), column(12)]))
-----
Step 1:

```

```

--- System Variables (assignments) ---
used(1) = true
used(2) = false
alloc(1) = true
alloc(2) = false
sentNum = 0
invariant_ = true

sal >

```

As a comparison, a simulation is also applied to the abstract model which its SAL specification is given in Appendix A.2. After three step of a simulation, the original specification consists of nine states, whereas the abstract model has just four states, a slightly less than the half of the original specification's one.

Another example of applying simulation can be seen in Appendix D. This example gives the states animation of **telephonenetwork.tex** specification. As mentioned earlier in Chapter 1, this specification produced run out of memory when it was verified by the SAL tool.

The (**sal/reset!**) command can be used to force garbage collections and reinitializes all data structures which are used by the simulator. It is necessary to perform this command before the simulation of a new module begins.

2.4.4 Conclusion

SAL is a tool that plays as an intermediate language tool and one of its roles is in model checking of system. Due to its capability to model check a transition system, this tool was chosen as an output language for a specification that is generated by the Z2SAL translator.

It is shown that, the original specification needs more time for verification by the SAL model checker on one property than the time needed by the abstract specification, though their difference is less than one tenth of a second. However, the numbers of states of these specifications are quite different as given by the SAL simulator. Indeed, from their transition systems, the original specification has three times of the number of states of the abstract specification.

Therefore, it seems that the abstract model is beneficial in terms of memory and time usage. In other words, it generates less number of states, which needs less memory than the original specification. Furthermore, it can be executed in the less time than the original specification.

The next section will discuss the Z2SAL translator. It can be classified into tools of the Z notation, especially the one that can translate a Z specification into other specification language.

2.5 The Z2SAL Translator

This section gives a brief description to the Z2SAL translator. This description flows from the time it was developed until its current version.

2.5.1 Introduction

The idea of translating Z into the SAL input language was due to Smith and Wildman [67] at the University of Queensland, Australia. However, since the basic idea given in [67], the idea was implemented in a tool set, and the current Z2SAL is extended in a different direction. In doing so, it has also had to tackle optimization issues [21], and thus is quite different from the ideas as originally envisaged.

In providing a translator for the Z language, which will translate a Z language specification into an input language of an existing tool, SAL was chosen since it has similar representation of many aspects of Z [22], such as a module mechanism of SAL represents appropriately a Z states transition system [67]. SAL also supports expressive mathematics, which is a necessity in model checking an expressiveness of a Z specification [67]. Moreover, there exists many different tools that use the SAL input language [21], which are offered freely by SRI under academic licences such that they attract users to engage in international groups.

Z2SAL translates a Z specification into a SAL module. This module will group a number of definitions including types, constant and modules for describing a Z states transition system [22]. A SAL module has a general format as follows:

```

State : MODULE =
  BEGIN
    INPUT ...
    LOCAL ...
    OUTPUT ...
    INITIALIZATION [ ... ]
    TRANSITION [
      ...
    ]
  END

```

Translating a Z specification into a SAL input language needs some adjustments due to some differences of both languages [21]. These adjustments are discussed briefly as follows:

First, it is *bounding the infinite*. Z supports *fully abstract* (non-grounded, non-constructive) specification styles, whereas SAL is a *concrete and grounded language*. For example, Z supports the built-in numerical types such as "Z", "N" and "N₁", whose ranges are infinite. On the other hand, SAL has the similar unbounded types INTEGER, NATURAL and NZNATURAL, which can be used only as base types of finite sub ranges in the actual specification. Z also supports basic types, which have semantics of un-interpreted set, such as [TAPE, NAME]. Therefore, the translations provided by Z2SAL should define a finite number for these sets.

The *mismatched formal paradigms* are the second difference. Z and SAL have very different styles of specifications and descriptions. A Z specification is built-up increasingly, which consists of state schemas and operational schemas. It views locally and functionally such that every operational schema operates on its input and output variables, or on variables of state schemas. In contrast, a SAL specification is created as a 'monolithic finite state automaton' such that all input, output and local variables are compiled into aggregate states and all operations act upon guard transitions from one state configuration to other state configurations [21]. Thus, this mismatch can be approached by re-ordering all information in a Z specification. Another mismatch is Z specifications often use partial functions. It is to express incomplete operations of operational schemas and to express associative data types, *maps* of the state schema, whose sizes are dynamics. As SAL is based on *Binary Decision Diagrams* (BDDs), SAL always needs a representation of the SAL function given as the SAL total function. This means one needs a work-around in order to present a partial function in Z specifications as a total function in SAL. Furthermore, a set cannot be treated as a monolithic of SAL, but as a 'poly-lithic collection of judgements' over its elements instead. Thus, several operations in a set need to be expressed differently, such as the cardinality of a set, which is not supported by SAL.

The last difference is an issue of *non-computable specifications*. A Z specification naturally supports non-constructive styles of a specification. These styles need to be expressed in computable styles of a specification in SAL, which essentially are different. Normally, a SAL specification consists of a series of update assignments to primed variables, which indicates posterior variable states. In contrast to Z, a direction of a constructive approach is not necessary in a Z specification. Z2SAL adopts an assertion of a posterior existence of variables and restricts their values in the precondition. This needs a searching for suitable precondition values.

The next section introduces to the current Z2SAL. It includes also the translation stages, which was implemented on this translator.

2.5.2 The Current Z2SAL Translator

Currently, the tool has two operating modes which it will either translate a single Z specification into the input format of SAL for model checking purposes, or translate a pair of Z specifications for refinement checking purposes [62]. The translated output is placed in the same directory as the source.

Regarding model checking, it is possible to add theorems at the end of this automaton, to check whether certain properties always hold, or eventually hold. However, other aim of our research is to propose embedded theorems on a Z specification. Our proposal can be reviewed in 3 on page 74. More information relating to Z2SAL can be found in the related references.

The current Z2SAL translates the example of 2.2.4 with an order as following. First, Z2SAL will generate three SAL files or three contexts; *club*, *count3* and *set*. Following is a description of the SAL file of the *club* specification and translation stages carried out by Z2SAL.

This SAL file will be given first as follows:

```

club : CONTEXT = BEGIN
NAT : TYPE = [0..4];
PERSON : TYPE = {PERSON_1, PERSON_2, PERSON_3};
MESSAGE : TYPE = DATATYPE
      OK
END;
PERSON__counter : CONTEXT =
count3 {PERSON; PERSON_1, PERSON_2, PERSON_3};
State : MODULE =
  BEGIN
    LOCAL members : set {PERSON;} ! Set
    INPUT name? : PERSON
    OUTPUT reply_ : MESSAGE
    OUTPUT sumC_ : NAT
    INITIALIZATION [
      members = set {PERSON;} ! empty AND
      reply_ = OK AND
      sumC_ = 4
      -->
    ]
    TRANSITION [
      JoinOk :
        NOT set {PERSON;} ! contains?(members, name?) AND
        PERSON__counter ! size?(members) < 3 AND
        members' = set {PERSON;} ! insert(members, name?) AND
        reply_' = OK
        -->
        members' IN {x : set {PERSON;} ! Set | TRUE};
        reply_' IN {x : MESSAGE | TRUE}
    ]
    LeaveClub :
      set {PERSON;} ! contains?(members, name?) AND
      members' = set {PERSON;} ! remove(members, name?) AND
      sumC_' = PERSON__counter ! size?(members)
      -->
      members' IN {x : set {PERSON;} ! Set | TRUE};
      sumC_' IN {x : NAT | TRUE}
    ]
  ]

```

```

        ELSE  -->
            members' = members
    ]
END;
END

```

The name of its *module* is **State** and **club** is a name of *context*. A *context* is used in SAL for providing a framework for declaring types, constant, modules, and module properties [18]. A SAL context is read from left to right, top to bottom, and an entity must be declared before it is used [18].

Z2SAL translates *basic types* of Z into finite and enumerated sets in SAL. By default, this set has three elements and sometimes has an extra *bottom* element [21]. Such the *bottom* element represents undefined element. This element is a useful technique to treat commonly the partial function of Z as the total function in SAL. Three is chosen as a default value since in [36] it was stated that most counter-example could be found in a defined type whose number of instances is three. Another reason why such a small number was chosen by Z2SAL is to prevent a longer time for verification by the SAL model checker.

For our SAL file, it has **PERSON** as a *basic type* and such a basic type is translated as a set of three elements, which are enumerated as follows:

```
PERSON : TYPE = {PERSON_1, PERSON_2, PERSON_3};
```

However, in some cases, a sentinel value is inserted at the end of that range, which means undefined element, *bottom*, as said previously. The *bottom* element indicates that such a type is required in a *context* that needed this element.

The translator generates also *built-in* types into finite sub ranges in SAL. For such an example, type "N" will have range of values given as follows:

```
NAT : TYPE = [0..4];
```

Such a range is chosen by the translator since Z2SAL preserves relations between numerical types, which are given as follows:

$$\mathbb{N}_1 \subset \mathbb{N} \subset \mathbb{Z}$$

Moreover, "N₁" has a default of at least three instances, then "N" will have four instances by including 0 as the lowest number, and "Z" will have five instances. The default strategy adapted in Z2SAL for literal constant is to expand the minimal range to 1 above the highest and 1 below the lowest [21], and it must be the smallest numerical range if it is possible. Thus, our "N" have five instances, whose range is from 0 to 4, by expanding 1 to the highest range. It is because there is a definition of a literal constant, **members**, as can be seen later.

Free type of *Z* is translated into a similar data type in SAL. Thus, the free type from our *Z* example will be translated into SAL declaration as follows:

```
MESSAGE : TYPE = DATATYPE
OK
END;
```

Furthermore, Z2SAL will generate a SAL module for one *Z* specification and its default name is **State**. Z2SAL assumes the first schema defined in a *Z* specification is a state schema and the second one is the initialization schema. Other schemas defined afterwards are operational schemas. Thus, Z2SAL supports only one state schema in a *Z* specification, whereas *Z* supports many state schemas in a *Z* specification as well as SAL supports many modules in a SAL specification respectively.

This module defines a finite-state-automaton (FSA) of club system. States for this FSA are taken from state variables, which are translated as *local* variables in the SAL file. In this example, only one state variable was declared, which is **members**.

```
LOCAL members : set {PERSON;} ! Set
```

Sometimes, translator will generate an extra *local* Boolean variable, **invariant_{__}**, and define its formula in the *Definition* sub-clause. Such a variable serves as an abbreviation for the longer expression and this will be useful, especially if it often occurs in many contexts [21].

The initialization schema will be translated into the similar SAL *Initialization* sub-clause of the *module* clause. In this system, **members** is initialized to be **empty**, which declares **empty** as an initial value of **member**.

Z2SAL uses guarded commands (see Section 2.4.1 on page 49 for its form) on its initialization sub-clause and forces variable bindings to be resolved in the guard. If there is an invariant, it will hold in this part since invariants act as preconditions for entering initial state [21].

```
State : MODULE =
  BEGIN
    LOCAL members : set {PERSON;} ! Set
    ...
    OUTPUT reply_ : MESSAGE
    OUTPUT sumC_ : NAT
    INITIALIZATION [
      members = set {PERSON;} ! empty AND
      reply_ = OK AND
      sumC_ = 4
      -->
    ]
```

Z2SAL translates every *Z* operational schema into two ways. Firstly, it is to translate variables; secondly, it is to translate predicates.

First, it treats every occurrence of either input, output variable or both of them with the similar input, output variable or both and puts them in the beginning of *module* clause. Every input and output is a *local* variable of each operational schema. In addition, Z2SAL replaces every "!" at the end of every output variable with "_".

<pre> JoinOk ΔClub; name? : PERSON; reply! : MESSAGE name? ∉ members #members < 3 members' = members ∪ {name?} reply! = OK </pre>

As an example, in the JoinOK schema, there are an input variable, **name?**, and it will be translated as it is, and an output variable, **reply!**, and it will be translated as **reply_**. Both of them are in the *local* scope of such an operational schema. In order to reduce initial state space, Z2SAL initializes output variables to arbitrary initial values of its range of values. For that operational schema, **reply!** is initialized to value **OK**.

Second, Z2SAL translates a predicate of these schemas into a guarded command and puts it in the *Transition* sub-clause. This is a last sub-clause of SAL *module* clauses.

Z2SAL expresses primed and non-primed variables relationship as model constraints in the guard [21]. However, a value other than an empty value should be assigned to the consequent part since SAL requires all changed primed variables appear in these constraints. For this operational schema, Z2SAL will translate those four lines of predicates as a conjunction of primed and non-primed variables relationship as a guard, as follows:

<pre> TRANSITION [JoinOk : NOT set {PERSON;} ! contains? (members, name?) AND PERSON_counter ! size?(members) < 3 AND members' = set {PERSON;} ! insert(members, name?) AND reply_' = OK → members' IN {x : set {PERSON;} ! Set TRUE}; reply_' IN {x : MESSAGE TRUE} [] </pre>
--

Another operational schema in this specification has a translation below. Two *local* variables are declared. Although their names are the same as the variable names in other schemas, their scopes are in the relevant operational schema, in other words, they are local variables, local to the schema, in which they are declared.

<pre> LeaveClub : set {PERSON;} ! contains? (members, name?) AND </pre>

```

members' = set {PERSON;} ! remove(members, name?) AND
sumC' = PERSON_counter ! size?(members)
->
members' IN {x : set {PERSON;} ! Set | TRUE};
sumC_' IN {x : NAT | TRUE}
[]

```

Z2SAL might include an `invariant_'` in the guard as an invariant will assert a state predicate in the after state of every transition. Together, primed `invariant_` in the guard part and non-primed `invariant_` in the *Initialization* clause perform the truth of state predicates in every part of this system.

Only one transition can be fired on each cycle of a system based on their enabled guards. It is indicated by a "[]" separator which is used between operational schemas, which means an asynchronous operation. Thus, the constraints are only enforced upon one set of input or output variables in each cycle [21].

The last sub-clause of guard command is the `ELSE` and this command is always included in every SAL file generated by Z2SAL. The addition of a default clause ensures that the transition relation is total so model checking is soundness [21]. If `ELSE` is fired then the state of automaton remains unchanged [21].

More on other translations by Z2SAL, especially relating to the Z mathematical tool-kit, see literature such as [21]. Not all aspects of Z specifications were implemented by Z2SAL, as well as not all of those are well supported by the SAL model checker, such as recursive types, processing tuples and their types [21].

There are two aspects of Z which have been not supported by Z2SAL will be discussed in the following sub-sections. It begins with generic constant definition.

The #1 Aspect: Generic Constant Definition

Below are two generic constant definitions which were taken from [57].

$$\begin{array}{|l}
\hline
\hline
[X, Y] \\
\hline
\text{swap2} : X \times Y \rightarrow Y \times X \\
\hline
\forall x : X; y : Y \bullet \text{swap2}(x, y) = (y, x) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\hline
[X] \\
\hline
\text{swap1} : X \times X \rightarrow X \times X \\
\hline
\forall x, y : X \bullet \text{swap1}(x, y) = (y, x) \\
\hline
\end{array}$$

Let usages for these generic definitions are specified in the following schema:

<i>Swap</i>
$a? : NAME; a!, b! : NAME; c? : \mathbb{N}; c! : \mathbb{N}; \exists State$
$(b!, a!) = swap1[NAME, NAME](name, a?)$ $(c!, a!) = swap2(name, c?)$

Z2SAL generated this error: 'Error in the Z input file. This should have been an expression not new identifier'. This error pointed to a line below the first usage on the **Swap** schema.

Thus, it looks as if Z2SAL does not support generic constant definitions. It is since **swap1** has been declared before this usage, but it was declared in a generic constant definition. Thus, it is not a new identifier as Z2SAL claims.

One potential solution to this would be to redefine those generic constant definitions to alternative definitions that are supported by Z2SAL. The alternative definition is an axiomatic definition. It is since both definitions have several similarities. In addition, Z2SAL supports also axiomatic definitions.

Thus, for above example, they could be redefined to equivalent axiomatic definitions as follows:

$swap1 : NAME \times NAME \rightarrow NAME \times NAME$
$\forall x, y : NAME \bullet swap1(x, y) = (y, x)$

$swap2 : NAME \times \mathbb{N} \rightarrow \mathbb{N} \times NAME$
$\forall x : NAME; y : \mathbb{N} \bullet swap2(x, y) = (y, x)$

The associated operational schema could be updated also as follows:

<i>Swap</i>
$a? : NAME; a!, b! : NAME; c? : \mathbb{N}; c! : \mathbb{N}; \exists State$
$(b!, a!) = swap1(name, a?)$ $(c!, a!) = swap2(name, c?)$

Afterwards, Z2SAL could generate a SAL specification from a translation of this Z specification as follows:

<pre> output_fSwap : CONTEXT = BEGIN B_NAT : TYPE = [0..4]; NAT : TYPE = {g : B_NAT g /= 4}; B_NAME : TYPE = {NAME_1, NAME_2, NAME_3, NAME_BB}; NAME : TYPE = {g : B_NAME g /= NAME_BB}; B_NAME_X_B_NAME : TYPE = [B_NAME, B_NAME]; NAME_X_NAME : TYPE = [NAME, NAME]; </pre>

```

NAME_X_NAT : TYPE = [NAME, NAT];

B_NAT_X_B_NAME : TYPE = [B_NAT, B_NAME];

State : MODULE =
BEGIN
  LOCAL swap1 : [NAME_X_NAME -> B_NAME_X_B_NAME]
  LOCAL swap2 : [NAME_X_NAT -> B_NAT_X_B_NAME]
  LOCAL name : NAME
  INPUT a? : NAME
  OUTPUT a_ : NAME
  OUTPUT b_ : NAME
  INPUT c? : NAT
  OUTPUT c_ : NAT
  LOCAL invariant_ : BOOLEAN
  DEFINITION
    invariant_ = (
      function {NAME_X_NAME, B_NAME_X_B_NAME;
        (NAME_BB, NAME_BB)} ! total?(swap1) AND
      function {NAME_X_NAT, B_NAT_X_B_NAME;
        (4, NAME_BB)} ! total?(swap2) AND
      (FORALL (q--1 : NAME, q--2 : NAME) :
        swap1((q--1, q--2)) = (q--2, q--1)) AND
      (FORALL (q--3 : NAME, q--4 : NAT) :
        swap2((q--3, q--4)) = (q--4, q--3)))
  INITIALIZATION [
    a_ = NAME_1 AND
    b_ = NAME_1 AND
    c_ = 3 AND
    invariant_
  ]
  TRANSITION [
    Swap :
      (b_' , a_' ) = swap1((name, a?)) AND
      (c_' , a_' ) = swap2((name, c?)) AND
      name' = name AND
      invariant_'
    ->
      name' IN {x : NAME | TRUE};
      a_' IN {x : NAME | TRUE};
      b_' IN {x : NAME | TRUE};
      c_' IN {x : NAT | TRUE}
    []
    ELSE ->
      name' = name
  ]
END;
END

```

Furthermore, this SAL file could be verified by the SAL model checker. However, the SAL model checker failed to verify it if a LTL theorem was added to this SAL file.

This has happened because the theorem that was added relates to a user-defined function, `swap1`. Thus, it seems that there is a SAL problem on a function translation performed by Z2SAL. Previously, problems with the function application have been experienced in our experiments (see [63]).

```
Error: [Context: output_fSwap, line(22), column(10)]:
Failed to convert function application (array selection).
The function/array does not have a finite domain,
or the argument is not in the domain of the function/array.
```

Above is an error produced by SAL when verifying the below theorem:

```
th1: theorem State |- G(FORALL (i,j: NAME): swap1(i,j) = (j,i));
```

One solution to the above problem would be to modify the SAL file to an alternative function translation that is supported by the SAL tool as follows:

```
output_fSwap_mod : CONTEXT = BEGIN
B_NAT : TYPE = [0..4];
NAT : TYPE = {g : B_NAT | g /= 4};
B_NAME : TYPE = {NAME_1, NAME_2, NAME_3, NAME_BB};
NAME : TYPE = {g : B_NAME | g /= NAME_BB};
B_NAME_X_B_NAME : TYPE = [B_NAME, B_NAME];
NAME_X_NAME : TYPE = [NAME, NAME];
NAME_X_NAT : TYPE = [NAME, NAT];
B_NAT_X_B_NAME : TYPE = [B_NAT, B_NAME];
swap1(q--1 : NAME, q--2 : NAME): B_NAME_X_B_NAME = (q--2, q--1);
swap2(q--3 : NAME, q--4 : NAT): B_NAT_X_B_NAME = (q--4, q--3);
State : MODULE =
  BEGIN
    LOCAL name : NAME
    INPUT a? : NAME
    OUTPUT a_ : NAME
    OUTPUT b_ : NAME
    INPUT c? : NAT
    OUTPUT c_ : NAT
    LOCAL invariant__ : BOOLEAN
    INITIALIZATION [
      a_ = NAME_1 AND
      b_ = NAME_1 AND
      c_ = 3 AND
      invariant__
    ]
    -->
  ]
END;
th1: theorem State |- G(FORALL (i,j: NAME): swap1(i,j) = (j,i));
END
```

The transition clause was not shown as there was no any difference between

the first SAL file and the modified one. This modified SAL file could be verified and simulated by the SAL tools.

Thus, based on the above discussion, it is one solution to have a system that can redefine a generic constant definition to an equivalent axiomatic definition. It is also necessary to have a SAL file that can be executed by the SAL tool.

Let us move to the second aspect of Z which are not supported by Z2SAL. It is schema calculus.

The #2 Aspect: Schema Calculus Definition

Z2SAL is able to translate several schema operators, such as " Δ ", " Ξ ", schema decorations, specifically " $'$ " and schema inclusions. However, they must be specified either vertically or horizontally in a schema.

Therefore, if a new schema is constructed from earlier schemas, Z2SAL does not support the schema construction. Thus, this kind of schema calculus is not supported by Z2SAL.

A constructed schema is specified by using " $\hat{=}$ " which is the same as the supported schema calculus, but the constructed one does not use " $[$ " and " $]$ " to surround its declaration of variables and predicates. The constructed schema is used commonly to define a more complex, modular and a much larger specification of a system.

Consider a Z specification as given below:

$[Seat, Customer]$	$Response ::= okay \mid sorry$										
$\mid initial_allocation : \mathbb{P} Seat$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 5px;"><i>BoxOffice</i></td></tr> <tr><td style="padding: 5px;"><i>seating</i> : $\mathbb{P} Seat$</td></tr> <tr><td style="padding: 5px;"><i>sold</i> : $Seat \rightarrow Customer$</td></tr> <tr><td style="padding: 5px;">$dom\ sold \subseteq seating$</td></tr> </table>	<i>BoxOffice</i>	<i>seating</i> : $\mathbb{P} Seat$	<i>sold</i> : $Seat \rightarrow Customer$	$dom\ sold \subseteq seating$						
<i>BoxOffice</i>											
<i>seating</i> : $\mathbb{P} Seat$											
<i>sold</i> : $Seat \rightarrow Customer$											
$dom\ sold \subseteq seating$											
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 5px;"><i>InitBoxOffice</i></td></tr> <tr><td style="padding: 5px;"><i>BoxOffice'</i></td></tr> <tr><td style="padding: 5px;">$sold' = \emptyset$</td></tr> <tr><td style="padding: 5px;">$seating' = initial_allocation$</td></tr> </table>	<i>InitBoxOffice</i>	<i>BoxOffice'</i>	$sold' = \emptyset$	$seating' = initial_allocation$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 5px;"><i>Purchase0</i></td></tr> <tr><td style="padding: 5px;">$\Delta BoxOffice$</td></tr> <tr><td style="padding: 5px;">$s? : Seat; c? : Customer$</td></tr> <tr><td style="padding: 5px;">$s? \in seating \setminus dom\ sold$</td></tr> <tr><td style="padding: 5px;">$sold' = sold \cup \{s? \mapsto c?\}$</td></tr> <tr><td style="padding: 5px;">$seating' = seating$</td></tr> </table>	<i>Purchase0</i>	$\Delta BoxOffice$	$s? : Seat; c? : Customer$	$s? \in seating \setminus dom\ sold$	$sold' = sold \cup \{s? \mapsto c?\}$	$seating' = seating$
<i>InitBoxOffice</i>											
<i>BoxOffice'</i>											
$sold' = \emptyset$											
$seating' = initial_allocation$											
<i>Purchase0</i>											
$\Delta BoxOffice$											
$s? : Seat; c? : Customer$											
$s? \in seating \setminus dom\ sold$											
$sold' = sold \cup \{s? \mapsto c?\}$											
$seating' = seating$											
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 5px;"><i>Success</i></td></tr> <tr><td style="padding: 5px;">$r! : Response$</td></tr> <tr><td style="padding: 5px;">$r! = okay$</td></tr> </table>	<i>Success</i>	$r! : Response$	$r! = okay$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 5px;"><i>Failure</i></td></tr> <tr><td style="padding: 5px;">$r! : Response$</td></tr> <tr><td style="padding: 5px;">$r! = sorry$</td></tr> </table>	<i>Failure</i>	$r! : Response$	$r! = sorry$				
<i>Success</i>											
$r! : Response$											
$r! = okay$											
<i>Failure</i>											
$r! : Response$											
$r! = sorry$											

$$Purchase \hat{=} (Purchase0 \wedge Success) \vee (Purchase0 \wedge Failure)$$

This specification contains a schema calculus definition at the bottom of this specification.

Z2SAL generated an error as shown by Fig. 2.8 on page 72. A variable with a certain type is required instead of a schema name.

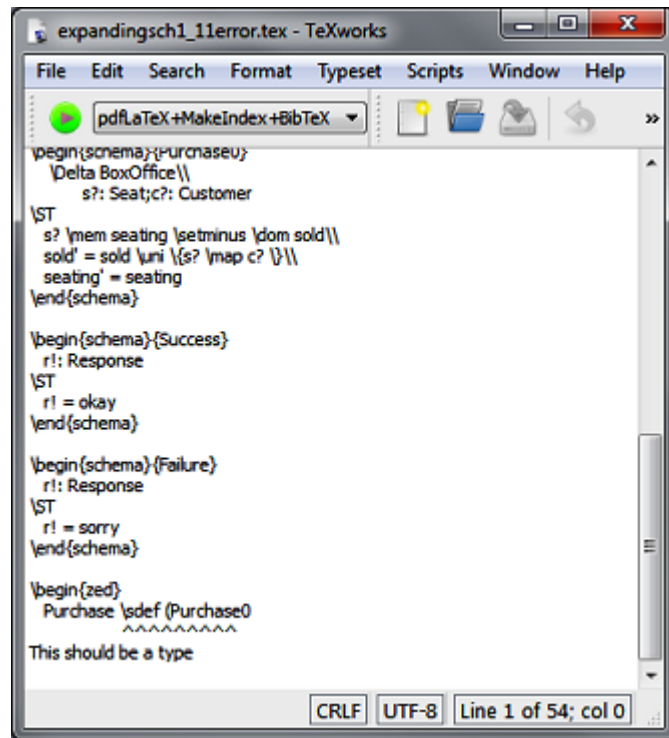


Figure 2.8: Error message from Z2SAL

One possible solution to this problem is to rewrite a schema calculus definition to a definition that is supported by Z2SAL. Such a definition is a schema definition. Thus, the schema calculus will be expanded that it will form a schema definition to create a new schema.

Let us have the below schema which was created from the above schema calculus definition:

$$\begin{array}{l}
 \text{Purchase} \\
 \Delta \text{BoxOffice}; r! : \text{Response}; s? : \text{Seat}; c? : \text{Customer} \\
 \hline
 (s? \in \text{seating} \setminus \text{dom sold} \wedge \\
 \text{sold}' = \text{sold} \cup \{s? \mapsto c?\} \wedge \\
 \text{seating}' = \text{seating}) \\
 \wedge \\
 ((r! = \text{okay}) \\
 \vee \\
 (r! = \text{sorry}))
 \end{array}$$

The new `Purchase` schema was created by expanding three schemas with specified operators as given in the associated schema calculus definition.

This expanded specification could be translated by Z2SAL. As well as the generated SAL could be verified and simulated by the SAL tools. Thus, this expansion can be a potential solution to a schema calculus problem.

The next section is a conclusion.

2.5.3 Conclusion

As a translator from a Z language specification to a SAL language one, Z2SAL can be grouped as one of the Z tools, specifically in supporting model checking Z specifications. The existence of this translator should be supported in order to solve the lack of Z tools particularly in verification of this language.

In order to support model checking for Z specifications, some experiments have been performed on this translator. Based on this, what support that could offer by this research is described further and implemented in the later chapters.

The next chapter is a discussion about our approach to the method to a translation of embedded theorems on Z specifications. It is because the current practices in using Z2SAL let users to add theorems in the generated SAL.

Chapter 3

Translation of Embedded Theorems in Z Specifications: A Proposed Method

In this chapter, how properties of a system are written will be discussed. Previously, a user added several theorems to the generated SAL file in order to let the system, which is modelled by the Z specification, verifies these theorems. By doing so, this user should know how the SAL language presents this theorem, which might be a problem to learn other language, the SAL language.

Then, an idea, how the user specifies the theorem inside the Z specification, was proposed. As a result the current Z2SAL can translate either a Z specification or a Z specification added with theorems.

The discussion with the old practice in verifying the theorems is given first. Later is the discussion of our idea.

3.1 Adding Theorems in the Generated SAL

At the end of our SAL file of `club.tex`, which is given on Section 2.5.2 on page 63, several LTL theorems and CTL theorems were added as presented on Section 2.3.1 on page 39. These theorems were written by using the SAL language shown as follows:

- `th1 : THEOREM State |- G (NOT (members = set {PERSON;} ! full));`
It is not the case such that a club ever gets full.
- `th2 : THEOREM State |- G (NOT(set {PERSON;}! empty?(members)));`
It is not the case such that the club ever be empty.

- **th3a** : THEOREM State $\vdash G (\text{EXISTS}(m, n: \text{PERSON}): m \neq n)$;
There exists at least one instance of `members`, who is different from other `members`.
- **th3b** : THEOREM State $\vdash G (\text{NOT}(\text{EXISTS}(m, n: \text{PERSON}): m \neq n))$;
It is not the case such that there is a member of `members`, who is different with others.
- **th4a** : THEOREM State $\vdash G (\text{FORALL}(m, n: \text{PERSON}): m \neq n)$;
All members are different.
- **th4b** : THEOREM State $\vdash G (\text{NOT}(\text{FORALL}(m, n: \text{PERSON}): m \neq n))$;
It is not the case such that all members are different.

Before these theorems were verified by using the SAL model checker, they were investigated manually. For the first theorem **th1**, it should be invalid since there is an operation `JoinOk` that can add a member to this club. Furthermore, this operation only stops if the maximum number of members is reached.

For **th2**, it is also invalid since in the initialization of this system, this club has no member, in other words this system has ever been empty. For **th3a**, it will be proved as the operation performed by the `JoinOk` schema will only add a new member who has not been available in this club. For **th3b**, it is a counter-example of the **th3a** theorem, thus it is invalid. Theorem **th4a** is also invalid due to the assignment of no members for this club in the initial state. Thus, in the initialization state, all members are the same, which are empty. The last theorem is the counter-example of theorem **th4a**, thus it will be proved.

Based on this prior knowledge, the SAL model checker was run on this generated SAL file to verify those theorems then. The summary of that verification is given as follows and they are the same as our expected results:

```
The assertion 'th1' located at [Context: club, line(55), column(0)] is invalid.
The assertion 'th2' located at [Context: club, line(58), column(0)] is invalid.
The assertion 'th3a' located at [Context: club, line(61), column(0)] is valid.
The assertion 'th3b' located at [Context: club, line(64), column(0)] is invalid.
The assertion 'th4a' located at [Context: club, line(67), column(0)] is invalid.
The assertion 'th4b' located at [Context: club, line(70), column(0)] is valid.
```

Many other examples given on other sections in this thesis show this practice. Let us now move to the next discussion on embedded theorems.

3.2 Embedded Theorems on Z Specifications

Duke and Smith say that by presenting a specification of a system using the Z notation, properties of this system such as liveness can be evaluated [25]. There are two alternatives to formulate these properties. Firstly, it is to use the Z notation itself. Another one is to use the temporal logic notation. Duke and Smith shown that by adapting the latter, predicates are more readable and shorter than the former.

King also added tags for presenting several temporal logics to his `oz` package [42]. Now, there are only three tags available, as follows:

- \square : for representing **always**
- \bigcirc : for representing **next**
- \diamond : for representing **eventually**

Despite the work of Duke and Smith [25], there are fewer tools that implements embedded temporal theorems on a Z specification. Therefore, this research proposed an extension to the Z standard notation adapted by Z2SAL to include also the temporal logic provided by King as can be seen in Fig. 3.1 on page 77. Let us discuss briefly this picture first.

Processes inside dotted-line boxes represent our works. From Fig. 3.1, our works can be divided into two categories. The first category relates to Z specification input files. The input file will be pre-processed by our tool in such a way to let it be translated successfully by Z2SAL. Thus, the output of this category is a tool to pre-process Z specifications.

Our pre-processed tool begins with scanning and parsing the input file. The further process can be either redefining generic constant definitions or expanding schema calculus definitions. It is based on a choice defined by a user. Further discussion in this work can be read in Sections 4, 5, 6, 7, 8 and 9.

The second category relates to SAL specification files. There are three works in this category. Proposing an algorithm for translation of embedded theorems in Z specifications is the first work. This proposal will be discussed in this section. The second work is proposing an algorithm for user-defined functions. This work will be discussed further in Section 5.5. This proposal has not been implemented in any tool. The last work in this category relates to modifying types to solve type incompatibility. The same as the second work, this is also manual works. These manual works can be seen in experiments with the generic constant redefinition, specifically in Sections 9.2.1 ,9.2.3 ,9.2.4 and 9.2.5.

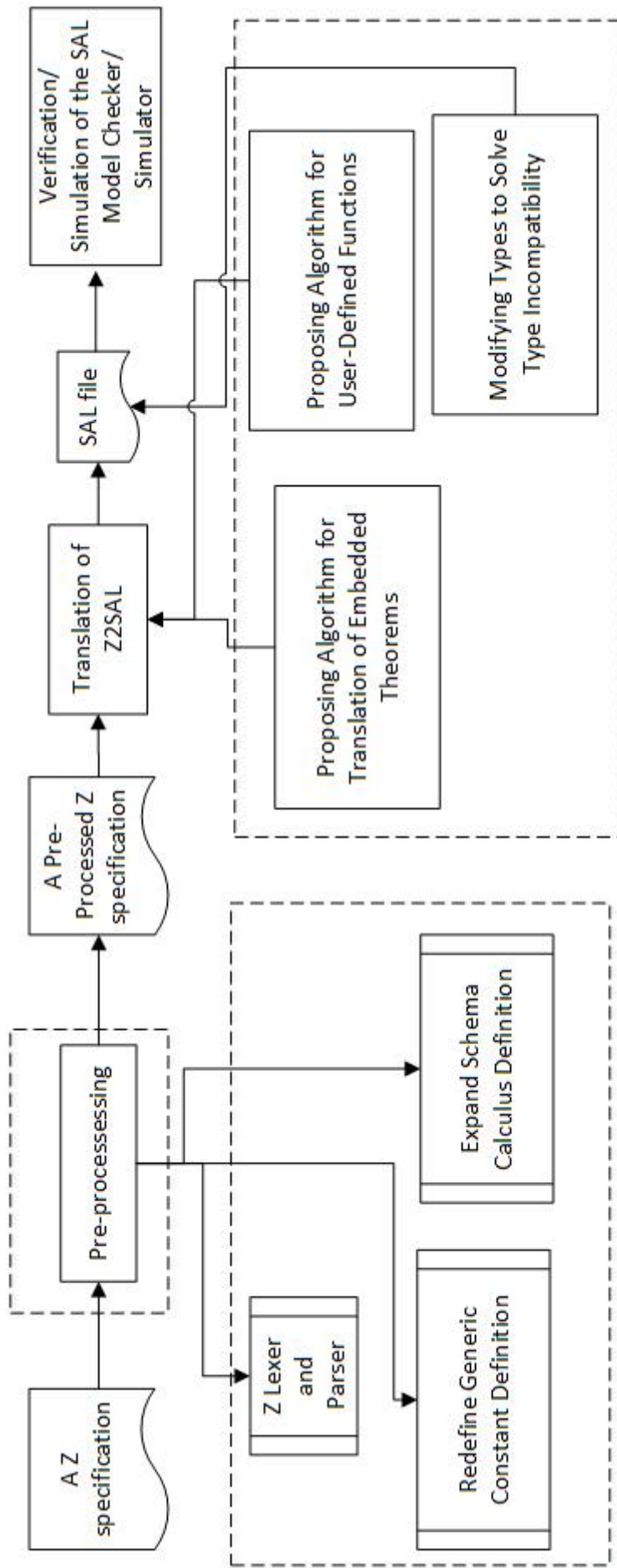


Figure 3.1: The Architecture

Having the above discussion, let us back again to the embedded theorem. The research proposed a method which is adapted from *Object-Z Concrete Syntax* [24] in specifying embedded temporal logics on a Z specification. This concrete syntax is an extension of *Spivey's Z Concrete Syntax* [69].

Following is the relevant definition to embedded theorems:

$$\begin{aligned} \textit{Specification} & ::= \textit{Paragraph NL} \dots \textit{NL Paragraph} \\ \textit{Paragraph} & ::= \dots \\ & \quad \dots \\ & \quad \textit{ClassBox} \\ & \quad \dots \end{aligned}$$

ClassBox

$\begin{aligned} & \textit{ClassName} \\ & \textit{LocalDefs} \\ & \textit{State} \\ & \textit{Init} \\ & \textit{Op} \\ & \dots \\ & \dots \\ & \textit{Op} \\ & [\\ & \textit{HistPred} \end{aligned}$

$$\begin{aligned} \textit{Predicate} & ::= \dots \\ & \quad \textit{Let} \end{aligned}$$

$$\textit{Let} ::= \textit{let Defn SEP} \dots \textit{SEP Defn} \bullet \textit{Predicate}$$

$$\textit{Defn} ::= \textit{DefLhs} == \textit{Expression}$$

$$\begin{aligned} \textit{HistPred} & ::= \forall \textit{Declaration} [\textit{HistPred}] \bullet \textit{HistPred} \\ & \quad \exists \textit{Declaration} [\textit{HistPred}] \bullet \textit{HistPred} \\ & \quad \exists_1 \textit{Declaration} [\textit{HistPred}] \bullet \textit{HistPred} \\ & \quad \textit{HistPred1} \end{aligned}$$

$\textit{HistPred1} ::= \textit{op} = \textit{Expression}$	– op is an identifier
$\textit{op} \in \textit{Expression}$	– which is used to access
$\textit{Predicate}$	– the last operation that
$\square \textit{HistPred1}$	– has occurred
$\diamond \textit{HistPred1}$	
$\bigcirc \textit{HistPred1}$	
$\neg \textit{HistPred1}$	
$\textit{HistPred1} \wedge \textit{HistPred1}$	
$\textit{HistPred1} \vee \textit{HistPred1}$	
$\textit{HistPred1} \Rightarrow \textit{HistPred1}$	
$\textit{HistPred1} \Leftrightarrow \textit{HistPred1}$	

Based on these syntax rules, temporal logics are formulated under the definition of `HistPred`. Several syntax rules are not given here, but they can be found on those references.

Proposed procedure of a translation of embedded properties on a `Z` specification will be given in the following sub-section.

3.2.1 Proposed Method in a Translation of Embedded Theorems on `Z` Specifications

This section gives a method on how to translate LTL theorems, which are embedded on a `Z` specification, into the suitable theorems on the generated SAL file. Our aim is to benefit `Z2SAL` so it will be able to translate directly embedded theorems on schemas on a `Z` specification to appropriate theorems on the SAL specification.

The translation on these embedded theorems will use the format of theorems specified previously by `Z2SAL`, which suits the SAL model checker's format [17], as follows:

<pre>th i: THEOREM name_of_module - temporal_logics;</pre>

- `th` is an identifier, so it can be changed to other name. Each theorem will be given a name `th` followed by `i`.
- `i` is a non-zero natural number starting from 1, which plays as an index. This `i` number will be defined for each line of predicates containing temporal logics with the first occurrence gets 1 and it will be incremented by 1 for each successor of such a line. It is also part of the user identifier, thus it can be changed to other identifier.
- `THEOREM` is a SAL keyword, but it is not case-sensitive.
- `name_of_module` is taken from the name of the SAL's module and is case-sensitive.
- Based on [17], the translation of King's \LaTeX style temporal logics [42] into equivalent temporal logics of the SAL model checker are as shown in Table 3.2.1 on page 80.

Other temporal logic such as $\mathbf{U}(p, q)$ or $\mathbf{p U q}$, which is not available in [42], they can be derived from the three temporal logics above by considering equivalences in temporal logics as follows:

$$U(p, q) \equiv W(p, q) \wedge Fq \tag{3.1}$$

Table 3.1: Equivalence of temporal logic notations

<i>Style of King</i>	<i>Style of SAL</i>
□	G
○	X
◇	F

$$W(p, q) \equiv R(q, p \vee q) \quad (3.2)$$

$$R(p, q) \equiv \neg (U(\neg p, \neg q)) \quad (3.3)$$

Thus, these equivalences can be defined as:

$$\begin{aligned}
 U(p, q) &\equiv W(p, q) \wedge Fq \\
 &\equiv R(q, p \vee q) \wedge Fq \quad \text{applying eq. 2} \\
 &\equiv \neg (U(\neg q, \neg (q \vee p))) \wedge Fq \quad \text{applying eq. 3} \\
 &\equiv \neg (U(\neg q, \neg q \wedge \neg p)) \wedge Fq \quad \text{applying De Morgan's law} \\
 &\equiv \neg (\neg q) \wedge Fq \quad \text{interpreting the sub - formula} \\
 &\equiv q \wedge Fq
 \end{aligned}$$

The SAL model checker does not support other temporal logics. Furthermore, in a case, there are new identifiers in the schema consisting of temporal logics notation; Z2SAL can choose one value for each of these identifiers, which suits their types.

The next section gives several experiments on this proposed translation method.

3.2.2 Experiment 1: Unique Allocator Specification

An example taken from Smith and Winter [68] is used in this experiment. This specification is given in Appendix C.6 on page 279.

As an experiment, the same property as given in [68] was used as follows:

$$G(n! \neq v \vee X(G(n! \neq v)))$$

, in which v has a type of " \mathbb{N}_1 ".

If an LTL theorem symbol is used, the theorem can be represented as:

$$\square(n! \neq v \vee \bigcirc(\square(n! \neq v)))$$

A new schema, which represents this theorem, was added to the specification. There are two approaches to specify that theorem: firstly, it uses the Z notation as follows:

<i>uniqueSend</i>
$\exists \text{Allocator}; n! : \mathbb{N}$
$\forall i : \mathbb{N}_1 \bullet n! \neq i \vee \forall j : \mathbb{N}_1 \bullet (j = i + 1 \wedge n! \neq j)$

Secondly, it uses the temporal logic notation, as follows:

<i>uniqueSend</i>
$\exists \text{Allocator}; n! : \mathbb{N}$
$\forall v : \mathbb{N}_1 \bullet \square(n! \neq v \vee \bigcirc(\square(n! \neq v)))$

The above schema can also be specified as follows:

<i>uniqueSend</i>
$\exists \text{Allocator}; n! : \mathbb{N}$
$\square(n! \neq 2 \vee \bigcirc(\square(n! \neq 2)))$

, in which an explicit value for v is chosen and it replaces the v .

Since writing LTL theorems by using the temporal logic notation is more concise and easier to read, this notation will be used in our experiments.

Z2SAL generated theorems for the last two schemas as follows:

th1: theorem State $\vdash G((n_ /= 2) \text{ OR } X(G(n_ /= 2)))$;
th2: theorem State $\vdash \text{FORALL}(q_2 : \text{NZNAT}) : G(n_ /= q_2 \text{ OR } X(G(n_ /= v)))$;

The `th1` theorem was generated from the last schema of the above specification, whereas the `th2` theorem was generated from the schema just before the last schema.

The generated SAL file should be modified so the SAL tools, as discussed on Chapter 2 can verify it. Finally, it could be verified and simulated by the SAL tools. Those theorems are valid.

3.2.3 Experiment 2: Counter 4 Modulo Specification

Another example is a 4 modulo operation. The full specification is given in Appendix C.5 on page 279.

One property that is required to be proven is as follows:

$G((\text{count!} = 1) \Rightarrow X(\text{count!} = 2))$;

Using tags of King, the theorem is as follows:

$$\square((\text{count!} = 1) \Rightarrow \bigcirc(\text{count!} = 2));$$

The theorem was presented in the below schema:

$\frac{\text{count1Next2}}{\exists \text{CounterMod4}; \text{count!} : \mathbb{N}}$
$\square(\text{count!} = 1 \Rightarrow \bigcirc(\text{count!} = 2))$

Another property is given as follows to show that `count!` is never greater than the maximum, which is defined to be 3:

$\frac{\text{countMax}}{\exists \text{CounterMod4}; \text{count!} : \mathbb{N}}$
$\square(\neg(\text{count!} > 3))$

Z2SAL translated the above schemas into these theorems:

<pre>th1: THEOREM State - G((count_ = 1) => X(count_ = 2));</pre>
<pre>th2: THEOREM State - G(NOT(count_ > 3))</pre>

The generated SAL file was modified before verification took place. For example: assigning 0 to `count_` and deleting the `ELSE` clause. The first modification is necessary so as the second theorem can be satisfied by the system as expected; whereas the second one is to avoid weird results generated by the SAL model checker. Both theorems are valid. This SAL file was successfully verified and simulated by the SAL tools.

3.2.4 Experiment 3: Cars Park Specification

This specification is taken from [49]. The whole specification can be seen in Appendix C.7 on page 280.

Assume, the properties to be proven were specified in this schema:

$\frac{\text{spaceChecking}}{\exists \text{CarsPark}; \text{space!} : \mathbb{N}}$
$\square(\diamond(\neg(\text{space!} > \text{maximum})))$
$\square(\diamond(\text{space!} \leq \text{maximum}))$
$\square(\text{space!} \neq 3 \vee \bigcirc(\text{space!} \neq 3))$

The first and the second theorems show that `space!`, which represents the remaining space of a park area, is never greater than maximum, which represents the maximum capacity of the park area some time in future. The last theorem shows that the number of space is never the same at any time.

Z2SAL produced these theorems:

<pre>th1 : theorem State - G (F(NOT (space_ > maximum)));</pre>
<pre>th2 : theorem State - G (F(space_ <= maximum));</pre>
<pre>th3 : theorem State - G (space_ /= 3 OR X(space_ /= 3));</pre>

All these theorems were proved by the SAL model checker after `space_` is assigned 0 for its initial value. The SAL simulator could also simulate this SAL file.

3.2.5 Experiment 4: Birthday Book Specification

This specification is obtained from [69] and can be seen in Appendix C.8 on page 280.

A property to be proven is *If it is known the birthday of a person then the person should be recognized*, and it was specified in the schema as follows:

<i>WhichDate</i>
$\exists \text{BirthdayBook}$
$\forall n : \text{NAME} \bullet \exists d : \text{DATE} \bullet$ $\square (d = \text{birthday}(n) \Rightarrow n \in \text{known})$

Z2SAL translated this schema into following theorems:

<pre>th1 : theorem State - (FORALL (q--2 : NAME) : (EXISTS (q--3 :DATE) : G (q--3 = birthday(q--2) => set {NAME;} !contains?(known, q--2))));</pre>

The theorem above was satisfied by the system; it is valid. This SAL file could be verified and simulated by the SAL tools.

3.2.6 Experiment 5: Paper Example Specification

This specification is taken from [21]. Full specification can be seen in the reference or Appendix C.9 on page 281. By using the same LTL theorems as given on this paper, new schemas were added to the specification.

<pre>th1: theorem State - G (rented = set {PERSON_X..TITLE;}!empty);</pre>

This theorem was a translation from following schema:

<i>RentedTheorem</i>
$\exists \text{State}$
$\square (\text{rented} = \emptyset)$

`rented` is a relation between `PERSON` and `TITLE`.

The definition of the mathematical tool kit set, such as the relation context can be seen in the separate SAL context produced by Z2SAL after translating this Z specification.

Another schema consisting of theorems were added given as follows:

<i>MembershipTheorem</i>	
$\exists State$	
\square	$(\neg (\#members = \#PERSON))$
\square	$(\#members = \#PERSON)$
\diamond	$(\neg (\#members = \#PERSON))$
\diamond	$(\#members = \#PERSON)$

These predicates were translated into four theorems as follows:

```
th2:theorem State |- G (NOT (Counter_PERSON ! size?(members)=3));
th3:theorem State |- G (Counter_PERSON ! size?(members)=3);
th4:theorem State |- F (NOT (Counter_PERSON ! size?(members)=3));
th5:theorem State |- F (Counter_PERSON ! size?(members)=3);
```

These predicates were put on the same schema since they have a similarity which is to verify **members**. SAL does not have a keyword to present a size of a set, so Z2SAL formulate this size directly by using the special function, **size?**. This operator can be seen in the counting function context.

```
th6 : theorem State |- G (copies_ /= 3);
th7 : theorem State |- G ((FORALL (q_1 : TITLE) : stockLevel(q_1) /= 3));
```

Both these theorems were presented in this schema:

<i>CopiesTheorems</i>	
$\exists State; copies! : \mathbb{N}$	
\square	$(copies! \neq 3)$
\square	$(\forall t : TITLE \bullet stockLevel(t) \neq 3)$

SAL has several keywords to represent the relevant operators of the Z notation, but there are slightly differences. For example, SAL uses a colon, ":", to represent a "•".

```
th8 : theorem State |- G ((FORALL (q_2 : TITLE) : stockLevel(q_2) >=
Counter_PERSON_X_TITLE ! size?(relation {PERSON, TITLE;} !
rangeRestrict(rented, set {TITLE;} ! singleton(q_2)))));

th9 : theorem State |- G (NOT (FORALL (q_3 : PERSON) :
(FORALL (q_4 : TITLE) : set {PERSON_X_TITLE;} !
contains?(rented, (q_3, q_4)) AND stockLevel(q_4) >= 3)));

th10 : theorem State |- G (NOT (FORALL (q_5 : PERSON) :
(FORALL (q_6 : TITLE) : set {PERSON_X_TITLE;} !
contains?(rented, (q_5, q_6)) AND
stockLevel(q_6) >= Counter_PERSON_X_TITLE !
size?(relation {PERSON, TITLE;} !
rangeRestrict(rented, set {TITLE;} ! singleton(q_6))))));
```

<i>RentedVideosTheorems</i>	
$\exists State$	
$\square (\forall t : TITLE \bullet stockLevel(t) \geq \#(rented \triangleright \{t\}))$	
$\square (\neg (\forall p : PERSON \bullet \forall t : TITLE \bullet (p, t) \in rented \wedge stockLevel(t) \geq 3))$	
$\square (\neg (\forall p : PERSON \bullet \forall t : TITLE \bullet (p, t) \in rented \wedge stockLevel(t) \geq \#(rented \triangleright \{t\})))$	

From those ten theorems, only the fourth theorem is valid. The SAL simulator could also simulate this SAL file.

3.2.7 Experiment 6: Shop Specification

Please refer to Appendix C.1 on page 274 for the whole specification. Assume there is a property that says that for every item in the stock there will be a price specified for it. However, the property is not true since sometimes there is an item which does not have any price.

<i>hasPrice</i>	
$\exists Shop$	
$\forall i : ITEM \bullet \exists p : \mathbb{N} \bullet \square (p = cost(i))$	

Z2SAL translated the schema into the theorem as follows:

$th1 : theorem State \mid - (FORALL (q_{-3} : ITEM) : (EXISTS (q_{-4} : NAT) : G (q_{-4} = cost(q_{-3}))));$
--

This SAL file could be verified and simulated by the SAL tools. The theorem, which has mentioned before, is invalid.

The next section summarises results and discusses them.

3.2.8 Result and Discussion

Results from experiments on embedded theorems are depicted in Table 3.2 on page 85.

Table 3.2: Execution Time of Embedded Theorem Experiments

Experiment	Number of Theorems	Execution Time (secs)
# 1	2	0.359
# 2	2	0.187
# 3	3	0.515
# 4	1	0.514
# 5	10	10.561
# 6	1	126.658

The time can be different though from the same specification if it is run several times. As can be seen in Table 3.2 on page 85, the last experiment

Table 3.3: Details of Execution Time of Experiment #6

Process	Time (secs)
ast generation for context function	0.0
type-checker for context function	0.016
ast generation for context set	0.0
type-checker for context set	0.0
ast generation for context shop_templog	0.0
type-checker for context shop_templog	0.015
module flattening	0.0
ast simplification	0.032
function application expansion	0.046
unfolding quantifiers	0.032
flat module – common subexpression elimination	0.312
flat module \rightarrow boolean flat module conversion	0.483
monitor generation	27.191
static order	1.248
cluster compression	0.078
flat module \rightarrow BDD conversion	96.955
verification	1.544

consumed the most of execution time just to verify one theorem. The large amount of time taken from Table 3.2 on page 85 was composed by several chunks of individual time as shown in Table 3.3 on page 86.

The most usage of time was on BDD conversion execution, it is nearly 80% of total used time. The second large amount of time taken was used to create monitor or buchi-automata for LTL property used on this theorem of this specification. On the other hand, the time for verification is less than 2 seconds. The execution of this specification had 44593 `total bdd node count` and visited 143699850.0 states.

Data in Table 3.2 on page 85 also shows us that a large number of theorems do not in line with total of execution time. It might be the specification consisting of the lower number of theorem can take much more time of execution. Thus, in addition to the number of theorems, complexity of a specification, complexity of a theorem, and size of used sets influence total execution time.

3.3 Conclusion

Based on these experiments, Z2SAL supports the translation of theorems, which are embedded on a Z specification. Furthermore, the SAL model checker could verify the theorems whether they were specified in the Z specification or in the generated SAL file. Therefore, the SAL model checker did not distinguish these theorems though they can be formulated using one of those methods. These experiments show also that our proposal representing algorithm to do the translation of embedded theorems works. This success is expected as our contribution to related research in this field. The proposal is part of the architecture of our research as shown by Fig. 3.1 on page 77.

Being able to translate LTL theorems, which are embedded on a Z specification, makes verification such a system faster and easier than previously. It is because a user of Z does not need to formulate their theorems at the end of the generated SAL file; they can formulate the theorems inside their Z specification instead. Having knowledge of writing Z specifications, such a user will write quickly theorems using the Z notation.

Following section discusses an implementation of our Z scanner and parser.

Chapter 4

Implementing A Z Scanner and Parser

As described on earlier chapter, our support is on one of generic constructs, which is a generic constant. Other type of support is the Z schema calculus. In addition to both types of support, a method in a translation of functions was also proposed. Furthermore, the first two types of support require other preliminary types of support: a scanner that scans Z input specifications into tokens, and a parser that parses these tokens conforming to the Z grammar. All these types of support are considered as our support for model checking Z specifications.

This chapter will discuss how our support for Z scanner and parser is implemented. Before this discussion begins, let us consider benefits that can be offered by these systems.

Z2SAL has been known also as a scanner and parser for Z specifications, specifically a hand-written scanner and a hard-coded parser. It is since Z2SAL researchers wrote their scanner and parser by using Java language; it is a language that is not specialized for writing scanners and parsers.

Thus, it is one reason for us not to reuse the Z2SAL scanner and parser to implement our Z scanner and parser. Other reason is that it will take time and be an effort to hand-write such a scanner and hard-code such a parser, such as to define regular expressions, and Z operators' precedencies and associativity, to match a sequence of tokens to one of the Z rules, and others. Another reason is that a JFlex lexer has been known to be faster than a hand-written scanner or lexer. Although a BYACC/J parser is not as fast as a hard-coded parser, the BYACC/J parser is easy to write and modify. Moreover, to learn code of somebody else is more difficult rather than to write code from scratch. More importantly, Z2SAL scanner and parser were integrated into the design of other parts of Z2SAL.

Based on these reasons, let us move to the first discussion. It is a design and implementation of our Z scanner.

4.1 A Z Scanner

This section is composed of several sub-sections. It begins with an introduction to a Java scanner generator, and a lexical specification. Based on both overviews, a Z scanner was designed and implemented as a part of our support for model checking Z specifications. Thus, both sub-sections will be followed by an implementation of our Z scanner. In that sub-section, a brief description of our Z scanner will be discussed. This Z scanner will scan Z tags in our Z specifications. A successful scanning will pass tokens, which are obtained from accepted Z tags, to a parser for further process. Otherwise, a lexical error on an involved line will be reported. This section ends up with a conclusion sub-section.

4.1.1 Introduction

JFlex is a Java lexical analysis generator (scanner generator) [43]. JFlex 1.6.1 is the current version which was released on 16 March 2015. It is free software which is published under a BSD-style license.

JFlex will generate a `.java` file from a JFlex specification `.flex`. Thus, this generator has an input which is the JFlex specification. The Java file has one class that consists of code for the scanner. This code is a lexer that reads input, matches the input against the regular expression, and runs an associated action.

A lexer is a part of a compiler, specifically the first front-end of it. The lexer will match keywords, comments, operators, etc. Then it will generate a stream of input tokens for a parser. However, it can also be used for other intentions. Built on a deterministic finite automaton (DFA), a JFlex lexer is fast since backtracking is not performed.

Several parser generators can be integrated with this lexer. For example: the LALR parser generator Construction of Useful Parsers (CUP) by Scott Hudson, the Java modification of Berkeley Yet Another Compiler Compiler (YACC), BYACC/J, by Bob Jamison [33]. To interface a generated scanner with BYACC/J, the command `%byacc` is used.

4.1.2 A Lexical Specification

A JFlex specification, as JLex, is composed of three parts that are separated by "%%". These parts are as follows: user code, options and declarations, and lexical rules.

User Code

This is the first part; begins from the first line until the first "%%". JFlex will copy the contents of this part to the generated Java file. Usually `package` and `import` statements are put here. JFlex will also generate automatically a Javadoc comment with or without being defined explicitly by a user.

Options and Declarations

This part contains a set of options, code for the generated scanner class, lexical states, and macro declarations. The option on JFlex must begin a line of the specification and must start it with a "%". The code that is written inside "%{" and "%}" is copied verbatim into the generated lexer class.

This code can contain member variables and functions that will be used inside scanner actions. There are also macros which are abbreviations for regular expressions. Macros make lexical specifications easy to read and understand. Its form is as follows:

macroidentifier = regular expression

Lexical Rules

This part consists of regular expressions. It also can contain actions in Java code, which will be executed when the associated regular expression is matched. The scanner will match the longest regular expression. In a case there are more than one regular expressions match with the input, the scanner will choose the first matched expression.

A lexical specification on JFlex is based on lexical rules which follow the Extended Backus-Naur Form (EBNF) grammar. This grammar can be read further on [43].

JFlex adapts standard operator precedencies in a regular expression as follows:

1. unary postfix operators (*, '+', '?', {n}, {n,m})

2. unary prefix operators ('!', '~')
3. concatenation (RegExp ::= RegExp RegExp)
4. union (RegExp ::= RegExp '|' RegExp)

In the above list, terminal symbols are enclosed by 'quotes' to differ from non-terminal symbols. As the number gets smaller, the precedence gets higher.

A regular expression is a 'mold' which is based on meta language, to specify particular patterns of interest [45]. If **a** and **b** are regular expressions, then:

- **a | b** (union), is the regular expression that matches all inputs matched by **a** or by **b**.
- **a b** (concatenation), is the regular expression that matches the input matched by **a** followed by the input matched by **b**.
- **a*** (Kleene closure) matches zero or more repetitions of the input matched by **a**.
- **a+** (iteration), is equivalent to **aa***.
- **a?** (option) matches the empty input or the input matched by **a**.
- **!a** (negation) matches everything but the strings matched by **a**.
- **~a** (upto), matches everything up to (and including).
- **a{n}** (repeat), is equivalent to **n** times the concatenation of **a**.
- **a{n,m}** is equivalent to at least **n** times and at most **m** times the concatenation of **a**.
- **(a)** matches the same input as **a**.
- **.** matches any single character except the newline character ("**\n**").
- **[]** (character class), matches any character within the brackets. If the first character is a circumflex ("**^**"), it matches any character except the ones within brackets. If there is a dash ("**-**") inside brackets, it means a character range.
- **^** matches the beginning of a line as the first character of a regular expression.

- `$` matches the end of a line as the last character of a regular expression.
- `\` to escape meta-characters
- `"..."` interprets everything within the quotation marks literally
- `/` matches the preceding regular expression but only if followed by the following regular expression.

JFlex requires a white space to separate a regular expression and its actions; otherwise JFlex will read it as different.

4.1.3 An Implementation of a Z Scanner

After two above sub-sections about a brief introduction to JFlex, a scanner generator which was used to implement our Z scanner, and a brief description on lexical specifications, this sub-section discusses our Z scanner. Thus, our Z scanner was implemented using JFlex.

Our Z scanner was implemented so it can scan several Z tags. In other words, our Z scanner does not support all Z tags. For a complete list of Z tags which can be scanned by our Z scanner, please see Appendix E on page 287.

As mentioned earlier, our scanner does not scan all Z tags as well as not all of our Z tags were accompanied by actions. Reasons behind the first statement are to be in line with Z2SAL as the translator does not support all Z tags. Thus, there is no point here to be able to scan a token represents a Z tag which is not supported by Z2SAL and sometimes it is not available also on [69]. Other reason for us not to include all Z tags is that it is not difficult to add a new token. Another one is our Z specifications could be scanned by our Z scanner, though this scanner does not support all Z tags. In other words, this scanner has implemented a list of Z tags which are suitable to our Z specifications. Several Z tags which were not specified in our Z scanner are: `\nexi`, `\nexists` for representing " $\#$ "; `\bool` for " \mathbb{B} "; `\iter` for "*iter*"; `\pred` for "*pred*"; `\post` for "*post*"; `\items` for "*items*"; `\bagcount` for "*count*"; `\buni` for " \cup "; `\varsdef` for " \triangleq "; R^+ for transitive closure; R^* for reflexive-transitive closure.

Let us move to three parts of our scanner. There was no user code which was put in the first part of our JFlex specification. The name of our JFlex specification is **Lexer.flex**.

For the second part, the `%byacc` directive was added. Another directive was added in this part, a directive to indicate a name of the generated Java

file. In this scanner, it was defined as `ScannerCl`. At first, `Scanner` was chosen, but then it turned out that the latter is one of Java class names. There are two methods specified in this second part. The first method is a constructor for the generated Java class. The second one is a method to get the line number of a particular line of our `Z` specification. This method is called in actions of `."` of our regular expression to indicate a lexical error. There were also declarations of two variables in this part. Both methods and these two variables were enclosed by `"%{"` and `"%}"`.

`Z` tags were specified in the third part. Several `Z` tags that have actions in them, these actions are quite similar in all these tags. The first action is to assign a matched tag which is returned by `yytext()` as a semantic value for the associated parser, shown as follows:

```
yyparser.yylval = new ParserVal(yytext());
```

The JFlex must store this value in `yylval` before it is returned. The routine `yyparser()` is the parser generated by YACC.

The second action is to return a token of the matched tag to the parser.

```
return Parser.BZED;
```

Above is an example of the action to return the `BZED` token to the parser. This token indicates `\begin{zed}` tag. Among these tags, not all of them were implemented with the first action.

JFlex matches input texts to patterns constructed by regular expressions based on a set of simple disambiguating rules as follows [45]:

- JFlex patterns only match a given input character or string once.
- JFlex executes the action of the longest possible matched input texts.

Thus, if our scanner returns a lexical error while scanning a particular `Z` specification, this error can inform us several cases after a further check on this `Z` specification. The first case, the error means that the associated `Z` tag has not been specified in our scanner. Having this error, a solution is to add this tag to our scanner.

As an example is shown by the below output:

```
run:
file parse: E:\Google Drive\Tesis\program\JavaCode\Tesis\src\carspark.tex
Lexical error on line: 1 : \
C:\Users\MUS\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 3 minutes 5 seconds)
```

It was generated by our system during running the modified Cars Park spec-

ification (see Appendix C.7 on page 280). The first line of this specification has been changed to:

```
\documentstyle[11pt,oz]{article}
```

Our scanner only recognizes 12pt as the font size.

The second case, the error means the tag, which is available in our scanner, has been written wrongly. Thus, the associated tag will be rewritten precisely.

Using the same example as above, below is the output generated by our scanner:

```
run:
file parse: E:\Google Drive\Tesis\program\JavaCode\Tesis\src\carspark.tex
Lexical error on line: 4 : \
C:\Users\MUS\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 1 minutes 2 seconds)
```

It is because the example has also been modified. Its fourth line is misspelt into:

```
\begin{schem}{CarsPark}
```

This lexical error should be fixed since the error will push the system to stop immediately. In order to proceed to the Z parsing, it indicates no lexical error which means all Z tags on associated Z specification are recognized as true Z tags and specified in our Z scanner.

By using the JFlex scanner generator, there were 1,746 states during a Non-Deterministic Finite Automaton (NFA) construction of our scanner. This large number of states was reduced to 778 states in a DFA construction before minimization and it was reduced again to 566 states in minimized DFA. There was neither error nor warning detected by the JFlex scanner generator. This is shown by Fig. 4.1 on page 95.

In a case the scanner generation is successful; the generated Java file will be generated. This Java file is located in the same place as the JFlex specification. This generation will generate the **ScannerCl.java** file from our scanner.

4.1.4 Conclusion

Although our Z scanner does not implement all Z tags, these tags were suitable for our experiments. As a result, this scanner provided our Z parser with accepted Z tokens. In addition, all of our Z specifications could pass the scanning.

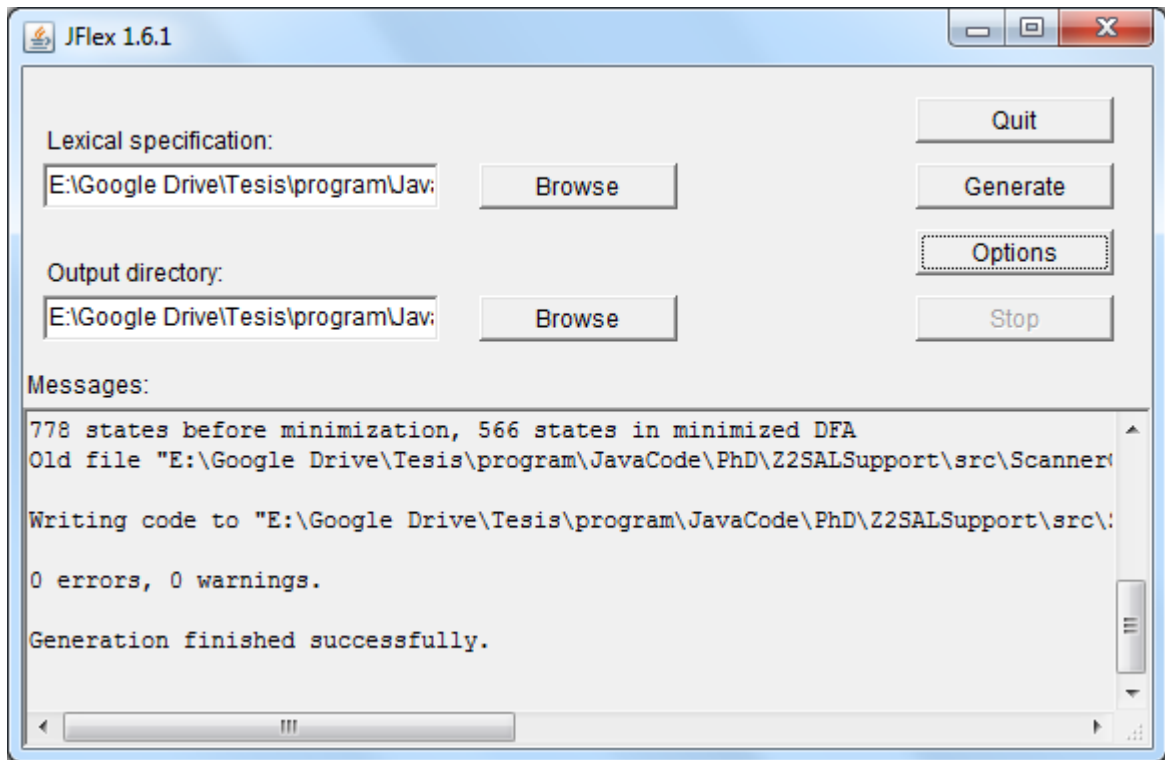


Figure 4.1: The JFlex scanner generator

It shows also that our Z scanner works. This success is expected as our contribution to related research in this field. The Z scanner is part of the architecture of our research as shown by Fig. 3.1 on page 77.

Furthermore, it is easy to add new Z tags to this scanner; just add the associated tags and add appropriate actions. Please remember to run the JFlex scanner generator for every modification which is performed on the JFlex specification. The old Java still exists, but its name will be ended with a tilde, "~~".

Based on our experiments, our Z scanner could correctly scan Z tags on our Z specifications. Since our Z specifications usually were obtained from Z books so it is argued that Z tags on our examples are accurate. Thus, Z tags on our scanner were also specified precisely since the scanner could scan the Z specifications. Other proof is tokens generated by our scanner could be parsed correctly by our Z parser. It means that a sequence of these tokens matches to our Z grammar which was taken from [69]. Another proof is the outcome of our system either a generic constants redefinition or a schema calculus expansion could be translated by Z2SAL. It means that

Z2SAL could parse associated Z specification.

As mentioned earlier, it is not difficult to add new Z tag to our scanner. It denotes the implementation of a Z scanner is not a big problem. As the outcome of our scanner will be processed further on our Z parser and continued to redefinition or expansion system, the use of the third party scanner generator makes the implementation of support for model checking Z specifications easier rather than to hand-write a scanner generator by using the Java language.

Let us now move to another discussion on this chapter. It is a design and implementation of a Z parser.

4.2 A Z Parser

This section describes the implementation of our Z parser. Our parser will read tokens passed by our scanner, and try to process whether these tokens match any rule in Z grammar specified in our Z parser. This section contains several sub-sections, which begins with an introduction to the BYACC/J parser generator.

4.2.1 Introduction

An extension to the Berkeley 1.8 YACC-compatible parser generator is used in our support [33]. It is BYACC/J, which is available in Microsoft, Linux, Macintosh, and SUN Solaris platforms [58], with version 1.15. By a flag "-J", the standard YACC tool will generate one or more Java parser files from a YACC source file .y. However, BYACC/J can also generate C/C++ parsers [58]. These Java files can be compiled to produce a LALR-grammar parser.

One of these files, which is usually generated, is the **Parser.java**. By reading this file, a user can see how a parsing algorithm of YACC works. This Java file generates a class which is an extension of **Thread**.

Another Java file is the **ParserVal.java**. The current version of BYACC/J allows a user to define an **int**, a **double**, a **String**, or an **Object** values. All these semantic values are stored in a public class, **ParserVal**.

4.2.2 A YACC Parser

JFlex recognizes regular expressions and returns tokens from the matched inputs, whereas YACC recognizes grammars in which it groups tokens logically [45]. Thus, YACC takes a grammar, which is specified by a user, and writes a parser that recognizes valid sentences in that grammar.

The BYACC/J parser generator supports all usual procedures of YACC. A YACC specification consists of three parts as follows: declarations area, actions area, and code area. They are separated by ”%%” at the start of a line given as follows:

```
DECLARATIONS
%%
RULES AND ACTIONS
%%
CODE
```

The first section includes declarations of the tokens used in the grammar, the types of values used in the parser stack, precedencies and associativity of operators, and a literal block Java code is enclosed by ”%{” ”%}”. As a position of an operator in the precedencies gets higher, its level of precedence gets lower. There are three associativities: %left, %right, and %nonassoc.

A single quoted character can be used as a token directly without prior declaration in it. A token can be declared with a particular type: <sval> for a string value and <dval> for a double value. These values are used also in the declaration of a type of a terminal symbol.

The second section consists of a list of grammar rules. A colon, ”:”, is used to separate the left from right-hand side and a semicolon is used to indicate the end of each rule. The symbol in the left-hand side of the first rule is normally the start symbol. However, if the %start directive is declared, then this directive will override the start symbol.

By default, the first rule is the highest-level rule [45]. In other words, a parser will find a list of tokens that matches this initial rule.

A rule can be a recursive rule which gives ability for YACC to parse long input sequences. A user can write their rules to be left recursive or right recursive. However, YACC handles left recursion much more efficiently than right recursion [45].

A symbol in YACC parser has its value which can be obtained by using the \$ symbol. This value symbol has an index commencing from 1 to indicate the first symbol in the right-hand side of a colon. The left-hand side has the \$\$ symbol for its value. The other name for left-hand side symbol is a non-terminal, a symbol which cannot stop the parsing.

The right-hand side contains of symbols and might have action code. The symbol might be a terminal symbol or a token. The action code is enclosed by ”{” and ”}”. This side can consist of several vertical bars, ”|”, which indicate that several symbols in the right-hand side have the same left-hand side symbol.

The last section is used to specify code in Java language which then will be copied verbatim to the parser. It is called the user subroutines section.

A YACC parser works by looking for rules that match the tokens seen so far. This parser will create a set of states, which each state reflects a possible position in one or more partially parsed rules.

Each time a parser reads a token that does not complete a rule; it pushes the token in an internal stack and switches to a new state reflecting the token that has just been read. This action is called a *shift*.

If a parser has found all the symbols in the right-hand side of a rule, it pops these symbols off the stack, pushes the left-hand side symbol onto the stack, and switches to a new state reflecting the new symbol in the stack. This action is called a *reduction*, since it usually reduces the number of items in the stack. Whenever YACC reduces a rule, it executes user code associated with this rule.

However, YACC sometimes might fail to parse a grammar specification file. Ambiguous or conflicts in the grammar can fail a parsing.

An ambiguity means that there are more than one possible parse trees for a single input string. This ambiguity can be inherently in several grammar specifications. For this case, YACC cannot handle this ambiguous grammar.

On the other hand, the grammar might not be ambiguous. The parsing technique that is used by YACC is not powerful enough to parse the grammar. This unambiguous grammar raises conflicts that push the parser to look many tokens ahead to decide which of two possible parses to use [45].

There are two conflicts when YACC tries to parse a grammar specification: shift/reduce and reduce/reduce conflicts. The next sub-sections discuss ambiguity and conflicts on parsing an input file.

Shift/Reduce Conflict

This conflict occurs when there are two possible parses for an input string. One of these parses can complete a rule, so a reduce option is applicable. Another parse cannot complete such a rule, which applies a shift option.

For example, the below grammar has one shift/reduce conflict:

```
%%  
e : 'X'  
  | e' +' e  
  ;
```

If there is an input "X+X+X", then there will be two possible parses: "(X+X)+X" or "X+(X+X)". The reduce option makes the parser use the first parse or the reduce imparts left associativity to the operator; otherwise the shift one uses the second parse or the shift imparts right associativity. YACC always chooses the shift unless operator precedencies or associativity

were declared. On the other hand, addition operator should be left associative. Thus, shift/reduce conflict occurred here. This will be discussed briefly in the associated sub-section.

Reduce/Reduce Conflict

A reduce/reduce conflict occurs when the same token can complete two different rules. For example:

```
%%  
prog : prog | prog;  
prog : 'X';  
prog : 'X';
```

Although usually the reduce/reduce conflict is less obvious than one shown in the above example. This conflict indicates a mistake in a grammar.

In a case there is a reduce/reduce conflict, Yacc will reduce the earlier rule or the rule with the smaller number of order [39].

Precedence, Associativity, and Operator Declarations

As mentioned earlier in the previous section (see Section 4.2.2 on page 98), the default action for YACC to choose if there is a conflict is to take shift option. However, sometimes the result is not as expected. Thus, YACC includes operator declarations that let a user change the way YACC handles shift/reduce conflicts.

Precedence controls which operators will be executed first in an expression, whereas an associativity controls the grouping of operators at the same level of precedence or at the same operators. In other words, precedence assigns a level to an operator. Operators at higher levels bind more tightly. As mentioned earlier, associativity can be left which means operators are grouping from the left, right which means operators are grouping from the right, or no grouping.

Operators are declared in increasing order of precedence. The same line of operators means they have the same level of precedence. To have another group of operators without continued the level of precedencies of previous groups, the new group must be declared in a different group from previous groups. Both groups are separated by a blank line between them.

In practice, it is better to rewrite a grammar rather to benefit from precedence to solve the conflict. The common use of precedence is to solve 'dangling else'.

Table 4.1: A list of unspecified Z rules

LHS	RHS
schema.expr1	pre schema.expr1
pred	<i>let</i> Let-Def.list • pred
Let-Def.list	Let-Def Let-Def.list
Let-Def	ident == expr (op.name) == expr
op.name	<i>_in-sym decor_</i> <i>_pre-sym decor_</i> <i>_post-sym decor</i> <i>_(_)decor</i> <i>_decor</i>
pred1	PREREL decor expr pre schema.ref
expr0	μ spot.tail μ word.schema.text <i>let</i> Let-Def.list • expr
expr	<i>if</i> pred <i>then</i> expr <i>else</i> expr
expr4	expr4 ^{expr}
in-sym	INFUN INGEN INREL
pre-sym	PREGEN PREREL
post-sym	POSTFUN

4.2.3 An Implementation of a Z Parser

Our Z parser can be seen in Appendix F on page 305. It was represented by a BYACC/J specification, which was named as **Parser.y**. Our parser does not implement all Z rules. The Z grammar in our BYACC/J specification refers to [69].

Several Z rules that were not specified by our BYACC/J specification are given in Table 4.1 on page 100. Z rules, which were listed in Table 4.1 on page 100, have not been implemented because of several reasons. The first reason is our examples do not contain any declaration or predicate which match one of those rules. Another reason is several of those rules caused the number of shift/reduce or reduce/reduce conflicts is even higher. Since then they were not included in our Z grammar.

If our parser is given a Z specification which has a declaration or predicate statement does not match any of our Z rules, then our Z parser will generate a syntax error. This error can fall into several sources. The first source is our grammar does not have a rule of such a declaration or predicate statement. Solving this problem is by adding this new rule. This might be necessary to check also relevant tokens to specify the rule since it can be such a token has not been specified in our scanner. The second source is indeed the rule of

the declaration or predicate statement has been specified in our Z grammar. However, there were conflicts on either declaration or predicate. For this case, a solution requires a further check on available grammars and solve any shift/reduce or reduce/reduce conflict if it exists.

For example is given by output below:

```
run:
file parse: E:\Google Drive\Tesis\program\JavaCode\Tesis\src\carspark.tex
syntax error
Please check line: 4
\nat with length: 4
BUILD SUCCESSFUL (total time: 34 seconds)
```

It was generated from the same example used in Section 4.1.3 on page 92. This time, the statement in line 5 has been modified incorrectly into:

count \nat \\\

Our scanner counts the number of line from 0. Thus, the real number of line should be added with 1.

The suspicious line is part of a declaration in a schema. Our parser expected that there is : between name and type of a variable.

In the first part of our BYACC/J specification, several imported packages were declared. The first half tokens were declared having string values, whereas the second ones have not had any values. Tokens were specified using capital letters. All types of terminal symbols in our parser have also string values.

Precedencies and associativities of several Z operators, which were formulated in our parser in this first part, can be seen in Table 4.2 on page 102. These precedencies and associativity follow ones specified in [69], but not the last two lines. Both these lines were specified by us.

The second part of our parser contains almost Z rules which were obtained from [69]. However, several of them have been rewritten to avoid shift/reduce and reduce/reduce conflicts. Although these conflicts exist on our parser, the numbers are less than the numbers of the same conflicts on original Z grammar. Not all our rules were accompanied by actions. These actions store information which will then be used on further process either redefinition or expansion.

One example of our Z rules is discussed here. It is a Z rule to parse a schema calculus definition. Our parser supports also many lines in one schema calculus definition box. It is since our parser passes information

Table 4.2: Precedencies and associativity of Z operators

Tokens	Precedencies	Associativity
PIPE	11 th	Left
SEMI	10 th	Left
HIDE	9 th	Left
PROJECT	8 th	Left
BIMPLIES	7 th	Left
IMPLIES	6 th	Right
OR	5 th	Left
AND	4 th	Left
NOT	3 rd	Non-association
'(2 nd	Non-association
)'	1 st	Right

about a separator on each different schema calculus definition. A separator which separates each line containing different schema calculus will be put on the `l1SchCal` list. This list will be used later in the schema calculus operation.

The associated Z rules to process the separator is shown as follows:

```

schema.def.horz: WORD SDEF
{
    schCal = true;
    if (separator){
        l1SchCal.add("separator");
        separator = false;
    }
}
schema.exp
| WORD gen.formals SDEF
{
    // has the same code as for WORD SDEF
}
schema.exp
;

```

The `schema.exp` non-terminal can be matched by two rules. One of them is `word.schema.exp1` and it will match with either `schema.exp1` or `WORD`. The latter is a terminal, in this parser, it is a Z token which a firing on it will store the token information on the `l1SchCal` list.

```

word.schema.exp1: schema.exp1
| WORD
{
    l1SchCal.add($1);
}
;

```

On the other hand, the `schema.exp1` non-terminal will store information to the list shown as follows:


```

schema.exp1: LSBRACK
    {
        l1SchCal.add($1);
    }
word.schema.text RSBRACK
    {
        l1SchCal.add($4);
    }
| schema.ref
...
| NOT word.schema.exp1
...
| word.schema.exp1 AND word.schema.exp1
...
| word.schema.exp1 OR word.schema.exp1
...
| word.schema.exp1 IMPLIES word.schema.exp1
...
| word.schema.exp1 BIMPLIES word.schema.exp1
...
| word.schema.exp1 PROJECT word.schema.exp1
| word.schema.exp1 HIDE '( 'WORD " ' ' )'
...
| word.schema.exp1 HIDE '( 'word.decl.name.list ' )'
...
| word.schema.exp1 SEMI word.schema.exp1
...
| word.schema.exp1 PIPE word.schema.exp1
| '( ' schema.exp ' )'
...
;

```

”...” can be seen in Appendix F on page 305.

The third part of our parser as can be seen in Appendix F on page 305 consists of declarations of several variables which were used on our actions. There is also a reference to our scanner. This will be discussed further on a relevant chapter later. There are several functions specified in this part. The first function is to establish an interface to our scanner. The second one is to report any syntax error that has been found. This function will call another function to perform this job. Another function is a constructor of the generated Java file later.

Before a parser generation is performed, a BYACC/J specification must be copied to the place at which JFlex generator is located. The command to generate our parser is as follows:

```
C:\jflex-1.6.1\bin>yacc -J Parser.y
```

Our Z parser generated two Java files at the end of this generation. In our case, they are **Parser.java** and **ParserVal.java**. Then, both these files are copied again to the place at which the BYACC/J specification is defined.

In addition to both Java files, inevitably, our Z parser generated also several warnings. These warnings relate to conflicts with our parsed Z grammar. The warnings are:

```
yacc: 1 shift/reduce conflict, 5 reduce/reduce conflicts.
```

These warnings have not been solved. It requires time and an effort to an elaborate check on the grammar and a rewriting in it. However, all of our examples could be parsed by our parser. Based on information gathered from actions defined in our parser, the way our parser was designed is sufficient and it could be said that our parser parse the input correctly.

Thus, these warnings are left as future works. In other words, until now, our system either a generic constants redefinition or a schema calculus expansion could run all of our examples correctly, though there were conflicts with our parser.

Fortunately, YACC provides also an output file during the parser generation. To obtain the output file, the above generation command is modified as follows:

```
C:\jflex-1.6.1\bin>yacc -v -J Parser.y
```

A file named **y.output** as default is generated after the above command is executed. This file contains the parse table of the parser. The parse table could be checked if there is conflict with the grammar.

Our parse table contains 382 states, 86 terminals, 95 non-terminals, and 220 grammar rules. The above conflicts are informed also as follows:

- State 69 contains 1 shift/reduce conflict.
- State 137 contains 1 reduce/reduce conflict.
- State 151 contains 1 reduce/reduce conflict.
- State 159 contains 1 reduce/reduce conflict.
- State 199 contains 1 reduce/reduce conflict.
- State 202 contains 1 reduce/reduce conflict.

However, it is possible that there are errors in gathered information if further type-checker or processing is added to our parser. Furthermore, it might these conflicts make our system fails to run other *Z* specifications. This case is beyond our expectation now.

Nevertheless, let us investigate which design of *Z* specifications could trigger problem on this conflict. Further information is given in separate sub-sections.

The First Conflict

The first conflict is a shift/reduce conflict on State 69. Its detail given by the output file is as follows:

```
69: shift/reduce conflict (shift 138, reduce 121) on ','
```

The shift/reduce conflict relates to comma (","). This conflict is either shift to State 138 or reduce to Rule 121. State 138 is as follows:

- `decl.name.list` : `WORD ' ' , ' . WORD ' ' (168)`
- `decl.name.list` : `WORD ' ' , ' . word.decl.name (170)`

On the other hand, Rule 121 is given on the below list of information on State 69.

Number inside brackets at the end of each rule is the number of rule. Thus, a shift requires to shift to other state, whereas a reduce requires to reduce to other rule number.

State 69 consists of other information as follows:

- `head.gen.actual.opt.rename.opt` : `WORD ' ' ' . (121)`
- `head.gen.actual.opt.rename.opt` : `WORD ' ' ' . gen.actual.opt.rename.opt (122)`
- `basic.decl` : `WORD ' ' ' . ':' expr.word (165)`
- `decl.name.list` : `WORD ' ' ' . ', ' WORD ' ' ' (168)`
- `decl.name.list` : `WORD ' ' ' . ', ' word.decl.name (170)`

The period, ".", points to the current location of the parse in the input string [58]. The first item calls for a reduce (because the period is the right-most) on ",". However, the fourth and five items in this state call for a shift on this input. The second item calls for a shift on "[", and the third item calls for a shift on ":". Thus, it is a shift/reduce conflict. YACC will shift on this case as it is its default action.

The fourth and five items could raise an error if in a Z specification there is a declaration of two schema inclusions. It is because a comma in this list, YACC will shift to read the next item. On the other hand, each item in this list needs to be reduced to a rule of schema reference.

The second item could raise an error if a schema inclusion is followed by a "[". It is because an open square bracket in this case, YACC will shift to read the next item. On the other hand, a schema inclusion needs to be reduced to a rule of schema reference.

The third item could raise an error if a schema inclusion is followed by a ":". It is because a colon in this case, YACC will shift to read the next item. On the other hand, a schema inclusion needs to be reduced to a rule of schema reference.

None of our Z specifications were designed to have either one of these four rules. Thus, our tool does not experience with errors originated from this shift/reduce conflict. Otherwise, each of the above cases will generate a syntax error.

The Second Conflict

Different with the first conflict, the second is a reduce/reduce conflict. For this conflict, YACC will run the earlier ordered rule in the related grammar.

The detail of this conflict is as follows:

137: reduce/reduce conflict (reduce 91, reduce 148) on ','

This conflict offers options either reduce to Rule 91 or Rule 148 on a reading of ",". Other information on this state is as follows:

- `expr4` : `word.decor` . (91)
- `expr4` : `word.decor` . `word.sq.bracks` (92)
- `expr4` : `word.decor` . `gen.actual.opt` (93)
- `decl.name` : `word.decor` . (148)

The first and last items call for a reduce on ",". However, the first item will reduce to Rule 91 and the last will reduce to Rule 148. YACC will reduce to the earlier specified rule based on the input sequence. The second and third items call for a shift on "[".

Based on our running on our Z specifications that were used for redefinition generic constant and expansion of schema calculus, none of these Z specification gave us error on such conflict. Thus, it could be said that our Z specification examples are parsed correctly on this conflict. This conflict would generate a syntax error if it is expected that the item will reduce to Rule 148, but YACC reduce to Rule 91 and the next input does not match with the rule, and vice versa.

The Third Conflict

The third conflict is also a reduce/reduce conflict. The detail of this conflict is given as below:

151: reduce/reduce conflict (reduce 161, reduce 198) on ')'

This conflict originates from a reading of ")'" and causes two options whether to reduce to Rule 161 or Rule 198. Other information on this state is given as follows:

- `expr1.word` : WORD . (71)
- `expr2.word` : WORD . (84)
- `expr4.word` : WORD . (86)
- `expr3.word` : WORD . (117)
- `schema.ref` : WORD . gen.actual.opt.rename.opt (120)
- `head.gen.actual.opt.rename.opt` : WORD . ''' (121)
- `head.gen.actual.opt.rename.opt` : WORD . ''' gen.actual.opt.rename.opt (122)
- `word.decor` : WORD . DECOR (151)
- `pred` : WORD . (161)
- `word.pred1` : WORD . (195)
- `expr.word` : WORD . (198)

The third last and last items call for a reduce on ")'". The third last item reduces to Rule 161 whereas the last reduces to Rule 198. The fifth item calls for a shift on "[". On the other hand, the sixth and seventh call for a shift on "'", and the eighth calls for a shift on DECOR. The rests call for a reduce to a rule as given inside brackets.

Based on our investigation on our Z specification examples, this conflict does not occur. It is sufficient to say that our examples are parsed correctly by YACC on this conflict.

The Fourth Conflict

Detail of this conflict is as follows:

159: reduce/reduce conflict (reduce 98, reduce 185) on ')'

This error means that there is a reduce/reduce conflict when YACC reads a ")'" after each of the below rules. It either can reduce to Rule 98 or Rule 185. Other information is as follows:

- `expr4` : `schema.ref` . (98)
- `pred1` : `schema.ref` . (185)

This conflict does not occur. As a result, it could be said that our Z specification examples are parsed correctly on this conflict by our parser.

The Fifth Conflict

The fifth conflict is as follows:

199: reduce/reduce conflict (reduce 173, reduce 198) on ','

It relates to choices for YACC to reduce to Rule 173 or Rule 198. Other information on this state is given as follows:

- `expr1.word` : `WORD` . (71)
- `expr2.word` : `WORD` . (84)
- `expr4.word` : `WORD` . (86)
- `expr3.word` : `WORD` . (117)
- `schema.ref` : `WORD` . `gen.actual.opt.rename.opt` (120)
- `head.gen.actual.opt.rename.opt` : `WORD` . ']' (121)
- `head.gen.actual.opt.rename.opt` : `WORD` . ']' `gen.actual.opt.rename.opt` (122)
- `word.decor` : `WORD` . `DECOR` (151)
- `schema.text` : `WORD` . `BAR pred` (156)
- `basic.decl` : `WORD` . ']' ':' `expr.word` (165)

- `decl.name.list` : `WORD . ' ' , ' WORD ' ' (168)`
- `decl.name.list` : `WORD . ' ' , ' word.decl.name (170)`
- `word.decl.name.list` : `WORD . (173)`
- `word.schema.text` : `WORD . (183)`
- `expr.word` : `WORD . (198)`

Based on our investigation on our *Z* specification examples, this conflict does not occur. It could be said that YACC could parse our examples correctly on this conflict.

The Sixth Conflict

The last conflict generated by YACC is as follows:

202: reduce/reduce conflict (reduce 98, reduce 164) on TES

The right hand side of both below rules could be reduced either to the first rule or second on a reading of "TES". "TES" is the closing marker for a set definition.

Other information on this state is given as follows:

- `expr4` : `schema.ref . (98)`
- `basic.decl` : `schema.ref . (164)`

There is none of our examples experienced with this conflict.

4.2.4 Conclusion

A successful parsing of our parser provides important information for a generic constant redefinition or a schema calculus expansion system. It shows that our *Z* parser works. This success is expected as our contribution to related research in this field. The *Z* parser is part of the architecture of our research as shown by Fig. 3.1 on page 77.

The outcome of a generic constant redefinition or a schema calculus expansion system will be translated by Z2SAL. If a user can obtain a generated SAL file from this translation, it is argued that our *Z* parser has parsed an associated *Z* specification accurately. On the other hand, if Z2SAL fails to translate the outcome of our system, it does not denote that our parser is not correct. It might be the associated *Z* specification contains *Z* notations which

does not supported by Z2SAL, as in one of our example of Z specification. Since our examples on our experiments can be translated by Z2SAL with an exceptional Z specification as mentioned earlier, our parser could parse Z specifications on our experiments correctly.

The use of the third party parser generator makes the implementation of support for model checking Z specifications easier rather than to hard-code a parser generator on the Java language. If a user is necessary to add a new rule, it can be performed easily on the existing parser.

As several conflicts still exist, further studies on YACC software as well as parser are required to solve these conflicts. Although it is only a warning in this tool, it says there is something not correct with the grammar. In our cases, these conflicts relate to rules that have only one non terminal in the right hand side of colons, whereas this non terminal was designed to exist in many different rules.

The output file provided by YACC worth as guidance in solving conflict on this tool. It is because this file supplies us with the next input, number of states, number of rules, and other useful information. These conflicts can be defined as future works.

Chapter 5

Redefining Generic Constants

The previous chapter discussed implementations of our Z scanner and parser. This chapter discusses a redefinition of generic constants. It begins with a brief introduction to generic constants.

5.1 Introduction

A generic constant is used to introduce a new constant [2]. Syntax for the generic constants was taken from [69] and the associated syntax for a generic constant is:

$[Gen - Formals]$
<i>Declaration</i>
<i>Predicate; ...; Predicate</i>

This box looks like a schema box, but the former box has several differences on its header to the latter box. Firstly, it is not necessary to specify a name for a generic constant definition box as in a schema box, but it is required to add generic parameters to its heading. The generic parameter is specified inside a pair of square brackets. Another difference is a double line instead of a single line on the upper line of the former box.

Not only does a generic constant definition let a user define generic parameters with the same type, but also a user can define different types of parameters. It is performed by using different literals such as **X**, **Y**, **Z**, and others.

In a case there is no such a generic constant, a user should define several different functions based on their types of parameters, though contents of these functions are all the same. Thus, it involves redundant work.

A generic constant is commonly used in defining mathematical tool-kit operators [2], which do not depend on existing elements in their constructions

[69]. Thus, a user can define a new mathematical operator easily without being bothered by types of each its element.

A generic constant can also define a general notion of a system, which in fact is used frequently. This general notion helps a user to gain more understanding on the system.

By using a generic constant, a user can define parameters that might have different types. It is the same as the generic schema which can be defined whatever data that will be operated on it.

An example of a generic constant taken from [69] is given as follows:

$$\frac{\frac{}{[X, Y]}}{\text{first} : X \times Y \rightarrow X}}{\forall x : X; y : Y \bullet \text{first}(x, y) = x}$$

where X and Y might have any type either a different or same type. As seen above, the **first** generic constant has two generic parameters. This generic constant can be used to find the first element between two elements.

These types can be specified such that they are the same. It is performed by using one generic parameter showed as follows:

$$\frac{\frac{}{[X]}}{\text{first} : X \times X \rightarrow X}}{\forall x, y : X \bullet \text{first}(x, y) = x}$$

A generic constant can be redefined to an axiomatic definition by specifying actual types replacing generic types explicitly. In other words, a formal generic parameter of a generic constant is usually formed from a single capital letter, which will be actualised by defining its type based on usages of the generic constant. A process of actualising is called with an instantiation [56].

For above example, if there is a usage for the generic constant **first** such as **first(2,5)**, the formed type is a set of $((Integer \times Integer) \times Integer)$. This expression has inputs whose type is *Integer* and an output whose type is *Integer*.

However, in a case of a generic constant that does not have a parameter/input, actualising its parameter is not always straightforward. It requires deduction to infer a type from the generic constant environment.

Furthermore, there are two methods on actualising generic parameters. The first method is defining directly actual types. A user can define the type by putting it inside a pair of square brackets, "[]", following either the generic constant name or the actual parameters. This method is named explicit actual types.

The second one is to infer the actual type from its surrounding; it is implicit to the user. The second method seems a challenge in designing an

automatic actualisation system, especially with a generic constant without an input.

Below is further discussion on redefining generic constants.

5.2 A Method for Redefining a Generic Constant

Our method to support Z2SAL in translating a generic constant is to implement a tool which will redefine a generic constant definition to an equivalent axiomatic definition based on usages of the generic constant. This redefinition is called as an actualization process, in which a generic type of a parameter will be actualised to its actual type of parameter. Plagge and Leuschel in [55] also proposed the same method as our method for translating a generic definition defined in a Z specification.

The difference of both definitions is the former definition uses generic parameters, whereas the latter one uses no generic parameter. Based on this difference, generic parameters will be replaced by their actual parameters which match with their usages. Thus, an axiomatic definition is generated from this generic constant definition which matches associated usage.

Based on this method, the next section discusses an implementation of generic constant redefinition system.

5.3 An Implementation of the Redefinition System

A simple redefinition tool, which will redefine a generic constant definition in a Z specification to a suitable definition written by using an axiomatic definition, was proposed and implemented.

This system starts from our main program, `Z_Preprocessing_Tool.java`, specifically `redefine` function. From this function, other functions specified in another Java program will be called as they are required. In this case the latter program is `generic.java`. In general, our redefinition system has an algorithm as follows:

5.3.1 Reading a Z specification

This first step will read the Z specification input line by line. Bytes of characters were chosen as a reading method, and firstly it was read to a buffer as follows:

```

BufferedReader br =
new BufferedReader(new FileReader(file.getParent()+"\\output_"+
file.getName()));
// irrelevant code
while ((s = br.readLine())!= null){
// irrelevant code
}
// irrelevant code

```

To see how the added code works, this code will be applied to a Z specification. For this purpose, one of our Z specifications from our experiments, `fHead.tex`, was used. Thus, a discussion on this working example will accompany important code. This Z specification and its redefined version generated by our system can be seen completely in Appendix G on page 327.

5.3.2 Spotting Generic Constant Definition

This step will spot generic constant definition boxes and change them into relevant axiomatic definitions.

```

if (s.startsWith("\\begin{gendef}")){
  if (s.contains("X")) {
    for (int i=0;i<s.length();i++){
      if (s.charAt(i) == 'X') generic.genPar.add("X");
    }
  }
  if (s.contains("Y")) {
    // the same as earlier code, but the string should be matched
  }
  if (s.contains("Z")) {
    // the same as earlier code, but the string should be matched
  }
}

```

Each generic parameter which is found in a generic constant definition box will be stored. In this system, only three literals are considered for generic parameters: **X**, **Y**, and **Z**. They are shown in the above code. This number is chosen since it is rare to have a number higher than this. These literals are also commonly used to represent generic parameters.

Each generic constant begins with `\begin{gendef}`. After it is spotted, it is changed into the axiomatic definition. This is obtained by replacing `\begin{gendef}` with `\begin{axdef}` as shown in the below code, and `\end{gendef}` with `\end{axdef}` as shown in the later code.

```

s = s.replace(s,"\\begin{axdef}");
s = s.trim();
generic.tempS.add(s);

```

The code below gets variables and their types from generic constant names. There was only one generic constant variable, `headSeq`, and its type is $(seq_1 X) \rightarrow X$.

Types are then divided into two categories: they contain brackets, or they do not contain brackets.

```

while (!(s = br.readLine()).equals("\\ST")){
  if (s.contains(":")){
    var = s.substring(0, s.indexOf(':'));
    var = var.trim();
    val = s.substring(s.indexOf(':')+1);
    val = val.trim();
    if (val.endsWith("\\\\"+"\\\\")){
      val = val.substring(0, val.indexOf("\\\\"+"\\\\"));
      val = val.trim();
    }
    // see the below code for details
  }
  generic.tempS.add(s);
}

```

The code below gives details of the above comment.

```

if (val.contains("(")){
  llType = new LinkedList();
  for (int idxVal = 0; idxVal < val.length();
  idxVal++){
    if (val.charAt(idxVal) == '\\'){
      // irrelevant code
    }
    else {
      if (val.charAt(idxVal) != '_') {
        llType.add(val.charAt(idxVal));
      }

      stVal = "";
    }
  }
  int idv=0, idxVal, newIdv;

  while (idv < llType.size()){
    stVal = llType.get(idv).toString().trim();

    if (stVal.equals("(")&& generic.indexOfById(llType, "(" , (idv+1)) > -1){
      // irrelevant code
    }
    else if (stVal.equals("(")&& generic.indexOfById(llType, "(" , idv+1) == -1){
      // irrelevant code
    }
    else idv++;
  }
  ...
}

```

If they do not contain bracket, the code is just a switch block shown as follows which details the above "...":

```

switch (Parser.hmTypeVar.get(var).toString()) {
  case "isFunction":
    generic.genCons.put(var, val);
    generic.hmFunc.put(var, val);
    break;
  case "isRelation":
    generic.genConsNotFunc.put(var, val);
}

```

```

    break;
default:
    generic.genConsCons.put(var, val);
    generic.hmFunc.put(var, val);
    break;
}

```

At this stage, the type of the generic constant is gathered also. It can be a function, a relation or a constant. They can be distinguished based on the generic constant declaration, given as follows:

- a function; if the outermost operator is one of infix generic functions. A complete set of these functions is " \rightarrow ", " \rightarrow ", " \rightarrow ", " \rightarrow ", " \rightarrow ", " \rightarrow ", and " \rightarrow ". These functions are collected in one token, namely **INGEN**. Being a function, it will have at least one input parameter and one output parameter. These parameters can have generic types.
- A relation; if a declaration uses " \leftrightarrow " tag in its outermost operator. This tag has **REL** as its token. As a relation, there is no output parameter, in other words the output is the relation itself, a pair of typed parameters.
- A constant; a constant means it does not require any input. Thus, a declaration of this generic constant only gives us the generic output parameter. This declaration denotes none of above tags in its outermost declaration.

These types of generic constant are formulated by following expressions which were specified in our parser:

```

expr1:  expr1.word REL decor
        {
            isRelation = true;
            if (genAxDef){
                hmTypeVar.put(llGenAxDefVar.getLast().toString(),
                    "isRelation");
                llGenAxDefTypes.add($2);
                if (!$3.isEmpty()) llGenAxDefTypes.add($3);
            }
            else if (schDef){
                ...
            }
        }
        expr1.word
|  expr1.word INGEN decor
  ...
|  expr1.word
  expr2.chain
  ...
|  expr2
  ...
;

```

"..." can be seen in Appendix F on page 305. The first rule indicates a relation, whereas the second one is a function. The third rule contains **CROSS** obtained from `expr2.chain`. This rule can be either a function or a relation, depending on which of the first two rules are fired previously. The last rule is a constant; it denotes that the function and relation rules cannot be matched.

If a type does not contain any bracket, the next process is to identify directly the type of generic constants which is the same as the above code. Afterwards, each declaration line will be stored to the list. It is shown by the last statement before the end of while block.

The next process is to get predicates. It is shown by the code as follows:

```

if (s.equals("\\ST")){
    LinkedList vcStr = new LinkedList();
    generic.tempS.add(s);
    s = br.readLine();
    ss = "";
    while (!s.equals("\\end{gendef}")){
        // see the below code
    }
}

```

If there is a quantifier in the predicate part, the associated lines will be joined until it reaches either a line separator or the end of this generic constant definition.

```

if (s.trim().endsWith("\\dot") || s.trim().
endsWith("\\spot") || s.trim().endsWith("\\cbar")){
    ss = s;
    while (!s.endsWith("\\"+ "\\") && !s.trim().equals("\\end{gendef}")){
        s = br.readLine().trim();
        if (!s.trim().equals("\\end{gendef}")){
            ss = ss + " " + s;
        }
    }
}

```

Our Z specification did not contain a quantified predicate as mentioned on above code. In order to show how the above code works, the associated quantified predicate is divided into two lines, though this predicate is a quite short predicate.

After these lines are joined, another function will be called. `getActualParam` is a function to identify usage of each known generic constant and get types of actual parameters. This function was specified in `generic.java` program. Since our system supports no usage on a generic constant definition box, both calls will not give any result.

This function calls another function at the beginning of its code, which is `getVarInPredicate`. The latter function which was specified in `generic.java` is to get variables declared in a predicate part. These variables and their associated types will be stored according to the value of the last parameter

passed to `getActualParam`. If the value is true as the below code, it indicates the variable is declared on a predicate part of a generic constant definition box. Otherwise, the variable is declared on a predicate part of schemas or axiomatic definitions.

A call to `getActualParam` function is shown as follows:

```

if (!ss.isEmpty()){
    generic.getActualParam(ss,vcStr,genConstant,true);
    if (!s.equals("\\end{gendef}")) s = br.readLine();
}
else{
    while (!s.equals("\\end{gendef}")){
        generic.getActualParam(s,vcStr,genConstant,true);
        s = br.readLine();
    }
}

```

Since a variable `s` was declared on a predicate part of the generic constant definition `headSeq`, this variable and its type `seq1 X` are stored on a storage `hmVarGC` containing variables declared on generic constant definitions. This storage is an instance of `HashMap`, a Java standard library class.

Afterwards, the process returns to the caller of `getActualParam` function, which is `redefine` function. If the end of this generic constant definition is reached, then the next process is as the below code. The end of a generic constant is changed to the end of an axiomatic definition. A header and footer of a generic constant definition has been changed to both of them of an axiomatic definition.

```

if (s.equals("\\end{gendef}")){
    s = s.replace(s,"\\end{axdef}");
    s = s.trim();
    generic.tempS.add(s);
    idxEAXDef = generic.tempS.size()-1;
}

```

Thus, after this step, every generic constant definition has been modified to an axiomatic definition. However, they still use generic parameters. The next step will change these generic type parameters into actual type parameters.

5.3.3 Spotting Usages of Generic Constants

This step will spot usages of those generic constants. It also reflects these by both writing verbatim the definition of those generic constants, in case they are not defined before or they have different types of parameters, and modify their callers. These usages will be searched in schema definitions shown as follows:

```

else if (s.startsWith("\\begin{schema}")){

```



```

generic.tempS.add(s);
ss = s.substring(s.lastIndexOf("{")+1, s.lastIndexOf("}"));
ss = ss.trim();
generic.schName.add(ss);
while (!(s = br.readLine().trim()).equals("\\ST") &&
!(s.equals("\\end{schema}"))){
    // variables processed here
}
if (s.equals("\\ST")){
    // predicates processed here
}
else generic.tempS.add(s);
}
else generic.tempS.add(s);

```

A schema can contain varieties of declarations: an inclusion, a primed, a " Δ ", and a " Ξ " of state schemas, and other variables. All of these are handled by code as follows:

```

if (!s.contains(generic.schName.get(0)+"\'') &&
!s.contains("\\Delta_" + generic.schName.get(0)) &&
!s.contains("\\Xi_" + generic.schName.get(0)) &&
!s.contains(generic.schName.get(0).toString())){
    // see The First Detail
}
else{
    if (s.contains(generic.schName.get(0).toString()) &&
!s.contains(generic.schName.get(0)+"\'') &&
!s.contains("\\Delta_" + generic.schName.get(0)) &&
!s.contains("\\Xi_" + generic.schName.get(0))){
        ...
    }
    else if (s.contains(generic.schName.get(0)+"\'') &&
!s.contains(generic.schName.get(0).toString()) &&
!s.contains("\\Delta_" + generic.schName.get(0))
&& !s.contains("\\Xi_" + generic.schName.get(0))){
        ...
    }
    else if (s.contains("\\Delta_" + generic.schName.get(0)) ||
s.contains("\\Xi_" + generic.schName.get(0)) &&
!s.contains(generic.schName.get(0)+"\'')){
        ...
    }
}
generic.tempS.add(s);

```

The first conditional is to process an ordinary variable, whereas other conditional if blocks process a name of a state schema which can either be added with a decoration or not.

The First Detail is shown as follows:

```

if (s.indexOf(';') > -1){
    // to process a line of variables which contains ";"s
}
else{
    // otherwise
}

```

Since a ";" can occur many times in a line of declared variable, inside "... " was specified a do-while block which will process variables as long as a ";" is still found. After there is no ";", a variable can be stored directly.

Since several variables can have a same type, this case was also specified. It denotes ";"s are available in a group of the same type of variables.

These variables are put into two different storages: a storage for state variables which is `hmStVar`, and a storage for operational schemas variables which is `hmVar`. Both of them are instances of `HashMap`.

"..."s are to process a name of a state schema with or without a decoration. The state schema variables will be read here and they will be added to a list of variables of an associated operational schema. These "..." are not displayed here since they do not really relate to a spotting of generic constants usages.

Let us now discuss a process on a predicate part of a schema in which such a usage usually occurs.

```

generic.tempS.add(s);
s = br.readLine().trim();
s = "...";
while (!s.equals("\\end{schema}")){
    checkModFunc = generic.tempS.size();
    // joins several lines which are ended either with a ".dot" or a ".cbar"
    if (!ss.isEmpty()){
        generic.getActualParam(ss, vcStr, genConstant, false);
        // irrelevant code
        if (!s.equals("\\end{schema}")) s = br.readLine().trim();
    }
    else{
        generic.getActualParam(s, vcStr, genConstant, false);
        // irrelevant code
        if (!s.equals("\\end{schema}")) s = br.readLine().trim();
    }
}
generic.tempS.add(s);

```

The joining will stop until the line is ended with a "\\ " or `\end{schema}`. This process is quite similar to the process in the earlier discussion.

As can be seen from the above code, `getActualParam` function is called in two different places and both calls have "false" for their last parameter. As mentioned in the above discussion, this function will identify types of actual parameter on usages of generic constants. The last parameter indicates that the call is not conducted on a predicate part of a generic constant definition. Because of this value, it is possible that this call will be processed further until the types are obtained.

There are several writings of these usages that necessary to be introduced. They were specified in our system, as follows:

- A generic constant is followed by a pair of brackets which encloses its parameters. This usage will call `getVariableBracket` function.

This function was specified in `generic.java` and it was called by `getActualParam` function.

The first process in `getVariableBracket` function is to get a type of a generic constant. This function will call another helping function specified also in `generic.java` which is `splitType` and it is shown as follows:

```
temp = Parser.hmGenAxDef.get(gcVar).toString();
if (temp.contains(",")){
    idx = 0;
    do{
        temp1 = temp.substring(idx,temp.indexOf(", ",idx)).trim();
        ingen.tempType.add(temp1);
        idx = temp.indexOf(", ", idx) + 1;
        if (temp.indexOf(", ", idx) == -1){
            ingen.tempType.add(temp.substring(idx).trim());
            break;
        }
    }
    while(idx < temp.length());
}
```

The above code is a process for a generic constant which its occurrences are higher than one on associated `Z` specification. It means that the type of this generic constant is not a simple type. For example, a type of `(X \pfun Y) \fun \pset X` will be split into `X`, `\pfun`, `Y`, `\fun`, `\pset`, `X`. Each type will be stored on the same generic constant which is separated by a `,` from each type. As can be seen from the above code, this function gets the type from the parser based on the associated generic constant. The associated grammar is shown as follows:

```
basic.decl: schema.ref
{
    ...
}
| WORD " " ' : ' expr.word
{
    String str;
    if (genAxDef){
        $$ = $1+" ";
        llGenAxDefVar.add($$);
        for (int i=0;i<llGenAxDefTypes.size();i++){
            if (hmGenAxDef.containsKey(
                llGenAxDefVar.getLast().toString())){
                str = hmGenAxDef.get(
                    llGenAxDefVar.getLast().toString()).toString();
                hmGenAxDef.put(llGenAxDefVar.getLast().
                    toString(), str + ","+
                    llGenAxDefTypes.get(i).toString());
            }
            else hmGenAxDef.put(llGenAxDefVar.getLast().
                toString(),llGenAxDefTypes.get(i).toString());
        }
    }
    ...
}
```

```

        else if (schDef){
            ...
        }
        else $$ = $1+"'" + " " + $3 +
            " " + $4;
        ...
    }
    | word.decl.name.list ':' expr.word
    ...
;

```

“...” can be seen in Appendix F on page 305.

If a type does not contain any “,”, this type is stored directly to `ingen.tempType`. The next process, which is applied to either the type with “,” or without “,”, is to read contents of `ingen.tempType`. If there are open brackets without any close bracket on every element of that list, the open brackets will be deleted. If there are close brackets without any open bracket on every element of that list, the close brackets will be deleted. Afterwards, if an element contains a generic parameter, this element will be stored in `gc.genParam`.

After a type of generic constant is obtained, the next process is to obtain an actual type for each parameter of this generic constant. This actual type will be matched with the generic parameter on the same order for every actual type. To do this, another function is called, which is `unifyTypeBrack`. This function adopts a simple unification process as follows:

```

tyVar = "";
for (int i=0;i<strTy.length();i++){
    if (i< tyTT.length()-1 && strTy.charAt(i) != tyTT.charAt(i)){
        if (strTy.charAt(i) == 'X' || strTy.charAt(i) == 'Y' ||
            strTy.charAt(i) == 'Z'){
            tyVar = tyVar + tyTT.substring(i).trim();
            tyVar = tyVar.trim();
        }
        else if (strTy.charAt(i) == '_'){
            tyVar = tyVar + tyTT.charAt(i);
        }
        else if (tyTT.charAt(i) == '(' || tyTT.charAt(i) == ')'){
            strTy = strTy.substring(0, i) + tyTT.charAt(i) +
                strTy.substring(i);
        }
    }
}
}

```

`getVariableBracket` calls also another helping `indexOfById` function. The latter function performs a forward search on a list based on an index input shown as follows. This search will find the index of the first occurrence, which is started with an input index, of an object.

```

static int indexOfById(List list, Object searchedObject, int idx){
for (int i = idx; i < list.size(); i++){
    if (list.get(i).toString().equals(searchedObject.toString()))
        return i;
}
return -1;
}

```

A List in Java can perform a forward search, but it is always started with the first index on the list, which is 0. An opposite function, which is to get the index of the last occurrence of an object and this search is started with an input index, was also created. It is shown as follows:

```

static int lastIndexById(List list, Object searchedObject, int idx){
for (int i = idx; i >= 0; i--){
    if (list.get(i).toString().equals(searchedObject.toString()))
        return i;
}
return -1;
}

```

- A generic constant is followed by explicit types of its parameters which are enclosed with a pair of square brackets. This usage will call `getVariableSquare` or `getConstType` function. The same as `getVariableBracket` function, processes on `getVariableSquare` function begins also with the same of `splitType` function. `getVariableSquare` function performs almost the same process as `getVariableBracket` function generally. However, this function calls a different function for unification which is `unifyTypeSqBrack`. All of above processes apply also to `getConstType`, but it calls `unifyType` for the unification. The unification is generally the same as the one discussed on above item.
- A generic constant is written in a membership operator or an equality. This usage will call `getTypes` or `getConstType` function. Generally, `getTypes` function performs almost the same processes as other functions. It calls also `unifyTypeBrack` function for the unification.

There were three usages on generic constant `headSeq`. The first usage has an actual type, `seq1NAME`, which was specified as an explicit type. Unification as shown in the figure uses three variables: `strTy` represents a generic type, `tyTT` represents an actual type, and `tyVar` holds a value for the generic literals. After a unification, the only one generic parameter `X` will be actualised to `NAME`. The second usage made `X` be actualised to "N", whereas the last usage made `X` be actualised to "Z".

Thus, after usages of generic constants are spotted and processed on every schema on a `Z` specification, generic parameters are changed along with the

spotted usages. If the number of usages of a generic constant higher than one and they have different types of parameters, all these types of parameters will be presented in a different copy of axiomatic definitions, which will be differentiated by adding an index following the name of the generic constant.

Our system declared a variable, `isFirst`, to indicate whether a generic constant with its actual parameters is the first occurrence on associated Z specification. Its initial value is true and is changed into false if there is a reading of such a redefined generic constant.

The case in which `isFirst` is false can be differentiated into two further cases. The first case is the generic constant has different type of its actual parameters comparing with the same name of generic constant. In this case, a new copied of the same name of a generic constant, but with different types of actual parameters, is processed. Further steps for this case is given earlier in above paragraph.

The second case is, the same name of generic constant and also the same types of actual parameters. In this case, there is no new copy of such a generic constant.

All of these three usages were conducted on the same generic constant name, but with different types of actual parameter. It is represented as "first", for the first usage, and "different" for each of the second and third usage.

In order to be able to see outputs of the above process, a statement to print contents of `tempS` was added. As can be seen from both figures, callers for this generic constant name have not been modified to reflect their indexes if any.

After all these processes, the next process is to modify the caller of these usages. In some occasions, there are indices added to these generic constants. In others, any square brackets specifying actual types of parameters explicitly are deleted. It is since Z2SAL does not support explicit types.

A usage of a generic constant can be found either in a generic constant definition, an axiomatic definition or a schema. The above discussion is only to identify usages of schemas, but identifying usages of a generic constant, and an axiomatic definition is generally the same and simpler than the one on schemas. However, nothing of Z specifications in our experiments has a usage of a generic constant in a generic constant definition box. Nonetheless, several of our examples have such usages, but the first usage was specified in schemas. The usage is indicated by calling the name of the generic constant and passing the actual parameters to replace the generic ones.

5.4 Important Findings around a Redefinition of Generic Constants

Our system can also be used to redefine other generic forms in a Z specification such as an abbreviation definition and a lambda expression. Z2SAL supports an abbreviation definition, but not the generic one. A user of Z specifications is common to declare global constants by using abbreviation definitions in their specification. By defining this abbreviation as generic, the definition will have a generic parameter that can have different types in its usages.

Firstly this generic abbreviation definition was redefined to a usual abbreviation one. However, a generic abbreviation definition is usually defined using a set comprehension; Z2SAL does not support this kind of abbreviation definition. Thus, another finding is the current Z2SAL supports an abbreviation definition, but not a one that has a set comprehension as its value. Since then, a generic abbreviation definition is redefined to a generic constant one instead.

Other finding in encountering the set comprehension in a generic abbreviation definition is a difference between Z and SAL language in defining such a set. The Z notation supports many parameters declared in a set comprehension, but neither of Z2SAL nor the SAL language supports such a set. Based on the SAL literature, a user can only declare one parameter in a set comprehension definition. The SAL syntax [18] for a set expression is given as follows:

$$\begin{aligned} \textit{SetExpression} &:= \textit{SetListExpression} \mid \textit{SetPredExpression} \\ \textit{SetListExpression} &:= \{\{ \textit{Expression} \}^+\} \\ \textit{SetPredExpression} &:= \{ \textit{Identifier} : \textit{Type} = \textit{Expression} \} \end{aligned}$$

This fact makes us redefine our generic abbreviation definition as well as our lambda expression to an axiomatic definition that does not use a set comprehension any more.

A lambda expression is used to define a function without a name [2]. Z2SAL does not support this expression, which inevitably it is commonly used in a generic constant definition. Our approach is to redefine the lambda expression to an axiomatic definition since both these definitions have similar behaviours. As it is known that an axiomatic definition can be used to define a function as well as a relation or a constant.

Several of our Z specifications from our experiments consist of abbreviation generic definitions and lambda expressions. The associated examples are `fMaxComSubSeq_orig.tex`, `fMaxComSubSeq.tex`, `fMaxComSubSeq_1.tex`, `fMonoSeq_1.tex`, and `fMonoSeq.tex`. These can be read on the later chapter.

The above discussion was performed on these specifications both manually and automatically.

During the development of the redefinition system, a new translation for a function, which is defined by a user in an axiomatic definition, was proposed. It can also be applied to a constant. This approach is another type of our support for model checking Z specifications and it will be discussed further in the next section.

5.5 A Proposed Translation of SAL Function

Based on our experiments, the current translation of several functions or constants by Z2SAL failed to be verified or simulated by the SAL tool. An unsupported error of either a failure to convert function application or an incompatible type in the equality operator was produced either by the SAL model checker or the SAL simulator during an execution of a SAL file. The SAL file was generated by Z2SAL from the Z input specification generated by our redefinition system. Thus, it is an issue on working with the redefinition system and this issue was decided to be discussed at the end of this chapter. This finding motivated us to propose a new SAL translation for these functions or constants.

This new translation was based on the SAL literature [18]. Fortunately, this translation could be verified or simulated successfully by the SAL tools as given by our experiments. However, such a translation cannot be applied to a relation since a relation does not have a type for its output parameter.

The current Z2SAL translates a function, a relation and a constant in the base module, in which Z2SAL defines `State` as a default name for a module, and puts variable declarations inside a definition clause. On the other hand, a user defined function, relation and constant are always declared outside the module and put inside a context clause, specifically in a constant declaration.

A constant declaration has a syntax as follows [18]:

$$\textit{ConstantDecl} := \textit{Identifier}[(\textit{VarDecls})] : \textit{Type}[= \textit{Expr}]$$

Thus, the generated SAL was modified to adapt a constant declaration formulated by SAL.

An example has been given in the previous section, specifically in Section 2.5.2 on page 67. However, let us look at back that example.

```

output_fSwap : CONTEXT = BEGIN
...
State : MODULE =
  BEGIN
    LOCAL swap1 : [NAME_X_NAME -> B_NAME_X_B_NAME]

```



```

LOCAL swap2 : [NAME_X_NAT -> B_NAT_X_B_NAME]
...
DEFINITION
  invariant_ = (
    function {NAME_X_NAME, B_NAME_X_B_NAME;
      (NAME_BB, NAME_BB)} ! total?(swap1) AND
    function {NAME_X_NAT, B_NAT_X_B_NAME;
      (4, NAME_BB)} ! total?(swap2) AND
    (FORALL (q--1 : NAME, q--2 : NAME) : swap1((q--1, q--2))
      = (q--2, q--1)) AND
    (FORALL (q--3 : NAME, q--4 : NAT) : swap2((q--3, q--4))
      = (q--4, q--3)))
INITIALIZATION [
  ...
  ->
]
TRANSITION [
  ...
  []
  ELSE ->
    name' = name
]
END;
END

```

The above code is a SAL file generated by Z2SAL from an associated Z specification. "... " hides several lines that are not necessary here.

As discussed earlier, this SAL file has a problem with the SAL tools. In other words, the SAL tool failed to execute this SAL file. Afterwards, it was modified manually and it was changed as follows:

```

output_fSwap_mod : CONTEXT = BEGIN
% the same as the ones in the previous SAL file
swap1(q--1 : NAME, q--2 : NAME): B_NAME_X_B_NAME = (q--2, q--1);

swap2(q--3 : NAME, q--4 : NAT): B_NAT_X_B_NAME = (q--4, q--3);

State : MODULE =
  BEGIN
    % the same as the ones in the previous SAL file
    INITIALIZATION [
      % the same as the ones in the previous SAL file
      ->
    ]
    TRANSITION [
      % the same as the ones in the previous SAL file
      []
      ELSE ->
        name' = name
    ]
  END;
  th1: theorem State |- G(FORALL (i,j: NAME): swap1(i,j) = (j,i));
END

```

This modified SAL file could be executed successfully by the SAL tool as discussed on the earlier section (see Section 2.5.2 on page 67).

To compare both SAL files, there are several keys to consider as follows:

- Those functions are declared on the different place. The above functions were declared by Z2SAL as a `LOCAL` variable inside a state clause `MODULE`, please see the first SAL file above. On the other hand, the same functions were declared by modifying manually the first SAL file as shown above in the `output_fSwap_mod` context clause, please see the second SAL file above.
- They have also different method of declaration. The modified version is quite similar with the declaration of function generally. A function declaration usually starts with a function name, and a list of parameters and their types which is enclosed with a pair of brackets. A type of the output of this function can be put before the function name or after that list of parameters. To compare these declaration, the below code is a declaration which is obtained from the modified version:

```
swap1(q--1 : NAME, q--2 : NAME): B_NAME_X_B_NAME
```

`swap1` is the function name. There were two parameters: `q_1`, and `q_2`; both of them have the same type `NAME`. The type of the output is `B_NAME_X_B_NAME`.

On the other hand, the generated SAL file has a declaration as follows:

```
LOCAL swap1: [NAME_X_NAME -> B_NAME_X_B_NAME]
```

- The declaration and body of the modified function are also quite simpler than the original function. The function in the modified SAL file only has the below code in its body:

```
= (q--2, q--1);
```

However, the same function in the original SAL file has tremendous code as follows:

```
function {NAME_X_NAME, B_NAME_X_B_NAME; (NAME_BB,NAME_BB)} !
total?(swap1) AND (FORALL (q--1 : NAME, q--2 : NAME) :
swap1((q--1, q--2)) = (q--2, q--1)) AND
```

5.6 Conclusion

This system is expected to help Z2SAL in translating generic constant definition. Rather to translate directly a Z specification into a SAL file as Z2SAL does, this system redefines a generic constant definition to an equivalent axiomatic definition.

Although our system covers a small amount of Z generic constant definitions, it can generate a redefined Z specification which can be translated by Z2SAL easily and executed by the SAL tool. It also shows that our redefinition system works. This success is expected as our contribution to related research in this field. The redefinition system is a part of the architecture of our research as shown by Fig. 3.1 on page 77.

A redefinition of generic constants relates also to a new translation of a SAL function and constant. Based on our experiments on this translation, it is argued that this translation is suitable to consider. However, this new translation has not been applied to function which has more complex body. Nonetheless, it can be set as a future work. It also shows that our proposal on translation of user-defined functions and constants works. This success is expected as our contribution to related research in this field. The proposal is also a part of the architecture of our research as shown by Fig. 3.1 on page 77.

More experiments on this system and an evaluation of this system can be seen in later chapter. The next chapter discusses our system to expand schema calculus.

Chapter 6

Expanding Schema Calculus

This section discusses a design and implementation of a schema calculus expansion system. It begins with an introduction to schema calculus.

6.1 Introduction

By using a schema calculus definition, a new schema can be obtained by combining previously specified schemas. Thus, schemas that have been specified can be reused to specify a new schema. Every schema has its operations in a specification, named a 'schema separation' [56].

Since every schema operator has its own definition, a schema operator affects how an expansion is performed. The expansion means that all unique variables of the involved schemas are listed in a new schema. It also means that predicates which are read from the involved schemas are added. These predicates are combined using schema operators.

There is a prerequisite for operating two schemas; common variables should have the same type. Furthermore, in a case of the negation operator, normalisation is also required.

Normalisation defines explicit constraints given by a declaration part of a related schema, which is performed just before negation, and specifies these constraints in a predicate part of the schema.

6.2 An Implementation of the Expansion System

In this section, the implementation of our expansion system is discussed. The expansion system was included in the tool as support for model checking Z

specifications, as well as the redefinition system. This system can expand a small subset of the Z schema calculus.

A list of schema operators which can be expanded by our system have been discussed in the previous section (see Section 6.1 on page 130). Although this list covers almost all schema operators in the Z schema calculus, a user must not specify schema calculus in a complex definition. For more description about whether a schema calculus definition can be expanded or not by our system, please read through the implementation of this system as discussed below, and experiments on this system in a later chapter.

Our method in this expansion system is to construct a new schema by expanding other schemas, in which they are connected by schema operators. This system supports several Z schema operators, which have been discussed on earlier section, such as: " \neg ", " \wedge ", " \vee ", " \Rightarrow ", " \Leftrightarrow ", " $/$ ", " \circ ", " \setminus ", " \uparrow ", " \exists ", " \forall ", and " \oplus ". However, all these schema operators should be specified after " $\hat{=}$ " and their paragraphs do not start with "[" and end with "]"; it is one of limitations of our system.

The following subsection describes how our system processes a schema calculus definition. This description is accompanied by associated code and an example of a Z specification on this code. A Z specification which was used for this purpose was taken from our experiments, `expandingschema_2.tex`. This specification and its expanded schema generated by our system can be seen in Appendix H on page 328. Since this example contains only one schema calculus definition which uses conjunction operator, several other operators will be also accompanied by an incomplete example. The example is incomplete as it does not start from the first process until the final one. The purpose is just to show how such an operator works.

6.2.1 Expansion Processes in Java Main Program

Our system starts an expansion process after a parsing on a Z specification is successful. The expansion process is then performed by executing the `expand` function. This function was specified in our main program, `Z_Preprocessing_Tool.java`.

Next, an expansion starts by a reading of a `\begin{zed}`, " $\hat{=}$ ", and one of schema operators, as shown below:

```

else if (s.startsWith("\\begin{zed}")){
    s = br.readLine();
    while (!(s.equals("\\end{zed}"))){
        if (s.contains("\\sdef")){
            if (s.contains("\\znot") || s.contains("\\zand") || s.contains("\\zor") ||
                s.contains("\\zimp") || s.contains("\\zeq") || s.contains("\\zfor") ||
                s.contains("\\zcmp") || s.contains("\\semi") || s.contains("\\zhide") ||
                s.contains("\\hide") || s.contains("\\zproject") || s.contains("\\project"))||

```

```
s.contains("\\zexi") || s.contains("\\zall") || s.contains("\\zovr")){
```

Our system recognizes also other zed definition boxes. All these definitions are not specified by a "≐" after a `\begin{zed}`. Thus, these definitions are not a schema calculus definition. They are specified for declaring an abbreviation definition, and a basic type definition.

A function to process schema calculus is called afterwards as given in the below code. This call will return a boolean value: true if a schema expansion can be performed, false otherwise. This function is defined in another Java program, `schCal`. All processes on expansion are performed on the latter Java program. These processes will be discussed on a separate sub-section afterwards. On the other hand, the process of the `expand` function, which was specified in the main program, will be discussed in this sub-section.

```
strExp = s;
ss = s.substring(0, s.indexOf("\\sdef"));
ss = ss.trim();
schCal.schemaBox[idx] = new schema(ss);
if (countSch > 0){
    schCal.tempS.add("_");
}
schCal.tempS.add("\\begin{schema}{"+ss+"}");
expanded=schCal.schConstruction(s.substring(s.indexOf("\\sdef")+5).trim(), idx);
if (expanded){
    idx++;
}
}
```

Thus, every time a `\begin{zed}`, "≐" and one of schema operators are read, a new schema is created. An index, `idx`, is used to number the new schema.

Our expansion system is assisted by our parser to obtain schema names and their operators. This process has been shown in Sub-section 4.2.3 on page 100. The `word.schema.exp1` non-terminal, which was used on that process and was not provided on that sub-section, is given as follows:

```
word.schema.exp1: schema.exp1
| WORD
  {
    llSchCal.add($1);
  }
;
```

For other non-terminals mentioned on `schema.exp1`, they can be seen in Appendix F on page 305. Contents of the `llSchCal` list will be read and used for schema expansion process.

Let us now move to another Java program, `schCal` to continue the expansion process. The first function to discuss is `schConstruction`.

6.2.2 Expansion Processes in schConstruction function

This function requires two parameters to operate: a string `s` represents a schema calculus definition and an integer number `index` represents an index number of a new schema.

The first process in this function is to check existences of redundant brackets. Redundant brackets mean that there is no schema operator which is enclosed with a pair of brackets. It is also redundant brackets if only a negation operator is specified inside a pair of brackets. Redundant brackets will be removed from a list of operators and arguments of a paragraph of schema calculus.

It is shown as follows:

```
while (getI < Parser.llSchCal.size()){
if (Parser.llSchCal.get(getI).toString().equals("(") &&
Parser.llSchCal.get(getI+1).toString().equals(")")){
if (!Parser.llSchCal.get(getI-1).toString().startsWith("\\\n")){
Parser.llSchCal.remove(getI+1);
Parser.llSchCal.remove(getI);
}
else if (Parser.llSchCal.get(getI-1).toString().startsWith("\\\nznnot")){
// same as previous block
}
else getI++;
}
}
```

Our example, `expanding_schema_2`, does not have any redundant brackets.

On the same while loop, there are other checks to pre-process contents of the `llSchCal` list before an expansion begins. The second check, which is an else clause of the above if block, is applied to schema decorations which are " Δ " and " Ξ ". The aim is to join the decoration with its successor and remove the successor from the list. It is shown in the below code.

```
else if (Parser.llSchCal.get(getI).toString().equals("\\\nDelta")){
Parser.llSchCal.set(getI, Parser.llSchCal.get(getI).toString() +
Parser.llSchCal.get(getI+1).toString());
Parser.llSchCal.remove(getI+1);
}
else if (Parser.llSchCal.get(getI).toString().equals("\\\nXi")){
// the same code as the above block
}
```

The third check is on hiding operator. The aim is to join all variables and the join stop until the close bracket.

```
else if (Parser.llSchCal.get(getI).toString().contains("\\\nzhide")){
getI--;
tempSch = "";
do {
tempSch = tempSch + "_" + Parser.llSchCal.get(getI).toString();
Parser.llSchCal.remove(getI);
if (Parser.llSchCal.get(getI).toString().equals(")")){
tempSch = tempSch + "\n";
}
}
}
```

```

    Parser.llSchCal.remove(getI);
    break;
  }
}
while (!Parser.llSchCal.get(getI).toString().equals("")){
  Parser.llSchCal.add(tempSch);
  getI++;
}

```

The last check on this while loop is applied to quantifiers. The aim is generally similar with two previous checks. This time it is to join variables until a "•" is read. If none of all above checks is relevant, the `getI` index is updated. Our system did not perform any of above checks on our *Z* specification example. It denotes that this example does not contain both decorations, a hiding operator, and a quantifier in its schema calculus definition.

In a separated part, but still in the same function, there is another check to find out an existence of an implication operator, " \Rightarrow ". If it is found, contents of the `llSchCal` list will be copied to a temporary list, `llTempImpl`. This temporary list is required to obtain an original sequence of schemas and operators before an implication is performed. It is shown as follows:

```

getI = 0;
if (Parser.llSchCal.contains("\\zimp")){
  while (getI < Parser.llSchCal.size()){
    llTempImpl.add(Parser.llSchCal.get(getI).toString());
    getI++;
  }
}

```

Since there is no implication operator on the only one schema calculus definition on our *Z* specification example, the above check was not performed either.

Another check is applied to horizontal schemas.

```

getI = 0;
while (Parser.llSchCal.size() > 0 && getI < Parser.llSchCal.size()){
  if (Parser.llSchCal.get(getI).toString().trim().equals("\\lsch")){
    if (Parser.llSchCal.indexOf("separator") < 0 ||
        (Parser.llSchCal.indexOf("separator") > 0 &&
         Parser.llSchCal.indexOf("separator") > Parser.llSchCal.indexOf("\\rsch"))){
      m = getI + 1;
      temp = Parser.llSchCal.get(m).toString().trim();
      if (temp.contains("|")){
        temp1 = temp.substring(0, temp.indexOf("|")).trim();
        getVarVal(temp1, schemaBox[index]);
        temp1 = temp.substring(temp.indexOf("|")+1).trim();
        schemaBox[index].llPred.add(temp1);
      }
      else if (temp.contains("\\bbar")){
        // the same code as previous block,
        // but with appropriate string to match and length
      }
      else if (temp.contains("\\zbar")){
        // the same code as previous block,
        // but with appropriate string to match and length
      }
    }
  }
  getI++;
}

```



```

    }
    else {
        getVarVal(temp, schemaBox[index]);
    }
    Parser.llSchCal.remove(m+1);
    Parser.llSchCal.remove(m);
    Parser.llSchCal.remove(m-1);
    Parser.llSchCal.add(getI,
        schemaBox[index].nameSch.trim());
}
else getI++;
}
else getI++;
}

```

Every if block calls the `getVarVal` function to assign variables to a new schema, as well as to assign predicate to the schema. The schema calculus on our Z specification example was not specified in a horizontal schema. Thus, the above function was not performed either.

The expansion process will start afterwards. Its code is as shown below:

```

while (Parser.llSchCal.size() > 0 && getI < Parser.llSchCal.size()){
    if (Parser.llSchCal.get(getI).toString().startsWith("\\") &&
        !Parser.llSchCal.get(getI).toString().equals("\\znot") &&
        !Parser.llSchCal.get(getI).toString().equals("\\Delta") &&
        !Parser.llSchCal.get(getI).toString().equals("\\Xi") &&
        !Parser.llSchCal.get(getI).toString().equals("\\zimp") &&
        !Parser.llSchCal.get(getI).toString().equals("\\zeq") &&
        !Parser.llSchCal.get(getI).toString().startsWith("\\zall") &&
        !Parser.llSchCal.get(getI).toString().startsWith("\\zexi")){

```

Thus, the above code will process schema operators " \wedge " and " \vee ".

There are several cases on how a schema calculus definition, which uses both operators, was specified. Each case will be discussed on following subsection.

An Operator with Two Operands

The first case is an operator which has two operands. It is as shown below:

```

if (getI-2 >= 0 && !Parser.llSchCal.get(getI-2).toString().equals("(") &&
    !Parser.llSchCal.get(getI-2).toString().equals(")") &&
    !Parser.llSchCal.get(getI-2).toString().startsWith("\\")){

```

On this case, distributive simplifications are performed automatically on predicates. These simplifications are as follows:

The First Simplification If a paragraph is on a form of **(A op1 B op2 (A op1 C))**, this paragraph will be simplified into the form of **A op1 (B op2 C)**. It is shown as follows:

```

if (llBracket.get(i).toString().contains(Parser.llSchCal.get(getI-2).
toString() + " " + Parser.llSchCal.get(getI).toString())){

```

```

// irrelevant code
tempSch = llBracket.get(i).toString().substring(llBracket.get(i).toString().
indexOf(Parser.llSchCal.get(getI).toString()) +
Parser.llSchCal.get(getI).toString().length()).trim();
getIdx = i;
llBracket.set(getIdx, Parser.llSchCal.get(getI-2).toString() +
"_" + Parser.llSchCal.get(getI).toString());
if (getIdx > 0 && llBracket.get(getIdx-1).toString().equals("(") &&
llBracket.get(getIdx+1).toString().equals(" ")){
    llBracket.set(getIdx+1, "(");
    llBracket.remove(getIdx-1);
    hmOperator.put(getIdx-1, "back");
    // of irrelevant code
}
llBracket.add(tempSch + "-" + Parser.llSchCal.get(getI-1).toString());
hmOperator.put(hmOperator.size(), "complete");
llBracket.add(")");
break;
}

```

The Second Simplification If it is on a form of **(B op1 A) op2 (C op1 A)**, the simplified form is **(B op2 C) op1 A**. This simplification is shown as follows:

```

else if (llBracket.get(i).toString().contains(
Parser.llSchCal.get(getI).toString() + "_" +
Parser.llSchCal.get(getI-1).toString())){
    // irrelevant code
    tempSch = llBracket.get(i).toString().substring(0, llBracket.get(i).
toString().indexOf(Parser.llSchCal.get(getI).toString()).trim());
    // irrelevant code
    llBracket.set(getIdx, tempSch + "-" + Parser.llSchCal.get(getI-2).toString());
    llBracket.add(Parser.llSchCal.get(getI).toString()
+ "_" + Parser.llSchCal.get(getI-1).toString());
    hmOperator.put(hmOperator.size(), "front");
    break;
}

```

Both the above simplifications are specified in a if block as follows:

```

if (llBracket.toString().contains(Parser.llSchCal.get(getI-2).toString() +
"_" + Parser.llSchCal.get(getI).toString()) ||
llBracket.toString().contains(Parser.llSchCal.get(getI).toString()
+ "_" + Parser.llSchCal.get(getI-1).toString())){

```

Sometimes, a pattern of schema calculus is not as regular as previous discussions. For this kind of schema calculus, our system specifies it as follows:

```

else if (llBracket.toString().contains(Parser.llSchCal.get(getI-1).toString()
+ "_" + Parser.llSchCal.get(getI).toString()) ||
llBracket.toString().contains(Parser.llSchCal.get(getI).
toString() + "_" + Parser.llSchCal.get(getI-2).toString())){

```

These patterns are grouped as follows:

The First Not Regular Simplification A not regular form such as **(B op1 A) op2 (A op1 C)** will be simplified into the form of **(B op2 C) op1 A**, the same as the second rule above. It is the first form of not regular patterns.

```

if (llBracket.get(i).toString().contains(Parser.llSchCal.get(getI-1).
toString() + " " + Parser.llSchCal.get(getI).toString())){
    // irrelevant code
    tempSch = llBracket.get(i).toString().substring(llBracket.get(i).
toString().indexOf(Parser.llSchCal.get(getI).toString())
+ Parser.llSchCal.get(getI).toString().length()).trim();
    getIdx = i;
    llBracket.set(getIdx, Parser.llSchCal.get(getI-1).toString()+
" " + Parser.llSchCal.get(getI).toString());
    if (getIdx > 0 && llBracket.get(getIdx-1).toString().equals("(") &&
llBracket.get(getIdx+1).toString().equals(" ")){
        llBracket.set(getIdx+1, "(");
        llBracket.remove(getIdx-1);
        hmOperator.put(getIdx-1, "back");
        // irrelevant code
    }
    llBracket.add(Parser.llSchCal.get(getI-2).toString() + "-" + tempSch);
    hmOperator.put(hmOperator.size(), "complete");
    llBracket.add(")");
    break;
}
}

```

The Second Not Regular Simplification Another not regular form such as **(A op1 B) op2 (C op1 A)** will be have the same simplified form as the first rule on regular patterns above. This is shown as follows as the second form of not regular patterns.

```

else if (llBracket.get(i).toString().contains(Parser.llSchCal.get(getI).
toString() + "⌊" + Parser.llSchCal.get(getI-2).toString())){
    // irrelevant code
    tempSch = llBracket.get(i).toString().substring(0, llBracket.get(i).
toString().indexOf(Parser.llSchCal.get(getI).toString()).trim());
    // irrelevant code
    llBracket.set(getIdx, tempSch + "-" + Parser.llSchCal.get(getI-1).toString());
    llBracket.add(Parser.llSchCal.get(getI).toString()
+ "⌊" + Parser.llSchCal.get(getI-2).toString());
    hmOperator.put(hmOperator.size(), "front");
    break;
}
}

```

Other than both forms of simplification, there is a form of schema calculus which does not form any pattern. For this form, an operation is just to store the schema calculus definition in the `llBracket` list in the form of an infix operation. It is shown by the code as follows:

```

else{
    llBracket.add(Parser.llSchCal.get(getI-2).toString() + "⌊" + Parser.llSchCal.
get(getI).toString() + "⌊" + Parser.llSchCal.get(getI-1).toString());
    hmOperator.put(hmOperator.size(), "complete");
}
}

```

Our example fell in this category.

After an operation of associated three forms above, the next process is to process other schemas and their operator including brackets. However, it will be discussed later after patterns on conjunction and disjunction operator are finished. The next discussion is the second case of these patterns.

An Operator with a Right Operand

The second case is a schema calculus definition whose left operand has been processed previously. It is shown as follows:

```
else if (getI-2 < 0 && getI > 0 && Parser.llSchCal.size() > 1 && isFront){
  llBracket.add(Parser.llSchCal.get(getI).toString()
+ "┌" + Parser.llSchCal.get(getI-1).toString());
  hmOperator.put(hmOperator.size(),"front");
  // irrelevant code
  if (getI > 0){
    isBack = true;
    isFront = false;
  }
  else {
    isBack = false;
    isFront = true;
  }
  if (isBack){
    llBracket.add(llBracket.size()-1, "modified_later");
    hmOperator.put(hmOperator.size(), hmOperator.get(
hmOperator.size()-1).toString());
    hmOperator.put(hmOperator.size()-2, "back");
  }
  if (Parser.llSchCal.size() > 1 && Parser.llSchCal.get(getI).toString().
equals("(") && Parser.llSchCal.get(getI+1).toString().equals("))"){
    // irrelevant code
  }
}
```

This case is necessary since it is often to have schema calculus which contains many schemas and operators. However, our system can only process a medium number of schemas involved in schema calculus. Several examples on our experiments show this case.

An Operator with a Left Operand

The third case is the opposite of previous case; the right operand has been processed previously. It is shown as follows:

```
else if (getI-2 < 0 && getI > 0 && Parser.llSchCal.size() > 1 && isBack){
  llBracket.set(llBracket.indexOf("modified_later"), Parser.llSchCal.
get(getI-1).toString() + "┌" + Parser.llSchCal.get(getI).toString());
  // irrelevant code
  if (getI > 0){
    isBack = true;
    isFront = false;
  }
}
```

```

else {
    isBack = false;
    isFront = true;
}
if (isBack){
    llBracket.add(llBracket.size()-1, "modified_later");
    hmOperator.put(hmOperator.size(),
    hmOperator.get(hmOperator.size()-1).toString());
    hmOperator.put(hmOperator.size()-2, "back");
}
if (Parser.llSchCal.size() > 1 && Parser.llSchCal.get(getI).toString().
equals("(") && Parser.llSchCal.get(getI+1).toString().equals(" ")){
    // irrelevant code
}
}
}

```

An Only Operator

The last case here is just an operator which is available. Its operands have been processed previously. This case has three variants. The first one is given as follows:

```

if (getIdx > -1 && oldIdx != getIdx){
    llBracket.set(llBracket.size()-2, llBracket.get(llBracket.size()-2).
toString().substring(0, llBracket.get(llBracket.size()-2).toString().
indexOf("-")) + " " + Parser.llSchCal.get(getI).toString()+ " " +
llBracket.get(llBracket.size()-2).toString().substring(llBracket.
get(llBracket.size()-2).toString().indexOf("-")+1).trim());
    getIdx = -1;
    oldIdx = -1;
}
}

```

The operator will replace the dash.

The second variant is generally similar to the above case. The difference is on the position of the `llBracket` list that will be modified. In this variant, it is the `getIdx` of `llBracket` that is processed.

The last variant is just the operator that will be add to the end of `llBracket`. Another process here is to add a string "complete" to the `hmOperator` on the its size as the key.

All above discussion relate to " \wedge " and " \vee " schema operators. Let us now see other operators.

The " \Rightarrow " and " \Leftrightarrow " schema operators will be changed to their equivalent forms. These equivalent forms have been discussed on the previous sections. Thus, the $\mathbf{A} \Rightarrow \mathbf{B}$ schema calculus will be changed to $\neg \mathbf{A} \vee \mathbf{B}$. On the other hand, the $\mathbf{A} \Leftrightarrow \mathbf{B}$ schema calculus will be changed to $(\mathbf{A} \Rightarrow \mathbf{B}) \wedge (\mathbf{B} \Rightarrow \mathbf{A})$.

The first sub-section discusses an implementation of the implication operator.

An Implication Operator

There are three outer cases for implication specified in our system. They are grouped based on the value of `getI`.

The first one is as follows:

```
if (getI-2 == 0){
  Parser.llSchCal.set(getI, "\\zor");
  Parser.llSchCal.add(getI-1, "\\znot");
  // irrelevant code
}
```

It is a simple case, an infix implication of two operands. The above code will change an implicative schema calculus to its equivalent as shown in the above discussion.

The second case is shown by the code as follows:

```
else if (getI-1 == 0){
  Parser.llSchCal.set(getI, "\\zor");
  if (isBack){
    Parser.llSchCal.add(getI, "\\znot");
    hmOperator.put(hmOperator.size()-2, "complete");
    llBracket.set(llBracket.indexOf("modified_later"), Parser.llSchCal.get(getI).toString() + "_" + Parser.llSchCal.get(getI-1).toString());
    hmOperator.put(hmOperator.size()-1, "complete");
    Parser.llSchCal.remove(getI);
    Parser.llSchCal.remove(getI-1);
    // irrelevant code
    continue;
  }
}
```

This case indicates the right operand has been processed previously. There are two further checks here about a content of the last position of the `llBracket` list.

If it does not equal `)`, the next process is as shown below:

```
if (!llBracket.getLast().toString().trim().equals("))"){
  StringTokenizer stBracks = new StringTokenizer(
    llBracket.getLast().toString(), "\\)", true);
  // irrelevant code
  while (stBracks.hasMoreElements()){
    temp = stBracks.nextElement().toString().trim();
    if (!temp.isEmpty()){
      if (temp.equals("\\)")){
        temp = temp+ stBracks.nextElement().toString();
        if (temp.startsWith("\\_")){
          temp = Parser.llSchCal.getLast().toString().trim() + temp;
          Parser.llSchCal.set(Parser.llSchCal.size()-1,temp);
        }
      } else if (temp.equals("\\Delta")){
        temp = temp + stBracks.nextElement().toString();
      }
      else if (temp.equals("\\Xi")){
        // the same code as the above block
      }
      else countOperator--;
      temp1 = temp;
    }
  }
}
```

```

    }
    else{
        Parser.llSchCal.add(getI ,temp);
        getI++;
    }
}
}
if (!temp1.isEmpty()){
    Parser.llSchCal.add(getI ,temp1);
    getI++;
}
Parser.llSchCal.add(getI , "\\znot");
// irrelevant code
}

```

The above check is to process an implication which its right operand is not enclosed with brackets.

For an implication with enclosed brackets, the code is as follows:

```

else if (llBracket.getLast().toString().trim().equals("")){
    StringTokenizer stBracks = new StringTokenizer(
llBracket.get(llBracket.size()-2).toString(),"_\\\"", true);
    // irrelevant code
    temp1 = "";
    while (stBracks.hasMoreElements()){
        // irrelevant code

        if (isFront){
            Parser.llSchCal.add(getI , "\\znot");
            getI++;
        }
        // the same code as the above code
    }
    if (!temp1.isEmpty()){
        if (isFront){
            switch (temp1) {
                case "\\zand":
                    temp1 = "\\zor";
                    break;
                case "\\zor":
                    temp1 = "\\zand";
                    break;
            }
        }
        Parser.llSchCal.add(getI ,temp1);
        if (isBack){
            getI++;
            Parser.llSchCal.add(getI , "(");
            getI++;
            Parser.llSchCal.add(getI , ")");
        }
    }
    // irrelevant code
}
}

```

The third case is when both left and right operands have been processed previously.

```

else if (getI == 0){
    Parser.llSchCal.set(getI , "\\zor");
}

```

```
if (llBracket.getLast().toString().trim().equals("")){
```

The first code afterwards is a do-while loop as follows:

```
do{
  if (llBracket.get(i).toString().equals("(")){
    // see the below code for detail
  }
  else i--;
}
while(i >=0);
```

It contains an if blocks. The detail of the above code is shown as follows:

```
j = i+1;
do{
  // irrelevant code
}
while (j < llBracket.size()-2);

for (j=0;j< llTemp.size();j++){
  // irrelevant code
}
if (countO != countC){
  // irrelevant code
}
else{
  Parser.llSchCal.add(getI, llBracket.getLast().toString());
  Parser.llSchCal.add(getI, llBracket.get(i).toString());
  Parser.llSchCal.add(getI, "\\znot");
  // irrelevant code
  do{
    StringTokenizer stBracks = new StringTokenizer
    (llTemp.get(j).toString(), "_|\\", true);
    temp1 = "";
    while (stBracks.hasMoreElements()){
      temp = stBracks.nextElement().toString().trim();
      if (!temp.isEmpty()){
        if (temp.equals("\\")){
          temp=temp+stBracks.nextElement().toString();
          if (temp.startsWith("\\-")){
            // irrelevant code
          }
          else if (temp.equals("\\Delta")){
            // irrelevant code
          }
          else if (temp.equals("\\Xi")){
            // several lines of irrelevant code
          }
          temp1 = temp;
        }
        else if (temp.equals("(")||temp.equals(")")){
          temp1 = temp;
        }
        else {
          Parser.llSchCal.add(getI, temp);
          getI++;
        }
      }
    }
  }
  if (!temp1.isEmpty()){
```



```

        // irrelevant code
    }
    j++;
}
while (j < llTemp.size());
// irrelevant code
}

```

Another if block after the above code is shown as follows:

```

if (llBracket.get(j-1).toString().trim().equals("")){
// irrelevant code
do{
    if (llBracket.get(i).toString().equals("(")){
        j = i+1;
        do{
            llTemp.add(llBracket.get(j).toString().trim());
            j++;
        }
        while (j < k);
        for (j=0;j< llTemp.size();j++){
            // irrelevant code
        }
        if (countO != countC){
            // irrelevant code
        }
        else break;
    }
    else i--;
}
while(i >=0);
Parser.llSchCal.add(oldI ,llBracket.get(k+1).toString());
Parser.llSchCal.add(oldI ,llBracket.get(i).toString());
i=Parser.llSchCal.size()-2;
j=llTempImpl.indexOf("\\zimp")-1;
do{
    if (Parser.llSchCal.get(i).toString().equals(llTempImpl.get(j).toString())){
        k=j;
        j--;
        i--;
    }
    else {
        if (Parser.llSchCal.get(i).toString().equals("\\znot")) i--;
    }
}
while(i >=0);
i=k-1;
do{
    Parser.llSchCal.add(0 ,llTempImpl.get(i).toString());
    i--;
}
while(i >=0);
// irrelevant code
}

```

Let us move to an implementation of the " \Leftrightarrow " schema operator.

A Bi-implication Operator

Our system changes this operator into its equivalent form which is on the form of a conjugation of two implications forms. It is shown as follows:

```
else if (Parser.l1SchCal.get(getI).toString().startsWith
("\\") && Parser.l1SchCal.get(getI).toString().equals("\\zeq")){
```

It has two cases. The first one is given as follows:

```
if (getI-2 == 0){
  Parser.l1SchCal.set(getI, "\\zimp");
  oldI = getI;
  getI++;
  Parser.l1SchCal.add(getI, "(");
  getI++;
  Parser.l1SchCal.add(getI, ")");
  getI++;
  Parser.l1SchCal.add(getI,
  Parser.l1SchCal.get(oldI-1).toString());
  getI++;
  Parser.l1SchCal.add(getI,
  Parser.l1SchCal.get(oldI-2).toString());
  getI++;
  Parser.l1SchCal.add(getI, "\\zimp");
  getI++;
  Parser.l1SchCal.add(getI, "(");
  getI++;
  Parser.l1SchCal.add(getI, ")");
  getI++;
  Parser.l1SchCal.add(getI, "\\zand");
  getI = oldI;
}
```

The second case is shown as follows:

```
else if (getI-1 == 0){
  Parser.l1SchCal.set(getI, "\\zimp");
  oldI = getI-1;
  getI++;
```

It is then divided into two further cases which the first is shown as follows:

```
if (getI == Parser.l1SchCal.size()){
  Parser.l1SchCal.add("(");
  Parser.l1SchCal.add(")");
  Parser.l1SchCal.add(Parser.l1SchCal.get(oldI).toString());
  getI = Parser.l1SchCal.size();
}
```

Below is the second further case:

```
else if (getI < Parser.l1SchCal.size()){
  // irrelevant code
  if (getI < Parser.l1SchCal.size()){
    Parser.l1SchCal.add(getI, ")");
    getI++;
    if (getI < Parser.l1SchCal.size()){
      // irrelevant code
      if (getI == Parser.l1SchCal.size())
```

```

        getI = Parser.llSchCal.size();
    }
    else{
        // irrelevant code
    }
}
else {
    // irrelevant code
}
}
}

```

Another further process is given as follows:

```

if (!llBracket.getLast().toString().trim().equals("")){
    StringTokenizer stBracks = new StringTokenizer
    (llBracket.getLast().toString(),"-|\\", true);
    // irrelevant code
    while (stBracks.hasMoreElements()){
        temp = stBracks.nextElement().toString().trim();
        if (!temp.isEmpty()){
            if (temp.equals("\\\\")){
                temp = temp + stBracks.nextElement().toString();
                if (temp.startsWith("\\\\_")){
                    // irrelevant code
                }
                else if (temp.equals("\\\\Delta")){
                    // irrelevant code
                }
                else if (temp.equals("\\\\Xi")){
                    // irrelevant code
                }
                else countOperator--;
                temp1 = temp;
            }
            else{
                // irrelevant code
            }
        }
    }
}
if (!temp1.isEmpty()){
    // irrelevant code
}
Parser.llSchCal.add(getI, "\\znot");
// irrelevant code
}

```

Above is to process the content of last position of llBracket which is not equal to ")". Otherwise, code is given as follows:

```

else if (llBracket.getLast().toString().trim().equals("")){
    Parser.llSchCal.add(oldI, ")");
    Parser.llSchCal.add(oldI, "(");
    getI = getI + 2;
    StringTokenizer stBracks = new StringTokenizer(
    llBracket.size()-2).toString(),"-|\\", true);
    temp1 = "";
    while (stBracks.hasMoreElements()){
        temp = stBracks.nextElement().toString().trim();
        if (!temp.isEmpty()){
            if (temp.equals("\\\\")){
                temp = temp + stBracks.nextElement().toString();
            }
        }
    }
}

```

```

        if (temp.startsWith("\\_")){
            // irrelevant code
        }
        else if (temp.equals("\\Delta")){
            // irrelevant code
        }
        else if (temp.equals("\\Xi")){
            // irrelevant code
        }
        temp1 = temp;
    }
    else {
        if (getI < Parser.llSchCal.size()-1){
            Parser.llSchCal.add(getI,temp);
            getI++;
        }
        else {
            Parser.llSchCal.add(temp);
            getI++;
        }
        llTemp.add(temp);
    }
}
}
if (!temp1.isEmpty()){
    if (getI < Parser.llSchCal.size()-1){
        Parser.llSchCal.add(getI,temp1);
        getI++;
    }
    else {
        Parser.llSchCal.add(temp1);
        getI++;
    }
    llTemp.add(temp1);
}
// see the below code for details

```

The details of the above code are shown as follows:

```

if (getI < Parser.llSchCal.size()-1){
    Parser.llSchCal.add(getI,"");
    getI++;
}
if (getI < Parser.llSchCal.size()-1){
    Parser.llSchCal.add(getI,"");
    getI++;
}
if (getI < Parser.llSchCal.size()-1){
    Parser.llSchCal.add(getI,"\\zimp");
    getI++;
}
if (getI < Parser.llSchCal.size()-1){
    Parser.llSchCal.add(getI,"");
    getI++;
}
if (getI < Parser.llSchCal.size()-1){
    Parser.llSchCal.add(getI,"");
    getI++;
}
if (getI < Parser.llSchCal.size()-1){
    Parser.llSchCal.add(getI,"\\zand");
}
}
else{
    Parser.llSchCal.add("\\zand");
}
}
else{

```

```

        // the same code as above
    }
}
else{
    // the same code as above
}
}
else{
    // the same code as above
}
}
else {
    the same code as above
}
}
else {
    the same code as above
}
}
for (int i=llTemp.size()-1;i>=0;i--){
    Parser.llSchCal.add(oldI, llTemp.get(i).toString());
}
// irrelevant code

```

Thus, these lines of code are symmetrical.

Below is a discussion on an implementation of the negation operator.

A Negation Operator

The " \neg " schema operator will be processed as follows:

```

else if (Parser.llSchCal.get(getI).toString().
startsWith("\\") && Parser.llSchCal.get(getI).toString().equals("\\znot")){
    countOperator++;
    if (getI-1 > -1){
        llBracket.add(Parser.llSchCal.get(getI).toString()
+ "_"+ Parser.llSchCal.get(getI-1).toString());
        hmOperator.put(hmOperator.size(),"complete");
        // irrelevant code
        if (getI > 0){
            // irrelevant code
            llBracket.add(llBracket.size()-1,"modified_later");
            hmOperator.put(hmOperator.size(), hmOperator.
get(hmOperator.size()-1).toString());
            hmOperator.put(hmOperator.size()-2, "back");
        }
        else{
            isBack = false;
            isFront = true;
        }
        if (Parser.llSchCal.size() > 0 && Parser.llSchCal.
get(getI).toString().equals("(") && Parser.llSchCal.
get(getI+1).toString().equals(")")){
            if (llBracket.size()-2 >= 0){
                llBracket.add(llBracket.size()-2, "(");
            }
            else {
                llBracket.add(0,"(");
            }
        }
        llBracket.add(")");
    }
}

```

```

    // irrelevant code
  }
}
else if (getI-1 == -1){
  llBracket.add(Parser.llSchCal.get(getI).toString());
  hmOperator.put(hmOperator.size(),"front");
  // irrelevant code
}
}

```

Following is a discussion on an implementation of separators.

Separators in a Schema Calculus Definition

Separators were specified also in our system. Processes on the separator require information provided by our parser. Please see Chapter 4 for a separator process.

The below code shows a process on a separator on the `schConstruction` function:

```

else if (Parser.llSchCal.get(getI).toString().equals("separator")){
  Parser.llSchCal.remove(getI);
  for (int ggetI = getI-1;ggetI>=0;ggetI--){
    Parser.llSchCal.remove(ggetI);
  }
  getI = 0;
  break;
}

```

The above code is to process a separator found in schema calculus. Our system is able to expand more than one paragraph of schema calculus in one zed box. This is achieved since a separator of each line of paragraphs is recorded in our parser.

Following discussion relates to other processes on the `schConstruction` function.

Other Processes

A read of other than above strings of operators as discussed above makes a value of the variable `getI` be updated by 1. This process is continued until conditions on while loop is not met. After the processes in this while loop, the `llSchCal` list might be empty as another result of above processes.

Afterwards, contents of the `llBracket` list will be processed. The first round of process is on contents of the `llBracket` which starts with a "(" and ends with a ")". It will be repeated so long the `llBracket` does not run out of size and a variable `isOperate` is true. It is shown as follows:

```

int n=-1;
m=0;
if (n == -1) n=m;

```

```

while (m<llBracket.size() && isOperate) {
    temp = llBracket.get(m).toString().trim();
    if (temp.equals("(") && llBracket.get(m+2).toString().trim().equals("))")){
        temp1 = llBracket.get(m+1).toString().trim();
        isOperate = operate(temp1, index, tempVar, tempPred, true, currIdx);
        if (isOperate && tempPred.size() > 0) {
            ...
        }
    }
    else if (temp.equals("\\znot") && m==0) {
        isOperate = operate(temp, index, tempVar, tempPred, false, currIdx);
        llBracket.remove(m);
        hmOperator.remove(n);
        n++;
    }
    else {
        m++;
    }
}

```

"..." hides several lines which are intended to rewrite contents of `tempPred` so they form better writing of a Z specification.

The second round is otherwise; the above conditions are not met. It is shown as follows:

```

m = 0;
if (n == -1) n=m;
while (m<llBracket.size() && isOperate){
    temp = llBracket.get(m).toString().trim();
    // see the below code
}

```

The first code contains other processes before the contents of `tempPred` are processed. It is shown as follows:

```

if (!temp.equals("(") && !temp.equals("))")){
    if (temp.startsWith("\\zall") || temp.startsWith("\\zexi")){
        tempPred.add(temp);
        temp = hiddenVars.trim();
        if (temp.contains("\\zbar")){
            temp = temp.substring(0, temp.indexOf("\\zbar")).trim();
        }
        temp1 = getVarVal(temp, tempVar);
        llBracket.remove(m);
        llBracket.set(m, llBracket.get(m).toString().trim() + "\\zhide_" + "(" + temp1 + ")");
        hmOperator.remove(n);
        n++;
        tempVar.add("finish_hidden_variables");
        continue;
    }
    isOperate = operate(temp, index, tempVar, tempPred, false, currIdx);
    if (tempPred.size() > 0 && (tempPred.getFirst().toString().startsWith("\\forall") || tempPred.getFirst().toString().startsWith("\\exists"))){
        temp = tempPred.getFirst().toString();
        tempPred.removeFirst();
    }
}

```

```
}
```

The `getVarVal` function has ever been discussed earlier above. Thus, it will not be discussed again here. A schema calculus definition in this example does not contain any brackets.

Below is other process on the second code:

```
else if (temp.equals("(")){
    if (llBracket.get(m+2).toString().trim().equals("))"){
        if (llTemporary.size() > 0){
            llTemporary.stream().forEach((llTemporary1) -> {
                tempPred.add(llTemporary1.toString());
            });
            llTemporary.removeAll(llTemporary);
            checkI = tempPred.size();
            currIdx = checkI;
            llBracket.remove(m+2);
            llBracket.remove(m+1);
            hmOperator.remove(n);
            n=n+1;
            llBracket.remove(m);
            continue;
        }
    }
    if (m > 0 && llBracket.get(m-1).toString().trim().equals("))"){
        tempPred.add(" outer");
    }
    tempPred.add("(");
    m++;
    continue;
}
```

The last process in this second code is as follows:

```
else if (temp.equals("))"){
    m++;
    if (tempPred.getLast().toString().trim().equals(" here")){
        tempPred.set(tempPred.size()-2,tempPred.get
            (tempPred.size()-2).toString().trim()+"");
    }
    else tempPred.set(tempPred.size()-1,tempPred.
        getLast().toString().trim()+"");
    continue;
}
```

All above processes are specified in the `schConstruction` function. As seen above, another function has been called. It is the `operate` function.

6.2.3 Expansion Processes in operate function

Before the `operate` function is executed, there are other processes that take place first. These processes are shown as follows:

```
StringTokenizer stS = new StringTokenizer(s," |\\|[[|]|(|)" , true);
if (tempVar.size() > 0 && tempVar.contains(" finish hidden variables")){
    int i = 0;
    do {
```



```

tempHiddenVars.add(tempVar.get(i).toString());
i++;
}
while (!tempVar.get(i).toString().equals(" finish hidden variables"));
do{
tempVar.remove(i);
i--;
}
while (i >= 0);
}
while (stS.hasMoreElements()){
temp = stS.nextElement().toString().trim();
if (!temp.isEmpty()){
if (temp.equals("\\")){
temp = temp + stS.nextElement().toString();
if (temp.startsWith("\\-")){
temp =schName.getLast().toString().trim()+temp;
schName.remove(schName.size()-1);
}
else if (temp.equals("\\Delta")){
temp = temp + stS.nextElement().toString();
}
else if (temp.equals("\\Xi")){
temp = temp + stS.nextElement().toString();
}
}
schName.add(temp);
}
}
}

```

The above code will not be executed in this example as none of them was specified.

As mentioned earlier (see Section 6.1 on page 130), each schema operator has its own operation. Thus, there are several schema operators specified in our system as cases below:

```

int i=0;
outer:
do{
switch (schName.get(i).toString()) {
case "\\zand":
case "\\zor":
...
break;
case "\\znot":
...
break;
case "\\zfor":
...
break;
case "\\hide":
case "\\zhide":
...
break;
case "\\zcmp":
...
break;
default:
i++;
break;
}
}

```

```

    }
  }
  while (i < schName.size());
  return isExpand;

```

The `operate` function returns a boolean value, which is determined by a value of `isExpand`.

Following are cases that are specified in this system to detail above "...".

Conjunction and Disjunction

There are several cases for these operators. The first case is each of these operators is in the first index of a list whose size is 1. In other words, each of them is the outer operator. Expansion, normalisation, and collapse are not performed in this first case.

The second one is the size of this list is greater than 1. It denotes that left arguments of each of this operator have been processed earlier. In this case, the right argument will be expanded, normalised, and collapsed.

In other case, the index is greater than 0 and less than the biggest index. For this case, each of these operators will be an infix operator; it has two arguments. Each of this argument will be expanded, and normalised. Afterwards, the updated states of these arguments are collapsed together.

The last case is the index is greater than 0 and equals to the biggest index. In this case, right arguments have been processed earlier. The left argument will be expanded, normalised, and collapsed.

A schema conjunction flagged by an `\zand` will be replaced by `\land`. On the other hand, a schema disjunction flagged by an `\zor` will be replaced by `\lor`. Afterwards, the `expand` function will be called for each argument which belongs to this operator.

Negation

The first case is a zero index and one sized list. This case means this operator will negate an argument that has been processed earlier. There is no further process.

The other case is to negate an argument immediately following it. The negation process consists of an expansion, normalisation, and negation. Expansion and normalisation are the same as both processes on conjunction and disjunction operators.

The `negateSch` function will be called. If a predicate starts with a quantifier, a " \neg " will be added after a " \bullet " and the rest of this predicate after the " \bullet " will be enclosed with brackets. A " \wedge " will be changed into " \vee ". Otherwise, a " \neg " and a "(" will be added at the beginning of a predicate,

and this predicate will end with a `)`". A `"^"` or a `"\"` will be changed into `"\vee"`.

Collapse is not performed on the negation operation as the predicates have just been negated. Variables and negated predicates are difficult to collapse to their previous variables.

The `\znot` schema negation will be replaced by `\lnot`. The specification and its expanded schema can be seen in Appendix I on page 330.

Renaming

A schema that is followed by this operator will be expanded first, then normalised. A renaming will come afterwards. Collapse is not performed on the renaming operation. This is because the renaming process alters one or more variables as well as predicate lines which contain these variables, but these variables are still specified in the same schema. Thus, these altered variables and predicates cannot be collapsed to their original schema before expansion is performed. Renaming will call the `rename` function.

The name of the specification is `expandingsch5_2.tex` which can be seen in Appendix J on page 331 as well as its expanded schema generated by our system.

Hiding

A schema starting with this operator will be expanded, and normalised. Afterwards, a hiding will be performed. Collapse is not performed on the hiding operation. Hiding calls the `hide` function. This function returns a Boolean value which indicates a schema can be expanded or not.

An example used here is `expandingsch6_1.tex`. This specification and its expanded schema can be seen in Appendix K on page 332.

Composition

The first process in this operator makes each argument representing a schema be expanded, and normalised. The primed variables from the first argument will be renamed to common names; the non-primed variables from the second argument will be renamed to the common names as used in the first renaming.

Updated variables and predicates from these arguments will be conjoined into one list of variables and one list of predicates. Conjoin operation will call the `conjoin` function. Contents of the `tempVar1` first list of variables will be added to the `tempVar` list of variables. Contents of the first `tempPred1` list of predicates will be added to the `tempPred` list of predicates. As well as contents of the second `tempVar2` list of variables, so long a particular variable

is not available on the `tempVar` list, it will be added to this list. The process of a predicate part is quite different for a predicate which is available on the `tempPred` list of predicate. A list of predicates which are formed from normalisation will be updated if the same predicate has already been in the `tempPred` list.

Upon having these new lists which are `tempVar` and `tempPred`, hiding will be performed. These arguments will be collapsed later.

The Z specification used here is `expandingsch7_1.tex`. This specification and its expanded schema can be seen in Appendix L on page 333.

6.2.4 Expansion Processes in `expand` function

Let us go back to the `operate` function. As mentioned before, expansion is performed in this function. Expansion will call another function, `expand`, and it is shown as follows:

```

if (strNameSch.equals(schemaBox[0].nameSch)){
    // irrelevant code
}
else if (strNameSch.equals(schemaBox[0].nameSch+" '")){
    // irrelevant code
}
else if (strNameSch.equals("\\Delta_" + schemaBox[0].
nameSch)){
    // irrelevant code
}
else if (strNameSch.equals("\\Xi_" + schemaBox[0].
nameSch)){
    // irrelevant code
}
else{
    idxVar = tempVar.size();
    for (int i = 0; i < schemaBox[index].hmVar.size(); i++){
        var=schemaBox[index].hmVar.get(i).toString().trim();
        if (var.equals(schemaBox[0].nameSch)){
            // irrelevant code
        }
        else if (var.endsWith(" '") && var.equals(schemaBox[0].nameSch+" '")){
            // irrelevant code
        }
        else if (var.startsWith("\\Delta")){
            // several lines of irrelevant code
        }
        else if (var.startsWith("\\Xi")){
            // several lines of irrelevant code
        }
        else{
            // several lines of irrelevant code
        }
    }
}
}

```

This function will expand variables of a schema as shown above which might have the same name as a state schema, the primed version of a name of a

state schema, a " Δ " or a " Ξ " added to a name of a state schema, or expand variables of other schema names which each of them will be checked also to a name of a state schema. If a variable has already been available, the variable will not get the second copy of it.

Afterwards, predicates will be expanded. It is shown as follows:

```

idxPred = tempPred.size();
if (inclusion){
  for (int j = 0; j < schemaBox[0].llPred.size(); j++){
    pred = schemaBox[0].llPred.get(j).toString().trim();
    // irrelevant code
  }
}
else if (primed){
  for (int j = 0; j < schemaBox[0].llPred.size(); j++){
    pred = schemaBox[0].llPred.get(j).toString().trim();
    // irrelevant code
  }
}
else if (delta || xi){
  for (int j = 0; j < schemaBox[0].llPred.size(); j++){
    pred = schemaBox[0].llPred.get(j).toString().trim();
    // irrelevant code
  }
}
if (!llPred.isEmpty()){
  // irrelevant code
}
if (index > 0){
  count = countPred;
  for (int i = 0; i < schemaBox[index].llPred.size(); i++){
    // irrelevant code
  }
}
}

```

This function returns a string containing the size of copied state variables, the total size of copied variables, the size of copied state predicates, the total size of copied predicates, the size of predicates, and the type schema whether it is an inclusion, primed, " Δ ", " Ξ " or not all of them. Our system expands a list of variables and predicates for each argument before expanding other argument.

6.2.5 Expansion Processes in normalised function

After the `expand` function was processed, normalisation will be performed. Normalisation will call `normalised` function. The first process on this function is to split a type by calling another helping function, `splitType`.

On redefining generic constant definitions, there is also `splitType` function which will split a type of a generic constant. On the other hand, in the expansion of schema calculus, this function will split a type of a variable that is normalised. These split types will be stored in `tempType` list.

The next process of `normalise` function is as follows:

```

for (int i=0;i<ingen.tempType.size();i++){
  oldVal = ingen.tempType.get(i).toString().trim();
  if (hmGlobalCons.containsKey(oldVal)){
    newVal = hmGlobalCons.get(oldVal).toString().trim();
    Set set = hmGlobalCons.entrySet();
    Iterator iterator = set.iterator();
    while (iterator.hasNext()){
      Map.Entry enumGlob = (Map.Entry)iterator.next();
      temp = enumGlob.getKey().toString();
      if (newVal.contains(temp)){
        newVal = newVal.replace(temp,enumGlob.getValue().toString());
      }
    }
  }
  else newVal = oldVal;
  // see the first detail
}
// see the second detail
return val;

```

The first detail is given as follows:

```

if (newVal.equals("\\nat")||newVal.equals("\\natone")){
  newVal = val.replace(oldVal, "\\integer");
  val = newVal.trim();
}
else if (newVal.contains("\\natone")){
  temp = val.substring(0,val.indexOf(oldVal)).trim();
  newVal = newVal.replace("\\natone", "\\integer");
  newVal = temp + "_" + newVal + "_" + val.substring
    (val.indexOf(oldVal)+oldVal.length()).trim();
  val = newVal.trim();
}
else if (newVal.contains("\\nat")){
  // the same as previous code, but match the string
}
if (newVal.contains("\\seq")){
  newVal = val.substring(val.indexOf("\\seq")+4).trim();
  newVal = "\\pset(\\integer_\\cross_+newVal+)";
  val = newVal;
}
else if (newVal.contains("\\seqone")){
  // the same as previous code, but match the string
}
ingen.tempType.set(i, val);

```

The second detail is as follows:

```

if (val.contains("\\tfun") || val.contains("\\fun") || val.contains("\\pfun")
|| val.contains("\\tsurj") || val.contains("\\surj")
|| val.contains("\\psurj") || val.contains("\\psur")
|| val.contains("\\tinj") || val.contains("\\inj")
|| val.contains("\\pinj") || val.contains("\\bij")
|| val.contains("\\finj") || val.contains("\\ffun")){
  idx = 0;
  if (val.contains("\\tfun")){
    oldVal = val;
    do{
      newVal = val.substring(0, val.indexOf("\\tfun",idx)).trim();
      val = val.substring(val.indexOf("\\tfun", idx)+5).trim();
    }
  }
}

```

```

        val = "\\pset("+ newVal + "_\\cross_" + val + ")";
        idx = val.indexOf("\\tfun", idx);
    }
    while (!val.contains("\\tfun"));
}
idx = 0;
if (val.contains("\\fun")){
    ...
}
idx = 0;
if (val.contains("\\pfun")){
    ...
}
idx = 0;
if (val.contains("\\tsurj")){
    ...
}
idx = 0;
if (val.contains("\\surj")){
    ...
}
idx = 0;
if (val.contains("\\psurj")){
    ...
}
idx = 0;
if (val.contains("\\psur")){
    ...
}
idx = 0;
if (val.contains("\\tinj")){
    ...
}
idx = 0;
if (val.contains("\\inj")){
    ...
}
idx = 0;
if (val.contains("\\pinj")){
    ...
}
idx = 0;
if (val.contains("\\bij")){
    ...
}
idx = 0;
if (val.contains("\\finj")){
    ...
}
idx = 0;
if (val.contains("\\ffun")){
    ...
}
}
}

```

"..." means the same as previous code, but match the string. This function returns a string stored in `val`.

The normalisation result was written in the form: variable, old type, new type after normalisation. The same as the `expand` function, normalisation

will be performed on every variable on each argument before normalising other argument. Thus, a sequence of processes should be: expanding variables, expanding predicates, and normalising variables for each argument.

Thus, several normalisation rules which are specified in our system are as follows:

- Every "N" or "N₁" in a declaration part is rewritten to a type of "Z".
- Every "seq" or "seq₁" is changed to $\mathbb{P}(\mathbb{Z} \times newVal)$, *newVal* is a type which comes after "seq" or "seq₁". The previous rule is applied also to *newVal*.
- Every function is changed to a pair of its left hand side type and its right hand side one. Both above rules are also applied to the type in the left and in the right.

As can be known from the above description, normalisation is applied also to other schema operators for the sake of simplicity.

6.2.6 Expansion Processes in collapse function

Another process of `operate` function is to collapse associated schemas. The latter function is `collapse`. Collapse means to fold state variables and state predicates into the name of the state schema as an inclusion, a primed, a "Δ" or a "Ξ" of the state schema.

Collapsing results from our example are different with both earlier functions, this function is called by passing both arguments simultaneously.

In addition to that process, this function also performs a simple simplification on a composition operation. Earlier section has discussed composition operation, please see the associated section to get better understanding how it works.

The first process in simplifying a resultant schema from a composition is to find out whether there is a predicate which contains a renamed state variable. It is shown as follows:

```

for (int j =0;j< schemaBox[0].hmVar.size();j++){
    string1 = schemaBox[0].hmVar.get(j).toString().trim()+"0";
    tPred.add(string1);
    for (i=1;i< tempPred.size();i++){
        if (tempPred.get(i).toString().trim().contains(string1)){
            string2 = tempPred.get(i).toString().trim();
            tPred.add(i);
        }
    }
}
}

```


The next process will find the same renamed state variables, which is given as follows:

```

for (i=0;i<schemaBox[0].hmVar.size();i++){
string1 = schemaBox[0].hmVar.get(i).toString().trim()+"0";
repeat:
for (int j=0;j<tPred.size();j++){
string2 = tPred.get(j).toString().trim();
if (string2.equals(string1)){
llLen = new LinkedList();
for (k=j+1;k<tPred.size();k++){
// see the first detail
}
sortLinkedList(llLen);
len = Integer.parseInt(llLen.getFirst().
toString().trim());
repeat1:
for (k=j+1;k<tPred.size();k++){
// see the second detail
}
llLen.removeAll(llLen);
break;
}
}
}

```

The first detail is given as follows:

```

string3 = tPred.get(k).toString().trim();
if (string3.length() < 4){
if (string3.matches("[0-9]+")){
string3 = tempPred.get(Integer.parseInt(tPred.get(k).
toString().trim())).toString().trim();
llLen.add(string3.length());
}
}
else break;

```

The above code will store the length of a predicate that contains renamed variables.

`sortLinkedList` function will sort these predicates based on their length. The shortest length will be put on the smallest index.

The second detail is as follows:

```

string3 = tPred.get(k).toString().trim();
if (string3.length() < 4){
string3 = tempPred.get(Integer.parseInt(tPred.get(k).
toString().trim())).toString().trim();
if (string3.length() == len){
if (string3.contains("=")){
getI = k;
string4 = string3.substring(0, string3.indexOf("=")).trim();
string5 = string3.substring(string3.indexOf("=")+1).trim();
if (string5.endsWith("\\\\"+"\\\\")){
string5 = string5.substring(0, string5.
lastIndexOf("\\\\"+"\\\\")).trim();
}
else if (string5.endsWith("\\land")){
string5 = string5.substring(0, string5.

```

```

        lastIndexOf("\\land").trim();
    }
    else if (string5.endsWith("\\lor")){
        string5 = string5.substring(0, string5.
            lastIndexOf("\\lor").trim());
    }
    if (string4.equals(string1)){
        // see the first check
    }
    else if (string5.equals(string1)){
        // see the second check
    }
    else{
        continue;
    }
}
break;
}
}

```

Our system just recognizes an equality operator to separate the left and right operands of a predicate containing a renamed variable; it is another limitation of our system. The left operand is stored on **string4** string, whereas the right one is on **string5** string. There are two further checks here, which are given as comments on the above code.

The first check is given as follows:

```

for (int l=j+1;l<tPred.size();l++){
    if (l != getl && tPred.get(l).toString().trim().length() < 4){
        string4 = tempPred.get(Integer.parseInt(tPred.
            get(l).toString().trim())).toString().trim();
        string4 = string4.replaceAll(string1, string5);
        tempPred.set(Integer.parseInt(tPred.get(l).toString().trim()), string4);
        isFound = true;
    }
}
if (isFound){
    tempPred.remove(Integer.parseInt(tPred.get(k).toString().trim()));
    for (int m=k+1;m<tPred.size();m++){
        if (tPred.get(m).toString().trim().length() < 4
            && Integer.parseInt(tPred.get(m).toString().trim())
            > Integer.parseInt(tPred.get(k).toString().trim()))
            tPred.set(m, Integer.parseInt(tPred.get(m).toString().trim()) - 1);
    }
    tPred.remove(k);
    isFound = false;
}
else {
    string3 = string3.replaceAll(string1, string5);
    tempPred.set(getl, string3);
}
}

```

It is for a case that **string4** string equals to the renamed variable **string1**.

The second check has the code as the first check, but change **string4** string to **string5** string.

6.2.7 Expansion Processes in negateSch function

This function is called in a negation process which will be used by negation operator. In general, this function combines two other functions which have been discussed above which are `expand`, and `normalised`. This combination was implemented by calling both functions inside this function. Having those processes performed, predicates will be negated then.

Its code is shown as follows:

```
strIdx = expand(strNameSch, index, tempVar, tempPred);
// irrelevant code
for (int i = idxS; i < tempVar.size(); i++){
    count++;
    var = tempVar.get(i).toString().trim();
    var = var.substring(0, var.indexOf(":")).trim();

    val = tempVar.get(i).toString().trim();
    val = val.substring(val.indexOf(":")+1).trim();
    if (val.endsWith("\\"+ "\\")){
        val = val.substring(0, val.lastIndexOf("\\"+ "\\"));
    }
    newVal = normalised(var, val).trim();

    if (!newVal.equals(val)){
        // irrelevant code
    }
}
...
```

"..." will perform a negation process and it is shown as follows:

```
// irrelevant code
for (int i = idxS; i < count; i++){
    if (tempPred.get(i).toString().trim().startsWith("\\forall") ||
        tempPred.get(i).toString().trim().startsWith("\\all") ||
        tempPred.get(i).toString().trim().startsWith("\\exists") ||
        tempPred.get(i).toString().trim().startsWith("\\exione") ||
        tempPred.get(i).toString().trim().startsWith("\\exists_1") ||
        tempPred.get(i).toString().trim().startsWith("\\exi")){

        strTemp = tempPred.get(i).toString().trim();
        if (strTemp.contains("\\spot") || strTemp.contains
            ("\\dot") || strTemp.contains("\\cbar")){
            do{
                prevIdxSpot = idxSpot;
                idxSpot = strTemp.indexOf("\\spot", idxSpot);
                len = 5;
                if (idxSpot == -1){
                    idxSpot = strTemp.indexOf("\\dot", prevIdxSpot);
                    len = 4;
                    if (idxSpot == -1){
                        idxSpot = strTemp.indexOf("\\cbar", prevIdxSpot);
                        len = 5;
                    }
                }
            }

            if (idxSpot > -1){
                if (!strTemp.substring(idxSpot+len).trim().contains("\\forall") &&
                    !strTemp.substring(idxSpot+len).trim().contains("\\all") &&
```

```

!strTemp.substring(idxSpot+len).trim().contains("\\exists") &&
!strTemp.substring(idxSpot+len).trim().contains("\\exione") &&
!strTemp.substring(idxSpot+len).trim().contains("\\exists_1") &&
!strTemp.substring(idxSpot+len).trim().contains("\\exi")){
    strTemp = strTemp.substring(0, idxSpot+len).
    trim() + "¬\\lnot_" + strTemp.substring(idxSpot+len).trim();

    if (tempPred.get(i).toString().trim().endsWith("\\land")){
        strTemp = strTemp.substring(0, strTemp.
        trim().lastIndexOf("\\"+"\\"));
        tempPred.set(i, strTemp + "\\lor");
    }
    else {
        strTemp = strTemp.substring(0, strTemp.
        trim().lastIndexOf("\\"+"\\"));
        tempPred.set(i, strTemp + "\\lor");
    }
    idxSpot = idxSpot + len;
}
}
}
while(idxSpot > -1);
idxSpot = 0;
}
}
else{
    if (tempPred.get(i).toString().trim().endsWith("\\"+"\\")){
        tempPred.set(i, tempPred.get(i).toString().
        substring(0, tempPred.get(i).toString().lastIndexOf("\\"+"\\")));
        tempPred.set(i, "\\lnot(" + tempPred.get(i) + "\\lor");
    }
    else if (tempPred.get(i).toString().trim().endsWith("\\land")){
        // has the same code as the above block,
        // but matches the string
    }
    else tempPred.set(i, "\\lnot(" + tempPred.get(i) + ")");
    if (i == count-1){
        if (tempPred.get(i).toString().trim().endsWith("\\lor"))
            tempPred.set(i, tempPred.get(i).toString().
            substring(0, tempPred.get(i).toString().lastIndexOf("\\lor")));
        if (!tempPred.get(idxS).toString().trim().startsWith("("))
            tempPred.set(idxS, "(" + tempPred.get(idxS).toString());
        tempPred.set(i, tempPred.get(i).toString() + ")");
    }
}
}
}
}
}

```

The idea of the above code is to put a "¬ (" in front of the predicate, a ")" at the end of the predicate before any separator or operator, and reverse this ending. The operator " \wedge " and the separator "\\ " will be changed to " \vee ", the operator " \vee " will be changed to " \wedge ". However, if a predicate starts with a quantifier, $\neg ($ will be put in front of the quantifier.

6.2.8 Expansion Processes in rename function

Renaming operation will call this function. It is given as follows:

```

do{
  for (int i=0;i< tempVar.size();i++){
    // see the first detail
  }

  if (isRenamed){
    // see the second detail
  }
}
while (oldV.size() > 0);

```

The first detail will be as follows:

```

if (tempVar.get(i).toString().startsWith(oldV.getFirst().toString())){
  temp = tempVar.get(i).toString().substring(0,
tempVar.get(i).toString().indexOf(":")).trim();
  temp1 = tempVar.get(i).toString().substring(
tempVar.get(i).toString().indexOf(":")).trim();
  if (temp.equals(oldV.getFirst().toString())){
    tempVar.set(i, newV.getFirst().toString() + "⌞" + temp1);
    isRenamed = true;
    break;
  }
}
}

```

The above code will rename a variable stored in `oldV` to a new name stored in `newV`.

Afterwards, each predicate containing the renamed variable will be altered also by renaming the old variable to the new name. It is given as follows which will describe the second detail:

```

for (int i=0;i< tempPred.size();i++){
  if (tempPred.get(i).toString().contains(oldV.getFirst().toString())){
    idx = 0;
    temp = tempPred.get(i).toString().trim();
    temp1 = "";
    do{
      if (temp.indexOf(oldV.getFirst().toString(),idx) > -1){
        idx = temp.indexOf(oldV.getFirst().toString(), idx);
        idxS = idx;
        do{
          temp1 = (temp1 + temp.charAt(idx)).trim();
          idx++;
          if (temp1.length() > oldV.getFirst().toString().length()){
            idx--;
            temp2 = ""+temp.charAt(idx);
            if (temp2.matches("[A-Za-z0-9?!'\\\"_ ]")){
              isRenamed = false;
              break;
            }
          }
          else{
            isRenamed = true;
            idxE = idx-1;
            break;
          }
        }
      }
    }
    while (idx < temp.length());
    if (isRenamed){

```

```

        if (idxS > 0 && idxE < temp.length()-1) {
            temp = temp.substring(0, idxS) + newV.getFirst().toString().
                trim() + "┌" + temp.substring(idxE+1).trim();
        }
        else if (idxS == 0){
            temp = newV.getFirst().toString().trim()
                + "┌" + temp.substring(idxE+1).trim();
        }
        else if (idxE == temp.length()-1){
            temp = temp.substring(0, idxS) + newV.
                getFirst().toString().trim();
        }
    }
}
else break;
temp1 = "";
}
while (idx < temp.length());
tempPred.set(i, temp);
}
}
// irrelevant code

```

6.2.9 Expansion Processes in hide function

The hiding operation calls this function. This function is shown as follows:

```

stS = new StringTokenizer(hiddenVars,"┌(|)|,");
while (stS.hasMoreElements()){
    temp = stS.nextElement().toString().trim();
    numOfHidVars++;
    for (int i=0;i<tempVar.size();i++){
        if (tempVar.get(i).toString().trim().startsWith
            ("\\Delta") || tempVar.get(i).toString().trim().startsWith("\\Xi")){
            // see the first detail
        }
        else if (tempVar.get(i).toString().trim().endsWith(" ") &&
            tempVar.get(i).toString().trim().startsWith(schemaBox[0].nameSch)){
            // see the second detail
        }
        else if (tempVar.get(i).toString().trim().equals(schemaBox[0].nameSch)){
            // see the third detail
        }
        if (oldI > -1){
            i = oldI;
        }
        if (tempVar.get(i).toString().startsWith(temp)){
            temp1 = tempVar.get(i).toString().substring(0,
                tempVar.get(i).toString().indexOf(":").trim());
            temp2 = tempVar.get(i).toString().substring
                (tempVar.get(i).toString().indexOf(":")+1).trim();
            if (temp2.endsWith("\\ "+"\\ ")) temp2 = temp2.
                substring(0, temp2.lastIndexOf("\\ "+"\\ ")).trim();
            if (temp1.equals(temp)){
                tempVar.remove(i);
                isHidden = true;
                oldI = -1;
                break;
            }
        }
    }
}

```

```

        else {
            oldI = -1;
            isHidden = false;
        }
    }
    else {
        oldI = -1;
        isHidden = false;
    }
}
if (isHidden){
    llHiddenVar.add(temp1 + "⌊:⌋" + temp2);
}
}
if (llHiddenVar.size() > 0 && numOfHidVars == llHiddenVar.size()){
    // see the fourth detail
}
else isHidden = false;
return isHidden;

```

It will hide associated hidden variables from `tempVar` list. Afterwards, these hidden variables will be quantified existentially on the first line of an associated predicate part.

The first detail is given as follows:

```

oldI = i;
temp1 = tempVar.get(i).toString().trim();
tempVar.remove(i);
if (temp1.endsWith("\\\\"+"\\\\")){
    temp1 = temp1.substring(0, temp1.lastIndexOf("\\\\"+"\\\\")).trim();
}
for (int j=0;j<schemaBox[0].hmVar.size();j++){
    temp2=schemaBox[0].hmVar.get(j).toString().trim();
    temp3 = temp2 + "⌊:⌋" + schemaBox[0].hmVal.get(temp2).toString().trim();
    if (j < schemaBox[0].hmVar.size()-1){
        tempVar.add(i, temp3+"\\\\"+"\\\\");
        i++;
        temp3 = temp2 + "⌊" + "⌋" + schemaBox[0].
hmVal.get(temp2).toString().trim();
        tempVar.add(i, temp3+"\\\\"+"\\\\");
        i++;
    }
    else{
        tempVar.add(i, temp3);
        i++;
        if (i == tempVar.size()-1){
            ...
        }
        else{
            ...
            tempVar.add(i, temp3);
        }
    }
}
}

```

The first "... " means the same as previous if block from the last fourth to second lines. The second "... " means the same as the last fourth to third lines of the same if block.

```

...
for (int j=0;j<schemaBox[0].hmVar.size();j++){
temp2=schemaBox[0].hmVar.get(j).toString().trim();
temp3 = temp2 + ";" + "_:" + schemaBox[0].hmVal.
get(temp2).toString().trim();
if (i < tempVar.size()-1){
tempVar.add(i, temp3+"\\"+"\\");
i++;
}
}
}

```

The above code details the second detail. "...” means the same as the top seven lines of previous if block.

```

...
for (int j=0;j<schemaBox[0].hmVar.size();j++){
temp2=schemaBox[0].hmVar.get(j).toString().trim();
temp3 = temp2 + "_:" + schemaBox[0].hmVal.get(temp2).toString().trim();
if (i < tempVar.size()-1){
tempVar.add(i, temp3+"\\"+"\\");
i++;
}
}
}

```

The above code details the third detail. "...” means the same as the top seven lines of first if block.

```

if (!tempPred.getFirst().toString().trim().
startsWith("\\zall") && !tempPred.getFirst().
toString().trim().startsWith("\\zexi")){
if (llHiddenVar.size() == 1){
temp = llHiddenVar.getFirst().toString();
temp = "\\exists_" + temp + "_\\spot_";
tempPred.add(0,temp);
llHiddenVar.remove(0);
}
else{
temp = "";
for (int i=0; i<llHiddenVar.size(); i++){
temp=temp+llHiddenVar.get(i).toString()+";_";
}
temp = temp.substring(0, temp.lastIndexOf(";"));
...
llHiddenVar.removeAll(llHiddenVar);
}
}
else{
temp = tempPred.getFirst().toString().trim();
temp = temp.replace("\\zall", "\\forall");
temp = temp.replace("\\zexi", "\\exists");
temp = temp.replace("\\zbar", "\\cbar");
tempPred.set(0, temp);
}
for (int i=1;i<tempPred.size();i++){
if (tempPred.get(i).toString().endsWith("\\\\"+"\\")){
tempPred.set(i, tempPred.get(i).toString().
substring(0, tempPred.get(i).toString().
lastIndexOf("\\\\"+"\\")).trim()+"_\\land");
}
}
}

```



```
isHidden = true;
```

The above code describes the fourth detail. ”...” means the same as previous if block, especially the last third and second lines.

Thus, a sequence of operations in `operate` function in general is: expanding, normalising, and collapsing. This sequence can be different for other operator such as negation in which `negateSch` function will be called also. Functions which will be called on each operator have been mentioned earlier.

After `operate` function finished, a value held by `isOperate` is returned to the caller which is `schConstruction` function. Next processes are performed on the latter function which is mainly to rewrite the predicates to get better appearances. The final results are shown in Fig. 6.1 on page 167, Fig. 6.2 on page 168, Fig. 6.3 on page 169, and Fig. 6.4 on page 170. Both figures were obtained by selecting **View** menu and clicking **Expansion** sub-menu from the GUI window of our system.

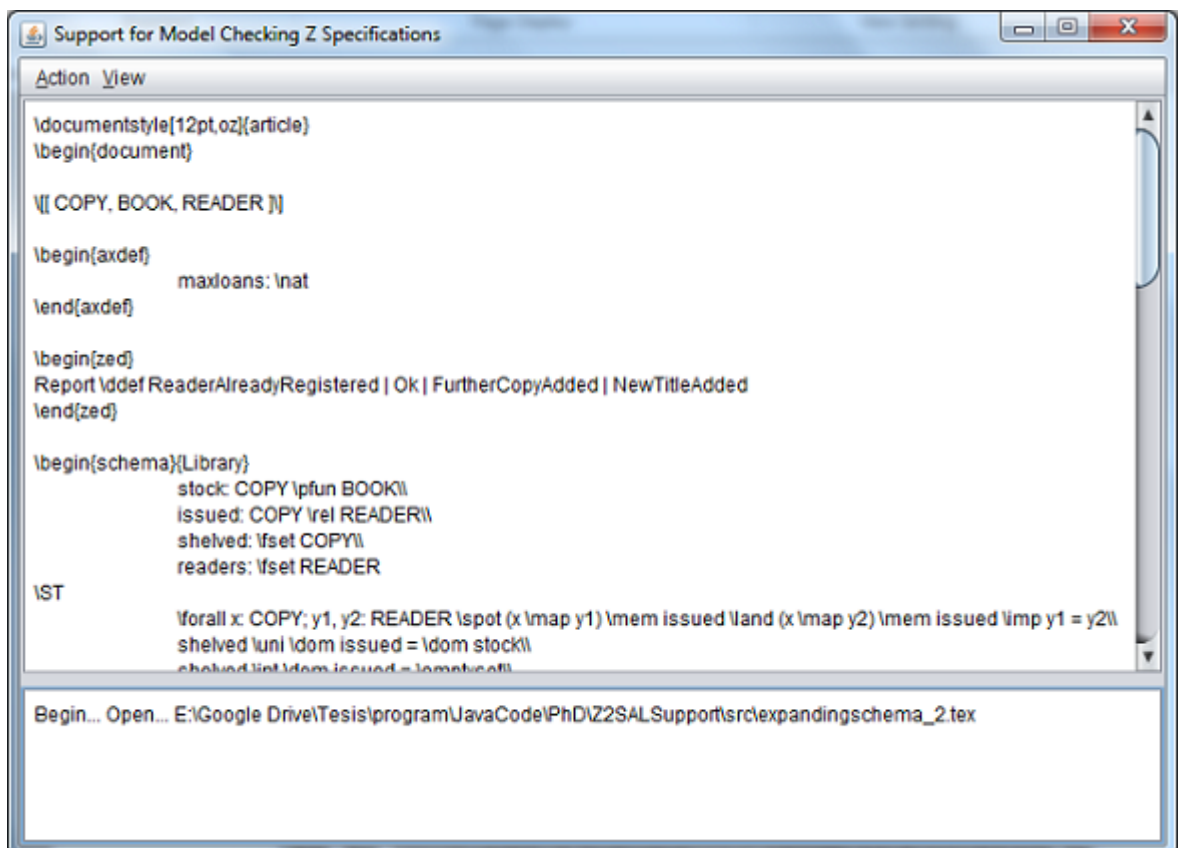


Figure 6.1: Finalising expansion process

```

shelved \in \dom issued = \emptyset
shelved \in \dom issued = \emptyset
\ran issued \subseteq readers
\forall r. readers \dot{\#}(\text{issued } r) \leq \text{maxloans}
\end{schema}

\begin{schema}{InitLibrary}
Library
\ST
shelved' = \emptyset
readers' = \emptyset
\end{schema}

\begin{schema}{EnterNewCopy}
\Delta Library
b? : BOOK
\ST
\exists c: COPY | c \in \text{mem} \wedge \text{stock} \dot{\#}(\text{stock}' = \text{stock} \vee \text{map } b?) \wedge \text{shelved}' = \text{shelved} \wedge \text{c} \in \text{issued}'
issued' = issued
readers' = readers
\end{schema}

\begin{schema}{AddCopy}

```

Figure 6.2: Finalising expansion process (continued)

Several functions were not put as headings of sub-sections of expansion processes. For example: `conjoin`, both `getVarVal`, `indexOfById`, `sortLinkedList`, and `splitType`. It is since they have been discussed on the discussion of other functions either in this chapter or previous chapter.

6.3 Conclusion

At the moment, Z2SAL does not support constructed schemas in which a schema is constructed by reusing other defined schemas operating by one or more schema operators as given above. If there is an operation between schemas, that operation is performed manually on the Z specification. The Z specification output is then passed to Z2SAL. In other words, a user should specify their specification in a style that does not use a constructed schema. The user should apply the schema operator and reflect this application in their specification. This method is then adapted to propose methods discussed earlier in this section, though neither all schema calculus on the Z

```

Support for Model Checking Z Specifications
Action View
\begin{schema}{AddCopyReport}
  \xi Library\
  b?: BOOK\
  rep!: Report
\ST
  b? \mem \ran stock \imp rep! = FurtherCopyAdded\
  b? \mem \ran stock \imp rep! = NewTitleAdded
\end{schema}

\begin{schema}{AddCopy}
\Delta Library\
b? : BOOK\
rep! : Report
\ST
((\exists c: COPY | c \mem \dom stock \dot (stock' = stock \fvar \xi c \imap b? \xi) \land shelved' = shelved \uni \xi c \xi)) \land
issued' = issued \land
readers' = readers)
\land
(stock' = stock \land
issued' = issued \land
shelved' = shelved \land
readers' = readers) \land

```

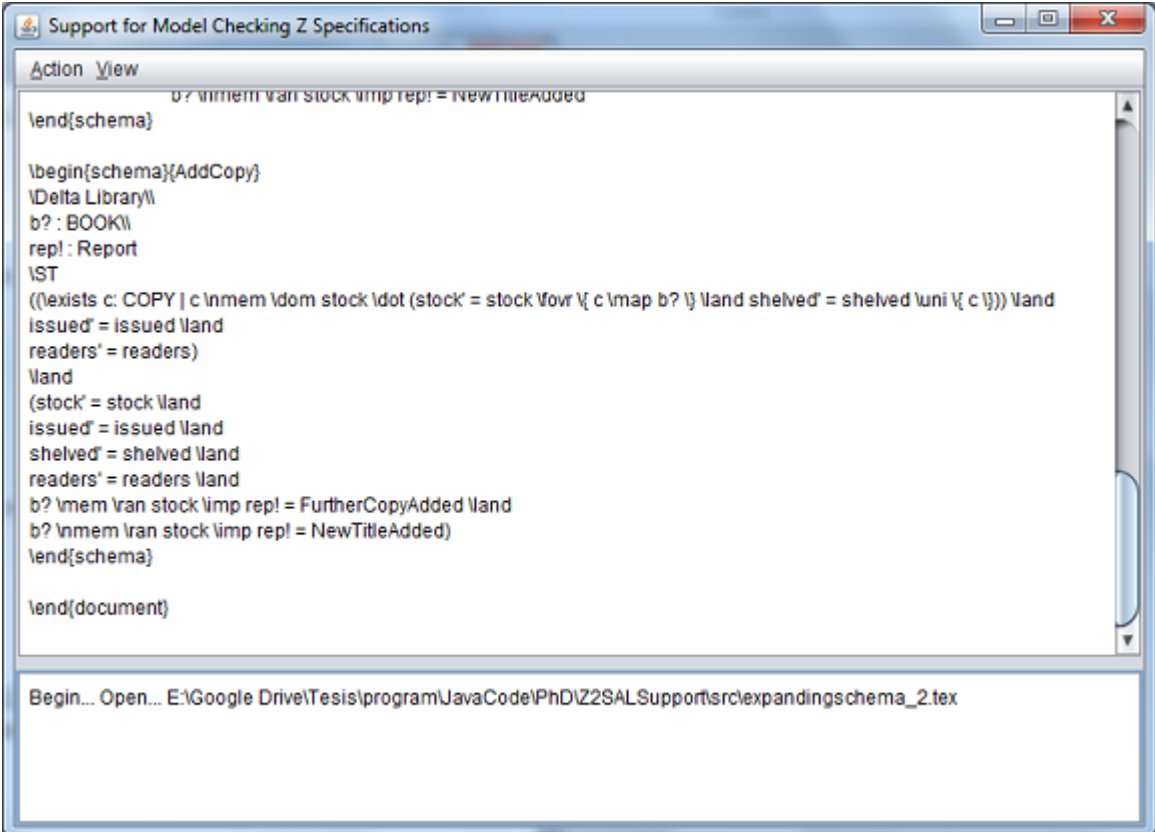
Begin... Open... E:\Google Drive\Tesis\program\JavaCode\PhD\Z2SALSupport\src\expanding_schema_2.tex

Figure 6.3: Finalising expansion process (continued)

language nor a complex schema calculus definition is supported by our system.

Based on experiments with this system, it seems that this constructed schema can be implemented straightforward, but not for the one that requires a heavy simplification on their predicates. In this case, it requires a more elaborate work to accomplish such a simplification process. This problem can be set up as future work. Nevertheless, our system can perform a simplification process, though it is a simple one. This success is expected as our contribution to related research in this field. The expansion schema calculus is a part of the architecture of our research as shown by Fig. 3.1 on page 77.

Our experiments on this system will be discussed on a later chapter.



```
Support for Model Checking Z Specifications
Action View
b? \mem \ran stock \imp rep! = NewTitleAdded
\end{schema}

\begin{schema}{AddCopy}
\Delta Library\
b? : BOOK\
rep! : Report
\ST
((\exists c: COPY | c \mem \dom stock \dot (stock' = stock \forall c \map b? \}) \land shelved' = shelved \uni \{ c \})) \land
issued' = issued \land
readers' = readers)
\land
(stock' = stock \land
issued' = issued \land
shelved' = shelved \land
readers' = readers \land
b? \mem \ran stock \imp rep! = FurtherCopyAdded \land
b? \mem \ran stock \imp rep! = NewTitleAdded)
\end{schema}

\end{document}

Begin... Open... E:\Google Drive\Tesis\programJavaCode\PhD\Z2SALSupport\src\expanding_schema_2.tex
```

Figure 6.4: Finalising expansion process (continued)

Chapter 7

Integration among the Scanner, the Parser, and Java Programs

This chapter describes our method to integrate our separate systems as discussed in the earlier chapters. For this purpose, another Java program has been built. This Java program which is named `Z_Preprocessing_Tool.java` is our main program.

By starting from this main program, our support for model checking Z specifications runs. The next discussion relates to this main program.

7.1 Java Main Program

Our main program has a simple GUI form as shown in Fig. 7.1 on page 172. As can be seen in Fig. 7.1 on page 172, there are two menus.

The first menu, **Action**, consists of two sub-menus: **Redefine**, and **Expand**. Based on their names, the first sub-menu performs a redefinition of generic constants, whereas the second one performs an expansion on schema calculus definitions.

The second menu, **View**, is just to display a result of a successful action. The result is a generated Z specification which either its generic constant definition has been redefined to an equivalent axiomatic definition or its schema calculus definition has been expanded to a new schema. Thus, this menu contains two sub-menus: **Redefinition**, and **Expansion**. It is either one of these menus which is enabled automatically by our system depending on a previous action.

As the main program, redefinition of a generic constant definition, and expansion of a schema calculus definition were designed and implemented in different files from the scanner and parser, all these systems should be

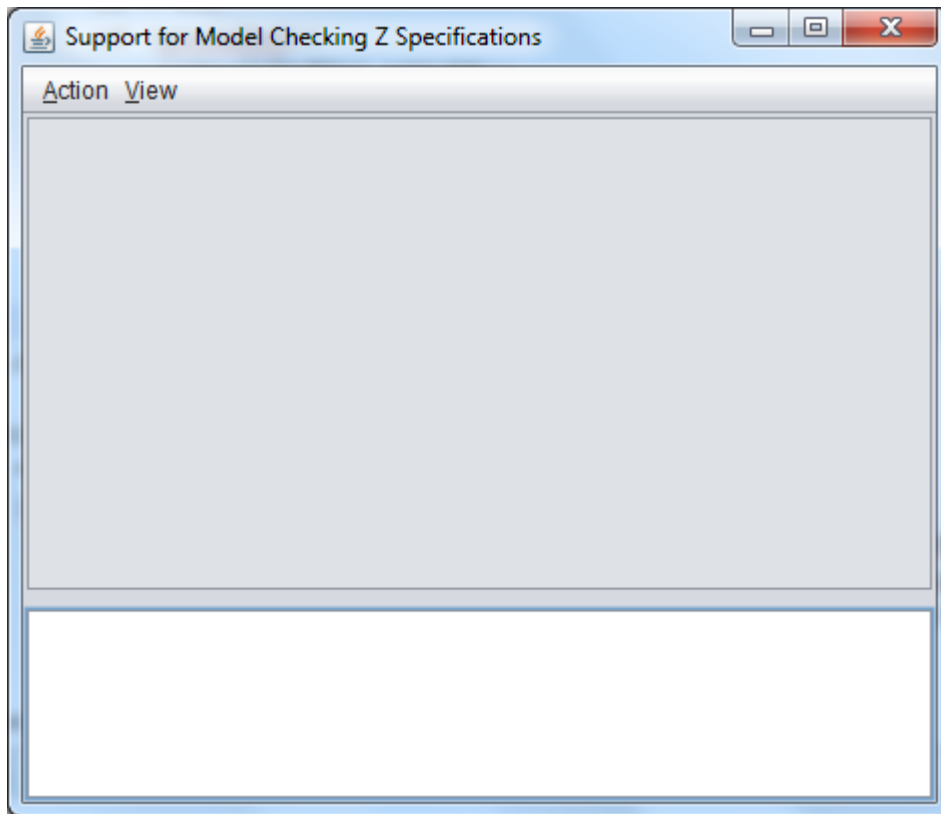


Figure 7.1: Support for Model Checking Z Specifications

integrated that they can connect to each other. This connection begins right after a Z specification file is read by the system.

Let us discuss a method to read a Z specification implemented in our system.

7.2 Reading a Z Specification

During a read process, characters of such a Z specification input will be checked line by line to get necessary information.

```
if (s.startsWith("\\begin{zed}")){
```

The above code is the first check. It will check the existence of a zed box definition. The zed box definition is classified as a definition without any box [69]. The first zed definition specified in our system is a generic definition which is indicated by the next line as follows:

```
if ((s.contains("\\defs") || s.contains("==")) && s.contains("X")){
```

If it is a case, this definition will be changed to a generic constant definition box. The first change is to replace "`\begin{zed}`" with "`\begin{gendef}`" as follows:

```
generic.tempS.add("\begin{gendef}{X}");
```

`tempS` is a variable to store bytes of characters during a read. It is a linked list typed variable which is an instance of a Java class `LinkedList`.

There are two further checks here. The first check is to see if a line is just a value as follows:

```
if (!s.contains("\defs")) generic.tempS.add(s);
```

In this case, the process is just to store the line to the list. The second one is to collect information of a variable name and its value. Our system just considers a value which begins with "`\{`" as follows:

```
if (val.startsWith("\{")){
```

Afterwards, variables, and their constraints are recorded. A bar, which separates a declaration part, and a predicate part, is checked then. The bar can be a "`|`" or a "`\cbar`".

If a "`•`" begins a predicate part, further checks are required. After these checks, obtained information will be processed. Our system just supports one generic parameter in a generic definition mentioned earlier, which is `X`.

```
if (generic.tempS.toString().contains(tempV)){
  for (int numList=0;numList<generic.tempS.size();numList++){
    tempVN = generic.tempS.get(numList).toString();
    if (tempVN.contains(tempV)&&(tempVN.indexOf(tempV) < tempVN.indexOf(":"))){
      if (tempVN.substring(tempVN.indexOf(tempV) + tempV.length()).
        startsWith("_") || tempVN.substring(tempVN.indexOf(tempV) +
        tempV.length()).startsWith("")){
        newVal = tempVN.substring(tempVN.indexOf(":")+1).trim();
        generic.tempS.add(var+"_:"+newVal);
        break;
      }
    }
  }
}
else generic.tempS.add(var+"_:"+"\pset_"+newVal);
```

The above code is to avoid any duplication of generic constant definitions. After a declaration of an associated generic constant is obtained, the next process is to get this generic constant predicate part as follows:

```
if (!val.endsWith("\ "+"")){
  generic.tempS.add("\ST");
  pred = var+"_="+val;
  generic.tempS.add(pred);
}
else{
  pred = var+"_="+val;
  generic.tempS.add(pred);
}
```

As can be seen from the above code, our system supports only a predicate part which is in a form of equality. All above processes will be repeated until a "\end{zed}" is read. If it is found, an associated line will be changed to "\end{gendef}" as follows:

```
if (s.equals("\\end{zed}"))
    generic.tempS.add("\\end{gendef}");
```

If the next line is not a generic definition, "\begin{zed}" is kept unchanged, the next process is to just store the number of line which has been read to the list; it is an ordinary definition which will not be processed further here.

The second check is to see whether "\lambda" expression exists in line by line of the Z specification as follows:

```
else if (s.contains("\\lambda") && !s.contains("==") && !s.contains("\\defs")){
```

Thus, this lambda should not a part of a definition. If this lambda expression is written on more than one line, these separate lines will be joined as follows:

```
if (s.trim().endsWith("\\dot") || s.trim().
endsWith("\\spot") || s.trim().endsWith("\\cbar")){
    ss = s;
    while (!s.endsWith("\\"+ "\\") || !s.trim().equals("\\end{axdef}") ||
    !s.trim().equals("\\end{gendef}") || !s.trim().equals("\\end{schema}")){
        s = br.readLine();
        if (!s.trim().equals("\\end{axdef}")){
            ss = ss + "\_" + s;
        }
        else if (!s.trim().equals("\\end{gendef}")){
            ss = ss + "\_" + s;
        }
        else if (!s.trim().equals("\\end{schema}")){
            ss = ss + "\_" + s;
        }
    }
}
```

The joined lines will be used afterwards.

Our system supports just one lambda expression in a predicate line; there is no nested lambda expression, as follows:

```
if ((s.indexOf("\\lambda") > s.indexOf("(") &&
s.indexOf("\\lambda") < s.indexOf(")")) ||
(s.indexOf("\\lambda") > s.indexOf("\\limg") &&
s.indexOf("\\lambda") < s.indexOf("\\ring"))){
```

From this lambda expression, variables and their types will be gathered, as well as predicates as follows:

```
idxLambd = s.indexOf("\\lambda");
if (s.contains("\\dot")){
    var = s.substring(idxLambd+7,s.indexOf("\\dot", idxLambd)).trim();

    if (s.indexOf("\\lambda", idxLambd+1) > -1){
        result = s.substring(s.indexOf("\\dot",idxLambd)+4,
```



```

                s.lastIndexOf(".") , s.indexOf("\\lambda" , idxLambd+1))). trim ();
    }
    else result = s.substring(s.indexOf("\\dot" ,
        idxLambd)+4, s.lastIndexOf("." )). trim ();
    newVar = "\\forall " + var + " \\dot ";
}
else {
    var = s.substring(idxLambd+7, s.indexOf("\\spot" , idxLambd)). trim ();

    if (s.indexOf("\\lambda" , idxLambd+1) > -1){
        ...
    }
    else ...
    ...
}
}

```

".." means the same as previous code, but this time it is for a string "\begin{spot}", not a string "\begin{dot}". In a case there are more declared variables, further processes are required as follows:

```

if (!var.contains(";")){
    variables = var.substring(0, var.indexOf(":")). trim ();
}
else{
    idxS = 0;
    do{
        if (!variables.isEmpty()){
            variables = var.substring(idxS, var.indexOf(":" , idxS)) + "," + variables;
        }
        else{
            variables = var.substring(idxS, var.indexOf(":" , idxS));
        }
        if (var.indexOf(";" , idxS) > -1){
            idxS = var.indexOf(";" , idxS)+1;
        }
        else{
            idxS = var.length();
        }
    }
    while (idxS < var.length());
}
}

```

After that, an operator that relates a generic constant to its lambda expression is checked. There are several operators specified here: "=", "∈", "≠", "∉", "⊆", "⊂", "<", "≤", ">", "≥", as follows:

```

idxS = s.lastIndexOf("(", idxLambd);
lhsVar = s.substring(0, idxS). trim ();
if (lhsVar.endsWith("=") || lhsVar.endsWith("\\mem") ||
    lhsVar.endsWith("\\in") || lhsVar.endsWith("\\neq") ||
    lhsVar.endsWith("\\nem") || lhsVar.endsWith("\\nmem") ||
    lhsVar.endsWith("\\subsetq") || lhsVar.endsWith("subset") ||
    lhsVar.endsWith("\\subs") || lhsVar.endsWith("\\psubs") ||
    lhsVar.endsWith("<") || lhsVar.endsWith("\\leq") ||
    lhsVar.endsWith(">") || lhsVar.endsWith("\\geq")){
    stVar = new StringTokenizer(lhsVar, "(");
    do{
        llVarGC.add(stVar.nextElement(). toString ());
    }
}

```

```
while (stVar.hasMoreElements());
```

The line containing a lambda expression will be changed into a new predicate line which begins with a universal quantifier as follows:

```
if (!llVarGC.getLast().toString().trim().equals("=")
&& !llVarGC.getLast().toString().trim().equals("\\mem")
&& !llVarGC.getLast().toString().trim().equals("\\in")
&& ...
&& !llVarGC.getLast().toString().trim().equals("\\geq")){
  lhsVar = llVarGC.getLast().toString().trim();
  if (lhsVar.endsWith("=")){
    llVarGC.set(llVarGC.size()-1,lhsVar.substring(lhsVar.indexOf("=")).trim());
    lhsVar = lhsVar.substring(0,lhsVar.indexOf("=")).trim();
  }
  else if (lhsVar.endsWith("<")){
    ...
  }
  else if (lhsVar.endsWith(">")){
    ...
  }
  else {
    llVarGC.set(llVarGC.size()-1,lhsVar.substring(lhsVar.
lastIndexOf("\\"+"\\").trim());
    lhsVar = lhsVar.substring(0,lhsVar.lastIndexOf("\\"+"\\").trim());
  }
  if (variables.contains(",")){
    variables = "(" + variables + ")";
  }
  else variables = "_" + variables;
  lhsVar = lhsVar + variables;
}
else{
  if (variables.contains(",")){
    variables = "(" + variables + ")";
  }
  else variables = "_" + variables;
  lhsVar = llVarGC.get(llVarGC.size()-2).toString() + variables;
}

newVar = newVar + lhsVar + "_" + llVarGC.getLast().toString() + "_" + result;
generic.tempS.add(newVar);
}
```

where "...s are the same as earlier associated lines for the rest of operators. The new predicate is **newVar** on the above code.

The third check is on neither a generic constant definition on a zed box nor a lambda expression contained in a line. For this case, there is no further process; the only process just stores the line to the list.

These three checks are repeated until end of file. Afterwards, a new file will be created and its name is the same as the input file name, but it starts with "output_". Contents of the **tempS** list will then be copied to this new file.

The next section discusses how a connection amongst all systems was built. It occurs after a *Z* specification is read as specified in the earlier code.

7.3 Establishing a Connection Amongst Systems

The read file, which might be pre-processed as mentioned earlier, will be passed to the parser to be processed further. The code below shows that process:

```
Parser p = new Parser(new FileReader(file.getParent()+"\\output_"+file.getName()));
p.out.flush();
p.yyparse();
```

Parser is a name which was specified for our parser. The above code shows how our main program connects to our parser. The code was specified in our main program.

Below is code to connect our parser to our scanner. It is specified in our parser shown as follows:

```
public Parser (Reader r) {
    lexer = new ScannerCl(r, this);
}
```

The above function is the only constructor in our parser which requires one parameter whose type is **Reader**. **Reader** is one class in Java. **ScannerCl** is a name of a Java class specified in our scanner that will be created when the scanner runs. It requires two parameters on its constructor shown as follows:

```
public ScannerCl(java.io.Reader r, Parser yyparser) {
    this(r);
    this.yyparser = yyparser;
}
```

The above constructor was specified in our scanner.

lexer is a variable which is an instance of **ScannerCl**. **yyparse** is a name of a method provided by BYACC/J which is an entry point to a yacc-generated parser [45].

Each time this method is called, an input stream will be parsed. It returns zero if the parse succeeds, otherwise it returns not-zero. A failure in parsing makes our system stop immediately. An error reporting this failure will be generated by our system.

The next process after a successful parsing can be a redefinition or a schema expansion which depends on what an option is chosen by a user. Both processes have been discussed on earlier chapters.

7.4 Conclusion

The method to interconnect our systems has been discussed above. It is a part of our research's architecture, which is shown by Fig. 3.1, though it was

written explicitly in that picture.

The connection starts with an execution in the main Java program. This program then connects to the parser. Subsequently the parser connects to the scanner. The scanner provides a sequence of accepted tokens to the parser. The parser checks whether this sequence matches the specified grammar. It also stores necessary information for redefinition or expansion system. Process of either redefinition or expansion is performed afterwards. Alongside each of this process, but in our experiments it is usually in the redefinition process, user-defined function modification could also be accomplished. It is because our examples for schema expansion are rare specified in having user-defined function. The modification is in the SAL file generated by Z2SAL.

By implementing all these programs separately, our system is more modular. Thus, it is easy to perform a further development. It is easy also to locate an error.

As mentioned earlier, our main program has a simple display. To inform a user about processes on each program, it might be better if this system can display line by line of a Z specification input which is accompanied by what process occurs in it. Furthermore, our system only supports a read in one file input of a Z specification in each execution of either redefinition or expansion menu. All of these improvements can be considered as future works.

Chapter 8

A Generic Constants Redefinition

This chapter discusses further our redefinition system, which is a part of our research's architecture as can be seen in Fig. 3.1. It consists of four sections: setting up questions for evaluation of this system, several experiments with this system, evaluation of this system, and conclusion.

The first section describes a few questions which are required on an evaluation stage. It will be followed by a description of our experiments on this system and a discussion on each experiment. There are 23 experiments conducted by us on this system. The next section is an evaluation on this system by answering earlier questions. A brief conclusion will end this chapter.

8.1 Setting up Questions for Evaluation

Evaluation means an action to find out over a system whether the system can perform specified aims. As mentioned earlier in Section 1.2 on page 15, our objective is to implement a tool which will redefine a generic constant definition to an equivalent axiomatic definition. As a result, this tool or system has been implemented with several limitations.

Over this system, questions are set up to assess its performances. These questions are as follows:

- Can this system redefine all generic constant definitions correctly?
- Can the outcome of this system always be translated by Z2SAL and then be executed by the SAL tool?
- Does the approach scale to larger specifications?

To obtain solutions for these questions, several experiments with this system have been conducted. It will be discussed on the following section.

8.2 Experiments with the Generic Constant Redefinition

Several of our experiments with these systems are given in this section. It also includes our proposed method on a new SAL translation of function or constant in associated experiments. Results and discussions relating to these experiments are given also in each sub-section. Thus, our method of a redefinition of generic constants as well as our proposed method can be followed in this section.

A list of experiments on this system were summarised in the table in [63]. There were a few experiments added to this table. All these experiments are as shown by Table 8.1 on page 181.

Before details of each experiment are discussed, a brief description of our Z specifications is given in Table 8.2 on page 182, where the below abbreviation words mean:

GCD = Generic Constant Definition
gp = generic parameter.

This table is arranged such that experiments which have the less number of generic constant definitions are listed at the top. Afterwards, experiments are ordered based on their number of usages, then number of generic parameters on the same type of generic constants.

Referring to this table, one Z specification input of our experiments can contain several generic constant definitions as well as several usages. Furthermore, our experiments also experienced of function, relation and constant as types of generic constants. Up to two generic parameters were specified in our Z specifications though our system was designed to cover at most three generic parameters. Moreover, these generic parameters should be either one of X, Y or Z.

The generic constant definition is getting more complex if a declaration of such a generic constant uses not only generic parameters but also Z tags, which are accepted in a declaration of variables in Z, such as: "P", "seq", "×", "↔", functions, brackets and others. These Z tags in turn can be used in together, to create declarations such as: "(X ↔ Y) P X", "X ↔ P X", "X × Y → X × Y", "(P X) × (P X) → (P X)", "P X", "X × Y → X", "(seq₁

Table 8.1: Several Experiments with the Redefinition System

Z Specification (* .tex)	Details	Verification time in secs	
		#Theorem = 0	#Theorem > 0
bbook	Modified SAL function		0.842
bbook_map	Modified SAL function	0.016	0.25
bbook_uni	Modified SAL function and other parts of SAL file	0.031	0.406
bbook_map_uni	Modified SAL function and other parts of SAL file		0.359
fDomRan	Modified SAL function	0.015	
fEmpty	OK		0.093
fEmptyImpl	OK		0.109
fFirst	Modified SAL function	0.015	0.187
fHead	Modified SAL function	0.031	
fHeadFunc	Modified SAL function and cannot be simulated: The set of initial states is empty	0.031	
fMaxComSubSeq	Modified other parts of SAL file and cannot be simulated: An out of memory error	0.047	
fMaxComSubSeq_1	Modified other parts of SAL file and cannot be simulated: An out of memory error	0.032	
fMaxComSubSeq_orig	Modified other parts of SAL file and cannot be simulated: An out of memory error	0.032	
fMonoSeq	OK. Long simulation	0.047	
fMonoSeq_1	OK. Long simulation	0.031	
fSwap	Modified SAL function	0.016	0.141
fUniqSeq	Ok. Cannot be simulated: An out of memory error	0.062	
fUniq1Seq	Ok. Cannot be simulated: An out of memory error	0.031	
fUniq2Seq	Ok. Cannot be simulated: An out of memory error	0.015	
tn	Modified other parts of SAL file and cannot be simulated: An out of memory error	0.03	
tnImpl	Modified other parts of SAL file and cannot be simulated: An out of memory error	0.0	
fFileStorage	Cannot be translated by Z2SAL	N/A	
fSet	Modified SAL function and other parts of SAL file	0.0	

Table 8.2: Summaries of Experiments

Number of GCDs	Z Specification (*.tex)	Details
1	bbook_uni (9.2.3 on page 228)	1 function, 1 gp, 1 usage
	fHeadFunc (9.2.10 on page 235)	1 function, 1 gp, 2 usages
	fUniq1Seq (9.2.18 on page 240)	1 function, 1 gp, 2 usages
	fUniq2Seq (9.2.19 on page 240)	1 function, 1 gp, 2 usages
	fHead (9.2.9 on page 234)	1 function, 1 gp, 3 usages
	bbook_map (9.2.2 on page 227)	1 function, 2 gps, 1 usage
	fEmpty (9.2.6 on page 232)	1 constant, 1 gp, 2 usages
	fEmptyImpl (9.2.7 on page 233)	1 constant, 1 gp, 2 usages
	tn (9.2.20 on page 241)	1 constant, 1 gp, 6 usages
	tnImpl (9.2.21 on page 241)	1 constant, 1 gp, 6 usages
2	bbook (9.2.1 on page 224)	1 relation, 1 gp, 1 usage
		1 function, 2 gps, 2 usages
	bbook_map_uni (9.2.4 on page 229)	1 function, 1 gp, 1 usage
		1 function, 2 gps, 1 usage
	fDomRan (9.2.5 on page 231)	1 function, 2 gps, 1 usage
		1 function, 2 gps, 1 usage
	fFileStorage (9.2.22 on page 242)	1 function, 1 gp, 1 usage
		1 function, 1 gp, 1 usage
	fFirst (9.2.8 on page 233)	1 function, 2 gps, 2 usages
		1 function, 2 gps, 2 usages
	fMonoSeq (9.2.15 on page 238)	1 constant, 1 gp, 1 usage
		1 constant, 1 gp, 2 usages
	fMonoSeq_1 (9.2.14 on page 238)	1 constant, 1 gp, 1 usage
		1 constant, 1 gp, 2 usages
fSwap (9.2.16 on page 239)	1 function, 1 gp, 1 usage	
	1 function, 2 gps, 1 usage	
3	fUniqSeq (9.2.17 on page 239)	1 constant, 1 gp, 3 usage
		1 constant, 1 gp, 3 usages
		1 function, 1 gp, 2 usages
4	fMaxComSubSeq (9.2.13 on page 237)	1 relation, 1 gp, 1 usage
		1 function, 1 gp, 1 usage
		1 function, 1 gp, 2 usages
		1 function, 1 gp, 2 usages
	fMaxComSubSeq_1 (9.2.12 on page 236)	1 relation, 1 gp, 1 usage
		1 function, 1 gp, 1 usage
		1 function, 1 gp, 2 usages
		1 function, 1 gp, 2 usages
	fMaxComSubSeq_orig (9.2.11 on page 236)	1 relation, 1 gp, 1 usage
		1 function, 1 gp, 1 usage
		1 function, 1 gp, 2 usages
		1 function, 1 gp, 2 usages
	fSet (9.2.23 on page 242)	1 relation, 1 gp, 1 usages
		1 relation, 1 gp, 3 usages
1 function, 1 gp, 1 usage		
1 constant, 1 gp, 7 usages		

$X \rightarrow X$ ", " $(\text{seq } X) \leftrightarrow (\text{seq } X)$ ", " $(\text{seq } X) \rightarrow \mathbb{P}(\text{seq } X)$ ", " $((\text{seq } X) \times (\text{seq } X)) \rightarrow \mathbb{P}(\text{seq } X)$ ", " $\mathbb{P}(\text{seq } X)$ ", " $X \times Y \rightarrow Y \times X$ ", " $X \times X \rightarrow X \times X$ ", " $(\text{seq } X) \rightarrow \text{seq } X$ ", " $(\text{seq } X) \rightarrow (\text{seq } X)$ ", " $\mathbb{P}(\mathbb{P}(\mathbb{P } X))$ ", " $((\text{seq } X) \times \mathbb{N}) \rightarrow (\text{seq } X)$ ", " $((\text{seq } X) \times \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow X)$ ", " $X \leftrightarrow X$ ". These cases require elaborate work on type unification.

Details of each experiment are given in the following sub-sections. These experiments are in alphabetical order, except the last two experiments. Both these experiments have problems during their running. Having our system fixed, the last experiment could be redefined by our system. However, another experiment has a problem with Z2SAL as discussed later.

Thus, our experiments as follows show how a generic constant is redefined correctly based on its usage. In addition, our experiments also show how when this generic constant is used for several usages it is still redefined correctly.

Following experiments also demonstrate how our method to solve problems that occurred during processes. These problems can either be in a redefinition process, a translation by Z2SAL or an execution by the SAL tool. The last two problems also resemble our proposed method on SAL function.

8.2.1 Experiment 1: bbook.tex

This specification was taken from [69, p. 3-6], but has been modified in several places. There are two generic constants definitions specified in this specification. The first generic constant definition is as follows:

$$\begin{array}{|l} \hline [X, Y] \\ \hline \text{domain} : (X \rightarrow Y) \rightarrow \mathbb{P} X \\ \hline \forall R : X \rightarrow Y \bullet \text{domain} R = \{x : X \mid \exists y : Y \bullet x \mapsto y \in R\} \\ \hline \end{array}$$

This definition has one generic constant, `domain`, which type is a function, specifically a total function. The `domain` generic constant has two generic parameters: `X`, and `Y`. The input to this generic constant is a partial function from `X` to `Y`. The output of this generic constant is a set of `X`.

Another generic constant definition is shown as follows:

$$\begin{array}{|l} \hline [X] \\ \hline \text{nonMember} : X \leftrightarrow \mathbb{P} X \\ \hline \forall x : X; S : \mathbb{P} X \bullet (x, S) \in \text{nonMember} \Leftrightarrow \neg (x \in S) \\ \hline \end{array}$$

This generic constant, `nonMember`, is a relation between `X` and a set of `X`. As can be seen from above definition, this generic constant was specified with one generic parameter, `X`.

These generic constants were used on several places on schemas of this specification. The first usage is as follows:

$$(birthday, known) \in domain$$

This is a usage of `domain` and it was written by using a membership operator, "∈". The first argument, which is an input, is `birthday` which type is a partial function from `NAME` to `DATE`. The second argument, `known`, is an output and its type is a set of `NAME`.

The second usage is given by the predicate:

$$houseKnown = domain(house)$$

At this time, the usage was written as a normal writing of a function. The input is `house` which type is a partial function from `NAME` to `ADDRESS`. Obviously, the output is `houseKnown` which type is a set of `NAME`.

The next usage is to the second generic constant.

$$(name?, known) \in nonMember$$

It was written by using a common written of a relation. It has two parameters: `name?` which type is an instance of `NAME`, and `known` is a set of `NAME`.

Another usage is as follows:

$$(name?, houseKnown) \in nonMember$$

Types for both parameters can be obtained from previous descriptions.

Thus, there were four usages for generic constant specified in this specification. After this specification was run by our system, three axiomatic definitions redefined previous two generic constant definitions.

The first usage as discussed above was redefined as follows:

$$\left| \begin{array}{l} domain : (NAME \rightarrow DATE) \rightarrow \mathbb{P} NAME \\ \forall R : NAME \rightarrow DATE \bullet domainR = \{x : NAME \mid \exists y : DATE \bullet x \mapsto y \in R\} \end{array} \right.$$

Thus, X is actualized to `NAME`, and Y is actualized to `DATE`.

The second usage is to the same generic constant as the first usage. However, their arguments have different types. Thus, this usage was redefined to a different axiomatic definition as shown below:

$$\left| \begin{array}{l} domain1 : (NAME \rightarrow ADDRESS) \rightarrow \mathbb{P} NAME \\ \forall R : NAME \rightarrow ADDRESS \bullet domain1R = \{x : NAME \mid \exists y : ADDRESS \bullet x \mapsto y \in R\} \end{array} \right.$$

The type for X is the same as the type of X from `domain`, but Y on `domain1` is `ADDRESS` which is different with Y from `domain`. The additional index, 1, at

the end of `domain` affects also the way the usage was written. The updated associated line of predicate is as follows:

$$houseKnown = domain1(house)$$

The last two usages have the same generic constant and the same types of their arguments. Thus, both usages are redefined to the same axiomatic definition, as follows:

$$\frac{nonMember : NAME \leftrightarrow \mathbb{P} NAME}{\forall x : NAME; S : \mathbb{P} NAME \bullet (x, S) \in nonMember \Leftrightarrow \neg (x \in S)}$$

The SAL file which was generated by Z2SAL could not be verified by the SAL model checker. The error as shown below was produced instead by the SAL model checker:

```
Error: [Context: output_bbook, line(50),
column(12)]: Incompatible types in the equality operator.
The following types are incompatible:
output_bbook!Set_B_NAME
[output_bbook!NAME -> bool]
```

Several lines of the SAL specification, including the associated line, are shown as follows:

```
49 (FORALL (q_1 : Set_C_NAME_X_B_DATE_I) :
50 domain(q_1) =
51 {q_2 : NAME | (EXISTS (q_3 : DATE) : q_1(q_2) =
52 q_3)}) AND
```

The type of `domain` is `[Set_C_NAME_X_B_DATE_I -> Set_B_NAME]`. Thus, the type of `domain(q_1)` is `Set_B_NAME`. However, the right-hand side of the equality operator is `NAME -> bool`.

Based on this error, the translations of the `domain` and `domain1` function are modified manually on the SAL file. These are given as follows:

```
domain(q_1 : Set_C_NAME_X_B_DATE_I): Set_NAME =
{q_2 : NAME | (EXISTS (q_3 : DATE) : q_1(q_2) = q_3)};

domain1(q_4 : Set_C_NAME_X_B_ADDRESS_I): Set_NAME =
{q_5 : NAME | (EXISTS (q_6 : ADDRESS) : q_4(q_5) = q_6)};
```

These function translations were put on the context clause.

There were two theorems formulated on this SAL file as shown below:

```
th1 : THEOREM State |- G(function {NAME, B_DATE; DATE_BB; }!empty?(birthday));
th2 : THEOREM State |- G(function {NAME, B_ADDRESS; ADDRESS_BB; }!empty?(house));
```

This SAL file could be verified by the SAL model checker after the modification. Total execution time is 0.857 second. Both theorems were invalid. The SAL file could also be simulated by the SAL simulator.

8.2.2 Experiment 2: `bbook_map.tex`

This specification originated from the same book as the first experiment. There is only one generic constant specified in this specification, `mapRel`. It is shown as follows:

$$\frac{}{[X, Y] \frac{}{mapRel : X \times Y \rightarrow X \times Y}}{\forall x : X; y : Y \bullet mapRel(x, y) = (x, y)}$$

This generic constant is a total function from a pair of X, and Y. There are two generic parameters: X, and Y.

Only one usage was specified upon this generic constant.

$$birthday' = birthday \cup \{mapRel(name?, date?)\}$$

`date?` has a type of DATE. Following axiomatic definition was generated to redefine above generic constant definition:

$$\frac{}{mapRel : NAME \times DATE \rightarrow NAME \times DATE}}{\forall x : NAME; y : DATE \bullet mapRel(x, y) = (x, y)}$$

The SAL file generated by Z2SAL could be verified by the SAL model checker. However, after theorems, which are the same as theorems on the previous experiment, were added to this SAL file, this SAL file could not be verified by the SAL model checker.

Then, the SAL translation for the `mapRel` function was modified manually and was put on the context clause instead.

$$mapRel(q_{-1} : NAME, q_{-2} : DATE) : NAME_X_DATE = (q_{-1}, q_{-2});$$

Total execution time by the SAL model checker is 0.25 second. This SAL file could also be simulated by the SAL simulator.

8.2.3 Experiment 3: `bbook_uni.tex`

This specification comes from the same book as two previous experiments. The only generic constant specified in this specification is as follows:

$$\frac{}{[X] \frac{}{uniSet : (\mathbb{P} X) \times (\mathbb{P} X) \rightarrow (\mathbb{P} X)}}{\forall S, T : (\mathbb{P} X) \bullet uniSet(S, T) = \{x : X \mid x \in S \vee x \in T\}}$$

It has one generic parameter, X , and it is a total function from a pair of a set of X , and a set of X , to a set of X .

The usage of this generic constant is as follows:

$$\text{birthday}' = \text{uniSet}(\text{birthday}, \{ \text{name?} \mapsto \text{date?} \})$$

As given in the previous experiments, the type of `birthday` is a partial function from `NAME` to `DATE`.

Based on [69], mathematical functions are special kind of relation. Thus, a function can be specified as a relation with additional conditions to assert its function behaviour. Furthermore, a relation of two objects is a synonym for a set of a pair of those objects.

Applying these synonyms, `birthday` can be written as a special relation, then a set of a pair of `NAME`, and `DATE`. Thus, following axiomatic definition was generated by our system for above usage on this generic constant:

$$\frac{\text{uniSet} : (\mathbb{P}(\text{NAME} \times \text{DATE})) \times (\mathbb{P}(\text{NAME} \times \text{DATE})) \rightarrow (\mathbb{P}(\text{NAME} \times \text{DATE}))}{\forall S, T : (\mathbb{P}(\text{NAME} \times \text{DATE})) \bullet \text{uniSet}(S, T) = \{x : (\text{NAME} \times \text{DATE}) \mid x \in S \vee x \in T\}}$$

However, the SAL file generated by Z2SAL could not be verified by the SAL model checker. An error was produced instead:

```
Error: [Context: output_bbook_uni, line(47), column(40)]:
Incompatible types in the equality operator.
The following types are incompatible:
output_bbook_uni!Set_C_B_NAME_X_B_DATE_I

[output_bbook_uni!NAME_X_DATE -> bool]
```

Associated SAL lines are as follows:

```
46 (FORALL (q--1 : Set_C_NAME_X_DATE_I, q--2 :
47 Set_C_NAME_X_DATE_I) : uniSet((q--1, q--2)) =
48 {q--3 : NAME_X_DATE | set {NAME_X_DATE;} !
49 contains?(q--1, q--3) OR
50 set {NAME_X_DATE;} ! contains?(q--2, q--3)}) AND
```

This error said that there was incompatible type between the output of `uniSet`, which was `Set_C_B_NAME_X_B_DATE_I`, and the right hand side of the equality operator, which was `[NAME_X_DATE -> bool]`.

Then, modification was made as follows to those SAL lines:

```
46 (FORALL (q--1 : set {NAME_X_DATE;} ! Set, q--2 :
47 set {NAME_X_DATE;} ! Set) : uniSet((q--1, q--2)) =
48 {q--3 : B_NAME_X_B_DATE | set {B_NAME_X_B_DATE;} !
49 contains?(q--1, q--3) OR
50 set {B_NAME_X_B_DATE;} ! contains?(q--2, q--3)}) AND
```

However, there was another error as follows:

```
Error: [Context: output_bbook_uni, line(49), column(21)]:
Type mismatch in the function application.
Expected type:
[set{output_bbook_uni!B_NAME_X_B_DATE}!Set,
output_bbook_uni!B_NAME_X_B_DATE]
Actual type:
[set{output_bbook_uni!NAME_X_DATE}!Set,
output_bbook_uni!B_NAME_X_B_DATE]
```

Since the error relates to bottomed function, which might not be a problem if the `uniSet` function is specified in the context clause, a modification was performed. The modification declaration for `uniSet` is as follows:

```
uniSet(q--1 : set {NAME_X_DATE;} ! Set, q--2 : set {NAME_X_DATE;} ! Set):
set {NAME_X_DATE;} ! Set = {q--3 : NAME_X_DATE | set {NAME_X_DATE;} !
contains?(q--1, q--3) OR set {NAME_X_DATE;} ! contains?(q--2, q--3)};
```

However, an error was still produced as follows:

```
Error: [Context: output_bbook_uni_mod, line(62), column(29)]:
Type mismatch in the function application.
Expected type:
[set{output_bbook_uni_mod!NAME_X_DATE}!Set,
set{output_bbook_uni_mod!NAME_X_DATE}!Set]
Actual type:
[output_bbook_uni_mod!Set_C_NAME_X_B_DATE_I,
set{output_bbook_uni_mod!NAME_X_DATE}!Set]
```

The related SAL lines are as follows:

```
61 NOT set {NAME;} ! contains?(known, name?) AND
62 birthday' = uniSet((birthday, set {NAME_X_DATE;} !
63 singleton((name?, date?))) AND
64 invariant_-'
```

The type of `uniSet` after modification is a pair of `set {NAME_X_DATE;} ! Set` and `set {NAME_X_DATE;} ! Set` as can be seen in line 62. This type was not compatible with the actual type passed to `uniSet` which was a pair of `Set_C_NAME_X_B_DATE_I` and `set {NAME_X_DATE;} ! Set`. `Set_C_NAME_X_B_DATE_I` is an alias for `[NAME -> B_DATE]`, specified by Z2SAL. Although in the Z language, a function is special type of a relation, and a relation is a set of a pair of types, it seems that SAL does not assume the types of the first argument of `uniSet` are the same.

Thus, this incompatible type was solved manually. It is since our tool has not been able to do it automatically.

Our modification was to keep the same alias for `birthday`, but this alias represents a relation, not a function any more.

```
Set_C_NAME_X_B_DATE_I : TYPE = set {NAME_X_DATE;} ! Set;
```

This changing affects the usage of `birthday`; it cannot any longer be used as a function.

```
function {NAME, B_DATE; DATE_BB} ! partial?(birthday) AND
```

The above line was deleted.

```
known = relation {NAME, DATE;} ! domain(birthday) AND
```

The above line was replaced by a line as follows:

```
known = function {NAME, B_DATE; DATE_BB} ! domain(birthday) AND
```

As well as a line as follows:

```
date_' = birthday(name?) AND
```

was replaced by the below line:

```
set {NAME_X_DATE;} ! contains? (birthday, (name?, date_')) AND
```

The modified SAL version could be verified either by the SAL model checker. The total execution time was 0.031 second. This SAL file could be simulated by the SAL simulator.

The same theorems as added to previous experiment were added also to this SAL file. The first theorem has been modified to reflect a relation.

```
th1: THEOREM State |- G(set {NAME_X_DATE;} !empty?(birthday));
```

```
th2: THEOREM State |- G(function{NAME,B_ADDRESS;ADDRESS_BB;}!
empty?(house));
```

Both theorems were INVALID and total execution time was 0.406 second.

8.2.4 Experiment 4: `bbook_map_uni.tex`

This specification combines two generic constant definitions which are specified in the `bbook.tex` and `bbook_map.tex`.

Two usages specified here are the same as found on those specifications.

$$birthday' = uniSet(birthday, \{mapRel(name?, date?)\})$$

Thus, this usage is a nested usage: there is a usage inside another usage.

There two axiomatic definitions generated by our system to redefine those two generic constant definitions as follows:

$$\frac{\text{mapRel} : \text{NAME} \times \text{DATE} \rightarrow \text{NAME} \times \text{DATE}}{\forall x : \text{NAME}; y : \text{DATE} \bullet \text{mapRel}(x, y) = (x, y)}$$

$$\frac{\text{uniSet} : (\mathbb{P}(\text{NAME} \times \text{DATE})) \times (\mathbb{P}(\text{NAME} \times \text{DATE})) \rightarrow (\mathbb{P}(\text{NAME} \times \text{DATE}))}{\forall S, T : (\mathbb{P}(\text{NAME} \times \text{DATE})) \bullet \text{uniSet}(S, T) = \{x : (\text{NAME} \times \text{DATE}) \mid x \in S \vee x \in T\}}$$

Modifications which were performed on **bbook_uni.tex** and **bbook_map.tex** were performed also in this specification. The same theorems as previous experiments were added to this SAL file.

This SAL file could be verified by the SAL model checker with 0.015 second and 0.359 second total execution times for no theorem and two theorems respectively. It could be simulated also by the SAL simulator.

8.2.5 Experiment 5: fDomRan.tex

This specification was taken from [57] and it has been altered in several places. There are two generic constants specified in one generic constant definition. This definition has two generic parameters: X, and Y.

The first generic constant is **domainSet**. It is a total function from a relation of X and Y, to a set of X.

Another generic constant is **rangeSet**. This generic constant has the same input as the first generic constant, but it is different on the output. At this generic constant, its output is a set of Y.

There are two usages; one for each of these generic constants.

$$\begin{aligned} (\text{occupies}, \text{current_guest}) &\in \text{domainSet} \\ (\text{occupies}, \text{occupied_room}) &\in \text{rangeSet} \end{aligned}$$

The type of **occupies** is a relation between **GUEST** and **HOTELROOM**. **current_guest** is a set of **GUEST**, **occupied_room** is a set of **HOTELROOM**.

Based on these usages, the following axiomatic definition was generated to redefine the above generic constants definition:

$$\frac{\begin{aligned} \text{domainSet} &: (\text{GUEST} \leftrightarrow \text{HOTELROOM}) \rightarrow \mathbb{P} \text{GUEST} \\ \text{rangeSet} &: (\text{GUEST} \leftrightarrow \text{HOTELROOM}) \rightarrow \mathbb{P} \text{HOTELROOM} \end{aligned}}{\begin{aligned} \forall R : \text{GUEST} \leftrightarrow \text{HOTELROOM} \bullet \\ \text{domainSet}R &= \{x : \text{GUEST} \mid \exists y : \text{HOTELROOM} \bullet x \mapsto y \in R\} \wedge \\ \text{rangeSet}(R) &= \{y : \text{HOTELROOM} \mid \exists x : \text{GUEST} \bullet x \mapsto y \in R\} \end{aligned}}$$

The generated SAL file could not be verified by the SAL model checker. The following error was produced:


```

Error: [Context: output_fDomRan, line(64), column(12)]:
Incompatible types in the equality operator.
The following types are incompatible:
output_fDomRan!Set_B_GUEST
[output_fDomRan!GUEST -> bool]

```

The SAL translations for these two generic constants were modified as follows:

```

domainSet(q--1 : Set_C_GUEST_X_HOTELROOM_I): Set_GUEST =
{q--2 : GUEST | (EXISTS (q--3 : HOTELROOM) :
set {GUEST_X_HOTELROOM;} ! contains?(q--1, (q--2, q--3)))};

rangeSet(q--1 : Set_C_GUEST_X_HOTELROOM_I): Set_HOTELROOM =
{q--4 : HOTELROOM | (EXISTS (q--5 : GUEST) :
set {GUEST_X_HOTELROOM;} ! contains?(q--1, (q--5, q--4)))};

```

Total execution time is 0.015 second. This SAL file was simulated also by the SAL simulator.

8.2.6 Experiment 6: fEmpty.tex

This specification, which has been added by us with a generic constant definition, was taken from [69, p. 81]. The type of generic constant specified here is a constant and it has one generic parameter: X .

$[X]$
$empty : \mathbb{P} X$
$empty = \{x : X \mid \text{false}\}$

There are two usages for this generic constant, as follows:

$$a! = empty[\mathbb{N}] \wedge b! = empty[\mathbb{Z}]$$

These usages, which are not available on the referenced book, were specified by using explicit type. Thus, obviously there will be two axiomatic definitions with the first X was actualised to " \mathbb{N} ", and the second one was to " \mathbb{Z} ".

$empty : \mathbb{P} \mathbb{N}$
$empty = \{x : \mathbb{N} \mid \text{false}\}$

$empty1 : \mathbb{P} \mathbb{Z}$
$empty1 = \{x : \mathbb{Z} \mid \text{false}\}$

The usages will be modified to:

$a! = \text{empty} \wedge b! = \text{empty1}$

$a!$ is a set of " \mathbb{N} ", and $b!$ is a set of " \mathbb{Z} ".

There was one theorem formulated on this SAL file:

th1: THEOREM State $\vdash G(\text{set}\{\text{INT};\}! \text{empty} = \text{empty1});$

This theorem was satisfied by the system. Total execution time is 0.094 second. It could be simulated by the SAL simulator.

8.2.7 Experiment 7: fEmptyImpl.tex

This specification is similar with the previous specification. The differences are on the explicit types; it was written implicitly.

$a! = \text{empty} \wedge b! = \text{empty}$

Our system is able to generate the same axiomatic definitions as previous experiment. As mentioned earlier, at this experiment, the actual types for output of each usage were inferred from these usages surrounding. Thus, the generated SAL file is also the same as the one from previous experiment.

8.2.8 Experiment 8: fFirst.tex

This specification was created by us. A generic constant definition which was taken from [69, p. 81] was added to this specification and another generic constant, `secondF`, was also created by us.

$\frac{[X, Y] \quad \begin{array}{l} \text{firstF} : X \times Y \rightarrow X \\ \text{secondF} : X \times Y \rightarrow Y \end{array}}{\forall x : X; y : Y \bullet \text{firstF}(x, y) = x \wedge \text{secondF}(x, y) = y}$
--

Both these generic constants are total functions from a pair of X and Y . The first generic constant has an output of type X , the second one has Y .

Several usages on these generic constants are as follows:

$b! = \text{secondF}(\text{number}, c?)$
 $a! = \text{firstF}(a?, b?)$
 $c! = \text{firstF}(c?, \text{number})$
 $d! = \text{secondF}(2, 4)$

All these usages were specified by us. `number` is a " \mathbb{N} " type state variable, whereas `c?` is a " \mathbb{N} " type non-state variable. `a?`, `b?` are `NAME` type variables.

Thus, the first usage and the third one are represented by the same axiomatic definition; the generic parameters are actualised by the same type,

” \mathbb{N} ”. On the other hand, the second usage make the generic parameters to be actualised to the same type also, **NAME**. The last usage is a numbered type, but it is a ” \mathbb{Z} ”. These all generated three axiomatic definitions that would be redefined the generic constant definition as follows:

$$\frac{\begin{array}{l} \text{firstF} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\ \text{secondF} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \end{array}}{\forall x : \mathbb{N}; y : \mathbb{N} \bullet \text{firstF}(x, y) = x \wedge \text{secondF}(x, y) = y}$$

$$\frac{\begin{array}{l} \text{firstF1} : \text{NAME} \times \text{NAME} \rightarrow \text{NAME} \\ \text{secondF1} : \text{NAME} \times \text{NAME} \rightarrow \text{NAME} \end{array}}{\forall x : \text{NAME}; y : \text{NAME} \bullet \text{firstF1}(x, y) = x \wedge \text{secondF1}(x, y) = y}$$

$$\frac{\begin{array}{l} \text{firstF2} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \\ \text{secondF2} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \end{array}}{\forall x : \mathbb{Z}; y : \mathbb{Z} \bullet \text{firstF2}(x, y) = x \wedge \text{secondF2}(x, y) = y}$$

These axiomatic definitions influenced those usages as follows:

$$\begin{aligned} b! &= \text{secondF}(\text{number}, c?) \\ a! &= \text{firstF1}(a?, b?) \\ c! &= \text{firstF}(c?, \text{number}) \\ d! &= \text{secondF2}(2, 4) \end{aligned}$$

The generated SAL file could be verified by the SAL model checker, in a case there was no theorem. However, in a case there was a theorem, this SAL file could not be verified by the SAL model checker. The error is as follows:

```
[Context: output_fFirst, line(25), column(10)]:
Failed to convert function application (array selection).
```

The below theorems were added to the SAL file:

```
th1: THEOREM State |- G(FORALL (i, j: NAT): i = j =>
firstF(i, j) = secondF(i, j));
th2: THEOREM State |- G(FORALL (i, j: NAT): firstF(i, j) /= secondF(i, j));
```

Those three axiomatic definitions were modified manually and they were put on the context clause.

```
firstF(q--1 : NAT, q--2 : NAT): NAT = q--1;
secondF(q--1 : NAT, q--2 : NAT): NAT = q--2;
firstF1(q--1 : NAME, q--2 : NAME): NAME = q--1;
secondF1(q--1 : NAME, q--2 : NAME): NAME = q--2;
firstF2(q--1 : INT, q--2 : INT): INT = q--1;
secondF2(q--1 : INT, q--2 : INT): INT = q--2;
```

Total execution time was 0.141 second. The first theorem was VALID, and the second one was INVALID. This generated SAL file could be simulated by the SAL simulator.

8.2.9 Experiment 9: fHead.tex

This Z specification was created by us and a generic constant definition which was taken from [69, p. 117] was added to this specification. There is one generic constant on this specification, `headSeq`. This generic constant is a total function from a sequence of `X` to a `X`.

$$\frac{[X]}{\frac{\text{headSeq} : (\text{seq}_1 X) \rightarrow X}{\forall s : \text{seq}_1 X \bullet \text{headSeq}(s) = s(1)}}$$

There are three usages on this generic constant as follows:

$$\begin{aligned} a! &= \text{headSeq}[\text{seq}_1 \text{NAME}](\text{name}) \wedge b! = \text{headSeq}(b?) \\ c! &= \text{headSeq}(\langle 5, 2, 4 \rangle) \end{aligned}$$

The first usage will actualise `X` with `NAME`, whereas the second usage has "N" for replacing `X`, and the last one is a direct usage which is obviously a sequence of integer number, "Z". `name` is a sequence of basic type variable, `NAME`, and `b?` is a sequence of natural number.

This system generated these three axiomatic definitions as follows:

$$\frac{\text{headSeq} : (\text{seq}_1 \text{NAME}) \rightarrow \text{NAME}}{\forall s : \text{seq}_1 \text{NAME} \bullet \text{headSeq}(s) = s(1)}$$

$$\frac{\text{headSeq1} : (\text{seq}_1 \mathbb{N}) \rightarrow \mathbb{N}}{\forall s : \text{seq}_1 \mathbb{N} \bullet \text{headSeq1}(s) = s(1)}$$

$$\frac{\text{headSeq2} : (\text{seq}_1 \mathbb{Z}) \rightarrow \mathbb{Z}}{\forall s : \text{seq}_1 \mathbb{Z} \bullet \text{headSeq2}(s) = s(1)}$$

Those three usages were modified as follows:

$$\begin{aligned} a! &= \text{headSeq}(\text{name}) \wedge b! = \text{headSeq1}(b?) \\ c! &= \text{headSeq2}(\langle 5, 2, 4 \rangle) \end{aligned}$$

The generated SAL file of this specification with no added theorem could be verified by the SAL model checker. Total execution time is 0.032 second. However, the SAL simulator failed to simulate this SAL file with an out of memory error.

Thus, these three functions were modified by specifying them on the context clause as shown below:

```

headSeq(q--1 : Seq--B--NAME): B--NAME = q--1(1);
headSeq1(q--2 : Seq--B--NAT): B--NAT = q--2(1);
headSeq2(q--3 : Seq--B--INT): B--INT = q--3(1);

```

There were three theorems added to this generated SAL file, as follows:

```

th1: THEOREM State |- G(EXISTS (i: NAME): i = headSeq(name));
th2: THEOREM State |- G(FORALL (i: NAME): i = headSeq(name));
th3: THEOREM State |- G(EXISTS (i: NAME): i = headSeq(name) =>
sequence{B--NAME; NAME--BB, 3}!contains?(name, i));

```

Total execution time was 0.281 second. Both **th1** and **th3** theorems are VALID, whereas the **th2** theorem is INVALID. This modified SAL file could be simulated by the SAL simulator.

8.2.10 Experiment 10: fHeadFunc.tex

This specification is a modification of previous specification. In this specification, the same generic constant as given above is used by functions defined on the axiomatic definitions. These functions are called on a schema.

$$\frac{fHead1 : (\text{seq}_1 \mathbb{N}) \rightarrow \mathbb{N}}{\forall n : \text{seq}_1 \mathbb{N} \bullet fHead1(n) = headSeq(n)}$$

$$\frac{fHead2 : (\text{seq}_1 NAME) \rightarrow NAME}{\forall n : \text{seq}_1 NAME \bullet fHead2(n) = headSeq(n)}$$

These below predicates call both functions:

$$a! = fHead2(name) \wedge b! = fHead1(b?)$$

With the same types for variables as given on previous experiment, two axiomatic definitions to redefine the generic constant definition are as follows:

$$\frac{headSeq : (\text{seq}_1 \mathbb{N}) \rightarrow \mathbb{N}}{\forall s : \text{seq}_1 \mathbb{N} \bullet headSeq(s) = s(1)}$$

$$\frac{headSeq1 : (\text{seq}_1 NAME) \rightarrow NAME}{\forall s : \text{seq}_1 NAME \bullet headSeq1(s) = s(1)}$$

Afterwards, the first usage will be modified to:

$$\forall n : \text{seq}_1 \mathbb{N} \bullet fHead1(n) = headSeq(n)$$

The second one is as follows:

$$\forall n : \text{seq}_1 \text{ NAME} \bullet \text{fHead2}(n) = \text{headSeq1}(n)$$

The generated SAL file of this specification with no theorem could be verified by the SAL model checker. Total execution time was 0.047 second. However, this SAL file could not be simulated by the SAL simulator. Modification version of these functions still could not be simulated by the SAL model checker. The set of initial states is empty was an error message produced by the SAL simulator.

8.2.11 Experiment 11: fMaxComSubSeq_orig.tex

This specification was created by us and several generic constant definition were taken from [2, p. 163-164] which were added to this specification. This will output the longest shared subsequence of two sequences.

There were four generic constant definitions on this specification. The fourth generic constant calls the third generic constant, the third generic constant calls the second generic constant, and the second generic constant calls the first generic constant. The fourth generic constant will be used on a schema.

$$\begin{array}{|l} \hline [X] \\ \hline \text{subseqOf} : (\text{seq } X) \leftrightarrow (\text{seq } X) \\ \hline \forall s, t : \text{seq } X \bullet (s, t) \in \text{subseqOf} \Leftrightarrow (\exists p : \mathbb{P} \mathbb{N}_1 \bullet s = p \upharpoonright t) \\ \hline \end{array}$$

$$\begin{array}{|l} \hline [X] \\ \hline \text{allSubseq} : (\text{seq } X) \rightarrow \mathbb{P}(\text{seq } X) \\ \hline \text{allSubseq} = (\lambda s : \text{seq } X \bullet \{t : \text{seq } X \mid (t, s) \in \text{subseqOf}\}) \\ \hline \end{array}$$

$$\begin{array}{|l} \hline [X] \\ \hline \text{commonSubseq} : ((\text{seq } X) \times (\text{seq } X)) \rightarrow \mathbb{P}(\text{seq } X) \\ \hline \text{commonSubseq} = (\lambda s, t : \text{seq } X \bullet \text{allSubseqs} \cap \text{allSubseqt}) \\ \hline \end{array}$$

$$\begin{array}{|l} \hline [X] \\ \hline \text{maxCommonSubseq} : ((\text{seq } X) \times (\text{seq } X)) \rightarrow \mathbb{P}(\text{seq } X) \\ \hline \text{maxCommonSubseq} = (\lambda s, t : \text{seq } X \bullet \\ \{x : \text{commonSubseq}(s, t) \mid (\forall y : \text{commonSubseq}(s, t) \bullet \#x \geq \#y)\}) \\ \hline \end{array}$$

As can be seen from above schemas, there is the Z tag, "↑", which will extract a new sequence from an old sequence based on a set of indices. This

tag has been specified in the Z2SAL mathematical tool-kit, but not on the Z2SAL parser. Then, Z2SAL's team upgraded the Z2SAL parser so this tag can be used on Z specification inputs.

Another Z tag that appears on this Z specification is the lambda expression. This tag is used commonly to define a function without specifies a name on it [2]. The lambda expression, $(\lambda \mathbf{S} \bullet \mathbf{E})$, represents a function and has arguments which are taken from \mathbf{S} . The output of this expression is the value of \mathbf{E} [69].

Z2SAL does not support this expression which inevitably is often to exist on a generic constant definition or on other definitions in a Z specification generally. Thus, our approach is to rewrite the lambda automatically and manually to an equivalent expression without any lambda expression. Afterwards, it is redefined to an axiomatic definition.

The first approach is to rewrite automatically this tag to its equivalent expression which can be supported by Z2SAL. The equivalent expression is on the form of universal quantification. There is another equivalent expression for the lambda expression which is on the form of a set comprehension. However, based on our experiences, the second equivalent expression can raise a problem in a case of more than one declared variable. Thus, the first equivalent expression was chosen to rewrite this lambda expression automatically.

Based on the usage of the fourth generic constant definition, which was specified by us, as follows:

$$a! = \text{maxCommonSubseq}(\text{num}, a?)$$

with num is a "seq \mathbb{N} " type state variable, and $a?$ is a sequence of " \mathbb{N} ", this generic constant definition was redefined as follows:

$$\left| \begin{array}{l} \text{maxCommonSubseq} : ((\text{seq } \mathbb{N}) \times (\text{seq } \mathbb{N})) \rightarrow \mathbb{P}(\text{seq } \mathbb{N}) \\ \hline \forall s, t : \text{seq } \mathbb{N} \bullet \\ \text{maxCommonSubseq}(s, t) = \{x : \text{commonSubseq}(s, t) \mid \\ (\forall y : \text{commonSubseq}(s, t) \bullet \#x \geq \#y)\} \end{array} \right.$$

This generic constant uses the third generic constant by passing "seq \mathbb{N} " to actualise the generic parameter. The generated axiomatic definition, but on the equivalent expression form of the lambda expression, is as follows:

$$\left| \begin{array}{l} \text{commonSubseq} : ((\text{seq } \mathbb{N}) \times (\text{seq } \mathbb{N})) \rightarrow \mathbb{P}(\text{seq } \mathbb{N}) \\ \hline \forall s, t : \text{seq } \mathbb{N} \bullet \text{commonSubseq}(s, t) = \text{allSubseqs} \cap \text{allSubseqt} \end{array} \right.$$

The equivalent expression of the second generic constant was generated as follows:

$$\frac{\text{allSubseq} : (\text{seq } \mathbb{N}) \rightarrow \mathbb{P}(\text{seq } \mathbb{N})}{\forall s : \text{seq } \mathbb{N} \bullet \text{allSubseqs} = \{t : \text{seq } \mathbb{N} \mid (t, s) \in \text{subseqOf}\}}$$

The last equivalent expression is given as follows:

$$\frac{\text{subseqOf} : (\text{seq } \mathbb{N}) \leftrightarrow (\text{seq } \mathbb{N})}{\forall s, t : \text{seq } \mathbb{N} \bullet (s, t) \in \text{subseqOf} \Leftrightarrow (\exists p : \mathbb{P} \mathbb{N}_1 \bullet s = p \upharpoonright t)}$$

The second approach was to specify two new Z specifications and rewrite this lambda expression to its equivalent expression. The first Z specification has a universal quantifier to rewrite the lambda expression. This will be discussed further on the 9.2.12 on page 236 sub-section. The second one has a set comprehension instead and it will be discussed on the 9.2.13 on page 237 sub-section.

The generated SAL file of this specification could not be verified by the SAL model checker. The error was produced as follows:

```
Error: [Context: output_fMaxComSubSeq_orig, line(43), column(19)]:
Type mismatch in the function application.
Expected type:
[sequenceoutput_fMaxComSubSeq_orig!B__NAT, 4, 3!Domain,
sequence{output_fMaxComSubSeq_orig!B__NAT, 4, 3!Sequence}
Actual type:
[set{output_fMaxComSubSeq_orig!NZNAT!Set,
output_fMaxComSubSeq_orig!Seq__B__NAT]
```

The associated SAL's lines are as follows:

```
38 (FORALL (q--1 : Seq__B__NAT, q--2 : Seq__B__NAT) : set
39 {Seq__B__NAT__X__Seq__B__NAT;} !
40 contains?(subseqOf, (q--1, q--2)) =
41 (EXISTS (q--3 : set {NZNAT;} !
42 Set) : q--1 = sequence {B__NAT; 4, 3} !
43 extract(q--3, q--2))) AND
```

The type of parameters which were passed to `extract` were not compatible with the type of parameters which were declared for `extract`.

Thus, modification was made as follows:

```
40 (FORALL (q--1 : Seq__B__NAT, q--2 : Seq__B__NAT) : set
41 {Seq__B__NAT__X__Seq__B__NAT;} !
42 contains?(subseqOf, (q--1, q--2)) =
43 (EXISTS (q--3 : sequence {B__NAT; 4, 3} ! Sequence) :
44 q--1 = sequence {B__NAT; 4, 3} !
44 extract(sequence {B__NAT; 4, 3} ! domain(q--3), q--2))) AND
```

on the context clause. There is also a new type as follows:

B_NZNAT : TYPE = [0..3];

This time, this SAL file could be verified by the SAL model checker with 0.032 second for execution time. However, the SAL simulator produced an out of memory error.

8.2.12 Experiment 12: fMaxComSubSeq_1.tex

This specification is the first equivalent expression for the lambda expression. Following is the associated generic constant definitions:

$\begin{array}{l} \text{subseqOf} : (\text{seq } X) \leftrightarrow (\text{seq } X) \\ \forall s, t : \text{seq } X \bullet (s, t) \in \text{subseqOf} \Leftrightarrow (\exists p : \mathbb{P}N_1 \bullet s = p \upharpoonright t) \end{array}$

$\begin{array}{l} \text{allSubseq} : (\text{seq } X) \rightarrow \mathbb{P}(\text{seq } X) \\ \forall s : \text{seq } X \bullet \text{allSubseqs} = \{t : \text{seq } X \mid (t, s) \in \text{subseqOf}\} \end{array}$
--

$\begin{array}{l} \text{commonSubseq} : ((\text{seq } X) \times (\text{seq } X)) \rightarrow \mathbb{P}(\text{seq } X) \\ \forall s, t : \text{seq } X \bullet \text{commonSubseq}(s, t) = \text{allSubseqs} \cap \text{allSubseqt} \end{array}$
--

$\begin{array}{l} \text{maxCommonSubseq} : ((\text{seq } X) \times (\text{seq } X)) \rightarrow \mathbb{P}(\text{seq } X) \\ \forall s, t : \text{seq } X \bullet \text{maxCommonSubseq}(s, t) = \{x : \text{commonSubseq}(s, t) \mid \\ (\forall y : \text{commonSubseq}(s, t) \bullet \#x \geq \#y)\} \end{array}$
--

To obtain the equivalent expression from the lambda expression is as follows:

- Add a universal quantifier on the first line of predicate. This quantifier has variables which are obtained from the lambda expression's variables.
- Add a "•" which is then followed by the name of the generic constant. In a case a generic constant is a function, and then the generic constant name will be followed by its parameters. For a case of a relation, there is no parameter.
- Add an equation sign and end it with the rest of old predicate until the last curly bracket.

The axiomatic definitions for these equivalent expressions are the same as previous experiments.

8.2.13 Experiment 13: fMaxComSubSeq.tex

This is the second equivalent expression for the lambda expression.

$$\begin{array}{l} \overline{\overline{[X]}} \\ \text{subseqOf} : (\text{seq } X) \leftrightarrow (\text{seq } X) \\ \hline \forall s, t : \text{seq } X \bullet (s, t) \in \text{subseqOf} \Leftrightarrow (\exists p : \mathbb{P} \mathbb{N}_1 \bullet s = p \upharpoonright t) \end{array}$$

$$\begin{array}{l} \overline{\overline{[X]}} \\ \text{allSubseq} : (\text{seq } X) \rightarrow \mathbb{P}(\text{seq } X) \\ \hline \text{allSubseq} = \{s : \text{seq } X \bullet (s, \{t : \text{seq } X \mid (t, s) \in \text{subseqOf}\})\} \end{array}$$

$$\begin{array}{l} \overline{\overline{[X]}} \\ \text{commonSubseq} : ((\text{seq } X) \times (\text{seq } X)) \rightarrow \mathbb{P}(\text{seq } X) \\ \hline \text{commonSubseq} = \{s, t : \text{seq } X \bullet ((s, t), \text{allSubseq}(s) \cap \text{allSubseq}(t))\} \end{array}$$

$$\begin{array}{l} \overline{\overline{[X]}} \\ \text{maxCommonSubseq} : ((\text{seq } X) \times (\text{seq } X)) \rightarrow \mathbb{P}(\text{seq } X) \\ \hline \text{maxCommonSubseq} = \{s, t : \text{seq } X \bullet ((s, t), \{x : \text{commonSubseq}(s, t) \mid (\forall y : \text{commonSubseq}(s, t) \bullet \#x \geq \#y)\})\} \end{array}$$

Though these generic constant definitions are written by using set comprehension forms, their axiomatic definitions are still written on forms of the first equivalent expression.

8.2.14 Experiment 14: fMonoSeq_1.tex

These generic abbreviation definitions were taken from the same book as above experiment, but from pages 336. Then, a specification was created by us which were filled by these generic definitions. These generic abbreviation definitions specify a sequence of a single element.

There are two generic definitions on this specification and they were specified in an abbreviation form. They are constants.

$$\text{monoSequence}[X] == \{s : \text{seq } X \mid \#(\text{ran } s) \leq 1\}$$

$$\text{monoSequenceOne}[X] == \{s : \text{monoSequence } X \mid \#s > 0\}$$

The second constant calls the first constant. This will output a non-empty sequence of a single element.

There were two usages specified by us for this specification as follows:

$$\begin{aligned} \text{alph} &= \text{monoSequence}[\text{ALPHABET}] \\ \text{b!} &= \text{monoSequenceOne}[\mathbb{Z}] \end{aligned}$$

ALPHABET is a basic type. Both usages were written on an explicit type of parameter.

Thus, axiomatic definitions were generated as follows:

$$\frac{\text{monoSequence1} : \mathbb{P}(\text{seq } \mathbb{Z})}{\text{monoSequence1} = \{s : \text{seq } \mathbb{Z} \mid \#(\text{ran } s) \leq 1\}}$$

$$\frac{\text{monoSequence} : \mathbb{P}(\text{seq } \text{ALPHABET})}{\text{monoSequence} = \{s : \text{seq } \text{ALPHABET} \mid \#(\text{ran } s) \leq 1\}}$$

$$\frac{\text{monoSequenceOne} : \mathbb{P}(\text{seq } \mathbb{Z})}{\text{monoSequenceOne} = \{s : \text{monoSequence1} \mid \#s > 0\}}$$

These axiomatic definitions influenced those usages as follows:

$$\begin{aligned} \text{alph} &= \text{monoSequence} \\ \text{b!} &= \text{monoSequenceOne} \end{aligned}$$

The generated SAL file could be verified by the SAL model checker. The total execution time was 0.031 second for no theorem. This SAL file could also be simulated by the SAL simulator, though it required a significant long simulation time.

8.2.15 Experiment 15: fMonoSeq.tex

This specification is an equivalent form of above specification. A generic abbreviation definition can be rewritten to a generic constant definition. Thus, both abbreviation definitions above were rewritten manually on this specification as follows:

$$\frac{[X]}{\frac{\text{monoSequence} : \mathbb{P}(\text{seq } X)}{\text{monoSequence} = \{s : \text{seq } X \mid \#(\text{ran } s) \leq 1\}}}$$

$$\frac{[X]}{\frac{\text{monoSequenceOne} : \mathbb{P}(\text{seq } X)}{\text{monoSequenceOne} = \{s : \text{monoSequence}X \mid \#s > 0\}}}$$

A method to obtain a generic constant definition from a generic abbreviation definition as follows:

- The name of generic constant is got from the abbreviation variable.
- Since it is a constant, there is no input parameter.
- The output type for this constant is obtained a type of variable declared after the abbreviation name. In this case, after the abbreviation name is a set comprehension with one declared variable which type is a sequence of X . Thus, this type is a set of a sequence of X .
- Predicate for this generic constant definition is specified as the name of abbreviation. It will be followed by an equation sign and all expressions after "==" .

The same usages as above experiment were used on this specification. The same axiomatic definitions as above experiment were generated.

8.2.16 Experiment 16: fSwap.tex

This specification was created by us and it was added by two generic constant definitions which were taken from [57]. There were two generic constant definitions specified in this Z specification as follows:

$$\begin{array}{l} \hline \hline [X, Y] \\ \hline \text{swap2} : X \times Y \rightarrow Y \times X \\ \hline \forall x : X; y : Y \bullet \text{swap2}(x, y) = (y, x) \\ \hline \end{array}$$

$$\begin{array}{l} \hline \hline [X] \\ \hline \text{swap1} : X \times X \rightarrow X \times X \\ \hline \forall x, y : X \bullet \text{swap1}(x, y) = (y, x) \\ \hline \end{array}$$

Both of them are total functions. The second generic constant has just one generic parameter. This generic constant swaps the order of its parameters.

Two usages were specified by us in those generic constants as follows:

$$\begin{array}{l} (b!, a!) = \text{swap1}[\text{NAME}, \text{NAME}](\text{name}, a?) \\ (c!, a!) = \text{swap2}(\text{name}, c?) \end{array}$$

The first usage is defined by using two parameters, but with the same types. In this usage, the explicit types for its parameters are also given in the same time as its actual parameters are given. The second usage is specified by passing NAME for X and "N" for Y .

This Z specification can be redefined by our system to its axiomatic definitions.

$$\frac{}{\frac{}{\text{swap1} : \text{NAME} \times \text{NAME} \rightarrow \text{NAME} \times \text{NAME}}{\forall x, y : \text{NAME} \bullet \text{swap1}(x, y) = (y, x)}}$$

$$\frac{}{\frac{}{\text{swap2} : \text{NAME} \times \mathbb{N} \rightarrow \mathbb{N} \times \text{NAME}}{\forall x : \text{NAME}; y : \mathbb{N} \bullet \text{swap2}(x, y) = (y, x)}}$$

Both usages will be modified to:

$$\begin{aligned} (b!, a!) &= \text{swap1}(\text{name}, a?) \\ (c!, a!) &= \text{swap2}(\text{name}, c?) \end{aligned}$$

The generated SAL file could be verified by the SAL model checker. It required 0.016 second execution time and no theorem. However, in a case there were theorems, an error was produced as follows:

```
Error: [Context: output_fSwap, line(22), column(10)]:
Failed to convert function application (array selection).
The function/array does not have a finite domain,
or the argument is not in the domain of the function/array.
```

Thus, both axiomatic function definitions were modified to forms as follows:

```
swap1(q--1 : NAME, q--2 : NAME): B_NAME_X_B_NAME = (q--2, q--1);
swap2(q--3 : NAME, q--4 : NAT): B_NAT_X_B_NAME = (q--4, q--3);
```

Theorems that were added to this SAL file are as follows:

```
th1: theorem State |- G(FORALL (i, j: NAME): swap1(i, j) = (j, i));
th2: theorem State |- G(FORALL (i, j: NAME): i = j => swap1(i, j) = swap1(j, i));
th3: theorem State |- G(FORALL (i, j: NAME): swap1(i, j) = swap1(j, i));
```

The first two theorems are VALID, the last one is INVALID. Total execution time was 0.156 second. This modified SAL file could be simulated by the SAL simulator.

8.2.17 Experiment 17: fUniqSeq.tex

The generic constant definition was taken from [2, p. 72-73]. This generic constant replaces multiple consecutive copies of an element in a sequence by a single copy of that element. Then, it was combined with the generic constant definition of the abbreviation expression from page 336 as used on the previous experiment. A specification was created afterwards.

The generic constant definitions are as follows:

$$\begin{array}{|l} \hline [X] \\ \hline \hline \text{monoSequence} : (\text{seq } X) \rightarrow \mathbb{P} X \\ \hline \text{monoSequence}X = \{s : \text{seq } X \mid \#(\text{ran } s) \leq 1\} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline [X] \\ \hline \hline \text{monoSequenceOne} : (\text{seq } X) \rightarrow \mathbb{P} X \\ \hline \text{monoSequenceOne}X = \{s : \text{monoSequence}X \mid \#s > 0\} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline [X] \\ \hline \hline \text{uniqSequence} : (\text{seq } X) \rightarrow \text{seq } X \\ \hline \text{uniqSequence} \langle \rangle = \langle \rangle \\ \forall x : X; s : \text{monoSequenceOne}X \bullet \text{ran } s = \{x\} \wedge \text{uniqSequences} = \langle x \rangle \\ \forall s, t : \text{seq}_1 X \bullet \text{last } s \neq \text{head } t \wedge \\ \text{uniqSequence}(s \hat{\ } t) = \text{uniqSequence}(s) \hat{\ } \text{uniqSequence}(t) \\ \hline \end{array}$$

There are three usages specified in these generic constants as follows:

$$\begin{array}{l} a! = \text{uniqSequence}(\text{num}) \\ b! = \text{uniqSequence}(b?) \\ c! = \text{monoSequenceOne}[\mathbb{Z}] \end{array}$$

As can be seen from the first usage, the generic constant which is the `uniSequence` function was used by passing its actual type "seq \mathbb{N} ". In the definition of this generic constant, another generic constant `monoSequenceOne` was used with the same number of generic parameter as `uniSequence`. Thus, the `monoSequenceOne` generic parameter would be replaced by the actual type read from `uniSequence`. The definition of `monoSequenceOne` also uses another generic constant `monoSequence` which has the same number of generic parameter as both earlier generic constants.

The second usage is a usage on the same generic constant as the first usage, but different on the actual type. In this usage, `b?` is a sequence of " \mathbb{Z} ".

The last usage uses `monoSequenceOne` and explicit output type, " \mathbb{Z} ". This specification can be redefined by our tool as follows:

$$\begin{array}{|l} \hline \text{monoSequence1} : \mathbb{P}(\text{seq } \mathbb{Z}) \\ \hline \text{monoSequence1} = \{s : \text{seq } \mathbb{Z} \mid \#(\text{ran } s) \leq 1\} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \text{monoSequenceOne1} : \mathbb{P}(\text{seq } \mathbb{Z}) \\ \hline \text{monoSequenceOne1} = \{s : \text{monoSequence1} \mid \#s > 0\} \\ \hline \end{array}$$

$\frac{\text{monoSequence} : \mathbb{P}(\text{seq } \mathbb{N})}{\text{monoSequence} = \{s : \text{seq } \mathbb{N} \mid \#(\text{ran } s) \leq 1\}}$
$\frac{\text{monoSequenceOne} : \mathbb{P}(\text{seq } \mathbb{N})}{\text{monoSequenceOne} = \{s : \text{monoSequence} \mid \#s > 0\}}$
$\frac{\text{uniqSequence} : (\text{seq } \mathbb{N}) \rightarrow (\text{seq } \mathbb{N})}{\begin{array}{l} \text{uniqSequence}\langle \rangle = \langle \rangle \\ \forall x : \mathbb{N}; s : \text{monoSequenceOne} \bullet \text{ran } s = \{x\} \Rightarrow \text{uniqSequences} = \langle x \rangle \\ \forall s, t : \text{seq}_1 \mathbb{N} \bullet \text{last } s \neq \text{head } t \Rightarrow \text{uniqSequence}(s \hat{\ } t) = \text{uniqSequence}(s) \hat{\ } \text{uniqSequence}(t) \end{array}}$
$\frac{\text{uniqSequence1} : (\text{seq } \mathbb{Z}) \rightarrow (\text{seq } \mathbb{Z})}{\begin{array}{l} \text{uniqSequence1}\langle \rangle = \langle \rangle \\ \forall x : \mathbb{Z}; s : \text{monoSequenceOne1} \bullet \text{ran } s = \{x\} \Rightarrow \text{uniqSequence1}s = \langle x \rangle \\ \forall s, t : \text{seq}_1 \mathbb{Z} \bullet \text{last } s \neq \text{head } t \Rightarrow \text{uniqSequence1}(s \hat{\ } t) = \text{uniqSequence1}(s) \hat{\ } \text{uniqSequence1}(t) \end{array}}$

Then, those usages were modified to:

$$\begin{array}{l} a! = \text{uniqSequence}(\text{num}) \\ b! = \text{uniqSequence1}(b?) \\ c! = \text{monoSequenceOne1} \end{array}$$

The generated SAL file with no theorem could be verified by the SAL model checker. The total execution time was 0.062 second. The SAL simulator could not simulate this generated SAL file. An error of out of memory was produced instead.

8.2.18 Experiment 18: fUniq1Seq.tex

The generic constant definition on this specification was taken from the same book as previous experiment. This generic constant definition performs the same as the generic constant definition from previous experiment. However, at this time, the generic constant does not call another generic constant as previous experiment.

$\frac{[X]}{\text{uniqSequence} : (\text{seq } X) \rightarrow (\text{seq } X)}$
$\begin{array}{l} \text{uniqSequence}\langle \rangle = \langle \rangle \\ \forall x : X \bullet \text{uniqSequence}\langle x \rangle = \langle x \rangle \\ \forall x : X; s : \text{seq } X \bullet \text{uniqSequence}\langle \langle x, x \rangle \hat{\ } s \rangle = \text{uniqSequence}\langle \langle x \rangle \hat{\ } s \rangle \\ \forall x, y : X; s : \text{seq } X \bullet x \neq y \wedge \text{uniqSequence}\langle \langle x, y \rangle \hat{\ } s \rangle = \langle x \rangle \hat{\ } \text{uniqSequence}\langle \langle y \rangle \hat{\ } s \rangle \end{array}$

The complete specification was created by us. Two usages were specified in this generic constant as follows:

$a! = \text{uniqSequence}(num)$
 $b! = \text{uniqSequence}(b?)$

Two axiomatic definitions were generated based on those usages which replace the generic constant definition.

$\text{uniqSequence} : (\text{seq } \mathbb{N}) \rightarrow (\text{seq } \mathbb{N})$
$\text{uniqSequence}(\langle \rangle) = \langle \rangle$
$\forall x : \mathbb{N} \bullet \text{uniqSequence}\langle x \rangle = \langle x \rangle$
$\forall x : \mathbb{N}; s : \text{seq } \mathbb{N} \bullet \text{uniqSequence}(\langle x, x \rangle \hat{\ } s) = \text{uniqSequence}(\langle x \rangle \hat{\ } s)$
$\forall x, y : \mathbb{N}; s : \text{seq } \mathbb{N} \bullet x \neq y \wedge \text{uniqSequence}(\langle x, y \rangle \hat{\ } s) = \langle x \rangle \hat{\ } \text{uniqSequence}(\langle y \rangle \hat{\ } s)$

$\text{uniqSequence1} : (\text{seq } \mathbb{Z}) \rightarrow (\text{seq } \mathbb{Z})$
$\text{uniqSequence1}\langle \rangle = \langle \rangle$
$\forall x : \mathbb{Z} \bullet \text{uniqSequence1}\langle x \rangle = \langle x \rangle$
$\forall x : \mathbb{Z}; s : \text{seq } \mathbb{Z} \bullet \text{uniqSequence1}(\langle x, x \rangle \hat{\ } s) = \text{uniqSequence1}(\langle x \rangle \hat{\ } s)$
$\forall x, y : \mathbb{Z}; s : \text{seq } \mathbb{Z} \bullet x \neq y \wedge \text{uniqSequence1}(\langle x, y \rangle \hat{\ } s) = \langle x \rangle \hat{\ } \text{uniqSequence1}(\langle y \rangle \hat{\ } s)$

The two usages were modified as follows:

$a! = \text{uniqSequence}(num)$
 $b! = \text{uniqSequence1}(b?)$

This generated SAL file with no theorem could be verified by the SAL model checker with 0.047 second total execution time. However, it could not be simulated by the SAL simulator with an out of memory error.

8.2.19 Experiment 19: fUniq2Seq.tex

This specification is another variance of previous experiments.

$\text{uniqSequence} : (\text{seq } X) \rightarrow (\text{seq } X)$
$\text{uniqSequence}(\langle \rangle) = \langle \rangle$
$\forall x : X \bullet \text{uniqSequence}\langle x \rangle = \langle x \rangle$
$\forall x : X; s : \text{seq } X \bullet \text{uniqSequence}(s \hat{\ } \langle x, x \rangle) = \text{uniqSequence}(s \hat{\ } \langle x \rangle)$
$\forall x, y : X; s : \text{seq } X \bullet x \neq y \wedge \text{uniqSequence}(s \hat{\ } \langle y, x \rangle) = \text{uniqSequence}((s \hat{\ } \langle y \rangle) \hat{\ } \langle x \rangle)$

The same usages as previous experiment were used on this specification. Two axiomatic definitions were generated as follows:

$\text{uniqSequence} : (\text{seq } \mathbb{N}) \rightarrow (\text{seq } \mathbb{N})$
$\text{uniqSequence}(\langle \rangle) = \langle \rangle$
$\forall x : \mathbb{N} \bullet \text{uniqSequence}\langle x \rangle = \langle x \rangle$
$\forall x : \mathbb{N}; s : \text{seq } \mathbb{N} \bullet \text{uniqSequence}(s \hat{\ } \langle x, x \rangle) = \text{uniqSequence}(s \hat{\ } \langle x \rangle)$
$\forall x, y : \mathbb{N}; s : \text{seq } \mathbb{N} \bullet x \neq y \wedge \text{uniqSequence}(s \hat{\ } \langle y, x \rangle) = \text{uniqSequence}((s \hat{\ } \langle y \rangle) \hat{\ } \langle x \rangle)$

$\begin{array}{l} \text{uniqSequence1} : (\text{seq } \mathbb{Z}) \rightarrow (\text{seq } \mathbb{Z}) \\ \text{uniqSequence1}\langle \rangle = \langle \rangle \\ \forall x : \mathbb{Z} \bullet \text{uniqSequence1}\langle x \rangle = \langle x \rangle \\ \forall x : \mathbb{Z}; s : \text{seq } \mathbb{Z} \bullet \text{uniqSequence1}(s \hat{\ } \langle x, x \rangle) = \text{uniqSequence1}(s \hat{\ } \langle x \rangle) \\ \forall x, y : \mathbb{Z}; s : \text{seq } \mathbb{Z} \bullet x \neq y \wedge \text{uniqSequence1}(s \hat{\ } \langle y, x \rangle) = \text{uniqSequence1}((s \hat{\ } \langle y \rangle) \hat{\ } \langle x \rangle) \end{array}$
--

This generated SAL file could be verified by the SAL model checker. The total time was 0.032 second for executing no theorem version of this SAL file. The SAL simulator failed to simulate this SAL file because of out of memory error.

8.2.20 Experiment 20: tn.tex

This specification was taken from [31, p. 31-34]. There was one generic constant definition in this specification.

$\begin{array}{l} \text{disjoint} : \mathbb{P}(\mathbb{P}(\mathbb{P} X)) \\ \forall \text{cons} : \mathbb{P}(\mathbb{P} X) \bullet \text{cons} \in \text{disjoint} \Leftrightarrow (\forall c1, c2 : \text{cons} \bullet c1 \neq c2 \Rightarrow c1 \cap c2 = \emptyset) \end{array}$
--

This generic constant is a constant. The usages of this generic constant were specified to be explicit type. This explicit type was PHONE. There are six usages as follows:

```

cons ∈ disjoint[PHONE]
cons0 ∈ disjoint[PHONE])
cons0 ∈ disjoint[PHONE]))
cons0 ∈ disjoint[PHONE]))))
cons0 ∈ disjoint[PHONE])))))
cons0 ∈ disjoint[PHONE]))))))

```

The second to the sixth usages are the same predicates.

The axiomatic definition was as follows:

$\begin{array}{l} \text{disjoint} : \mathbb{P}(\mathbb{P}(\mathbb{P} \text{PHONE})) \\ \forall \text{cons} : \mathbb{P}(\mathbb{P} \text{PHONE}) \bullet \text{cons} \in \text{disjoint} \Leftrightarrow (\forall c1, c2 : \text{cons} \bullet c1 \neq c2 \Rightarrow c1 \cap c2 = \emptyset) \end{array}$
--

It would redefine the generic constant definition. Those usages were still the same, but with the explicit types were removed.

Following error was produced by the SAL model checker:

```

Error: [Context: output_tn, line(110), column(27)]:
Type mismatch in the function application.
Expected type:
power{output_tn!CON}!Power
Actual type:

```

output_tn!Set__CON

Associated SAL lines are as follows:

```
108 (engaged_' = No => NOT set {PHONE;} !
109 contains?(power {CON;} !
110 generalUnion(cons), ph?) AND
```

`generalUnion` outputs a set, so `power` should no be passed with a set typed variable. Those SAL lines were modified as follows:

```
108 (engaged_' = No => NOT set {PHONE;} !
109 contains?(power {PHONE;} !
110 generalUnion(cons), ph?) AND
```

This modified SAL file could be verified by the SAL model checker. By no added theorem, the total execution time was 0.016 seconds. However, this SAL file could not be simulated by the SAL simulator because of out of memory error.

8.2.21 Experiment 21: tnImpl.tex

It is a variance of the previous experiment. The difference is the type was implicit. The usages are also the same as previous experiment. Thus, the same axiomatic definition was generated.

8.2.22 Experiment 22: fFileStorage.tex

This specification was taken from [31, p. 48-55]. There were two generic constant definitions on this specification. Both of them are functions.

$$\begin{array}{l} \overline{\overline{[X]}} \\ \text{after} : ((\text{seq } X) \times \mathbb{N}) \rightarrow (\text{seq } X) \\ \hline \forall s : \text{seq } X; \text{offset} : \mathbb{N} \bullet \\ \text{dom}(\text{after}(s, \text{offset})) = (1 \dots \#s - \text{offset}) \wedge (\forall n : \mathbb{N} \bullet \\ (n + \text{offset}) \in \text{dom } s \Rightarrow \text{after}(s, \text{offset})(n) = s(n + \text{offset})) \end{array}$$

$$\begin{array}{l} \overline{\overline{[X]}} \\ \text{shift} : ((\text{seq } X) \times \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow X) \\ \hline \forall s : \text{seq } X; \text{offset} : \mathbb{N} \bullet \\ \text{dom}(\text{shift}(s, \text{offset})) = \{i : \text{dom } s \bullet i + \text{offset}\} \wedge \\ (\forall n : \text{dom}(\text{shift}(s, \text{offset})) \bullet \text{shift}(s, \text{offset})(n) = s(n - \text{offset})) \end{array}$$

There were two usages on these generic constant definitions as follows:

$$\begin{array}{l} \text{data!} = (1 \dots \text{length?}) \triangleleft \text{after}(\text{file}, \text{offset?}) \\ \text{file}' = \text{zero}(\text{offset?}) \oplus \text{file} \oplus \text{shift}(\text{data?}, \text{offset?}) \end{array}$$

Based on these usages, following axiomatic definitions were generated:

$after : ((seq(0..255)) \times \mathbb{N}) \rightarrow (seq(0..255))$	$\forall s : seq(0..255); offset : \mathbb{N} \bullet \text{dom}(after(s, offset)) = (1.. \#s - offset) \wedge (\forall n : \mathbb{N} \bullet (n + offset) \in \text{dom } s \Rightarrow after(s, offset)(n) = s(n + offset))$
$shift : ((seq(0..255)) \times \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow (0..255))$	$\forall s : seq(0..255); offset : \mathbb{N} \bullet \text{dom}(shift(s, offset)) = \{i : \text{dom } s \bullet i + offset\} \wedge (\forall n : \text{dom}(shift(s, offset)) \bullet shift(s, offset)(n) = s(n - offset))$

However, the redefined version of this specification could not be translated by Z2SAL. Z2SAL produced a message that the current Z2SAL does not support the use of functions as the range of other functions.

8.2.23 Experiment 23: fSet.tex

This specification was a modification of the same specification found on [68]. There were several generic constant definitions as shown below:

$[X]$	$notEqual : X \leftrightarrow X$ $nonMember : X \leftrightarrow \mathbb{P} X$
	$\forall x, y : X \bullet (x, y) \in notEqual \Leftrightarrow \neg (x = y)$ $\forall x : X; S : \mathbb{P} X \bullet (x, S) \in nonMember \Leftrightarrow \neg (x \in S)$

$[X]$	$empty : \mathbb{P} X$
	$empty = \{x : X \mid \text{false}\}$

$[X]$	$uniSet : (\mathbb{P}(X)) \times (\mathbb{P}(X)) \rightarrow (\mathbb{P}(X))$
	$\forall S, T : \mathbb{P}(X) \bullet uniSet(S, T) = \{x : X \mid x \in S \vee x \in T\}$

The last two ones were specified in other Z specifications as well as `nonMember`. Several usages are as follows:

$$\begin{aligned}
used &= empty[\mathbb{N}_1] \\
alloc &= empty[\mathbb{N}_1] \\
alloc &= (empty[\mathbb{N}_1] \wedge (used, \mathbb{N}_1) \in notEqual) \Rightarrow (\exists n : \mathbb{N}_1 \bullet \\
&(n, used) \in nonMember \wedge alloc' = \{n\} \wedge used' = uniSet(used, \{n\})) \\
((alloc, empty[\mathbb{N}_1]) \in notEqual \vee used = \mathbb{N}_1) &\Rightarrow (alloc' = alloc \wedge used' = used) \\
(alloc, empty[\mathbb{N}_1]) \in notEqual &\Rightarrow n! \in alloc \wedge alloc' = empty[\mathbb{N}_1] \wedge used' = used \\
alloc = empty[\mathbb{N}_1] &\Rightarrow n! = 0 \wedge alloc' = alloc \wedge used' = used
\end{aligned}$$

They were grouped based on how they were specified in the associated Z specification. Each group is on different line on relevant predicate parts.

This specification could be redefined by our system. All generic constant definitions were redefined by following axiomatic definitions:

$$\begin{array}{|l}
\hline
\text{empty} : \mathbb{P} \mathbb{N}_1 \\
\hline
\text{empty} = \{x : \mathbb{N}_1 \mid \text{false}\} \\
\hline
\text{notEqual} : (\mathbb{P} \mathbb{N}_1) \leftrightarrow (\mathbb{P} \mathbb{N}_1) \\
\text{nonMember} : (\mathbb{P} \mathbb{N}_1) \leftrightarrow \mathbb{P}(\mathbb{P} \mathbb{N}_1) \\
\hline
\forall x, y : (\mathbb{P} \mathbb{N}_1) \bullet (x, y) \in \text{notEqual} \Leftrightarrow \neg (x = y) \\
\forall x : (\mathbb{P} \mathbb{N}_1); S : \mathbb{P}(\mathbb{P} \mathbb{N}_1) \bullet (x, S) \in \text{nonMember} \Leftrightarrow \neg (x \in S) \\
\hline
\text{notEqual1} : \mathbb{N}_1 \leftrightarrow \mathbb{N}_1 \\
\text{notMember1} : \mathbb{N}_1 \leftrightarrow \mathbb{P} \mathbb{N}_1 \\
\hline
\forall x, y : \mathbb{N}_1 \bullet (x, y) \in \text{notEqual1} \Leftrightarrow \neg (x = y) \\
\forall x : \mathbb{N}_1; S : \mathbb{P} \mathbb{N}_1 \bullet (x, S) \in \text{notMember1} \Leftrightarrow \neg (x \in S) \\
\hline
\text{uniSet} : (\mathbb{P} \mathbb{N}_1) \times (\mathbb{P} \mathbb{N}_1) \rightarrow (\mathbb{P} \mathbb{N}_1) \\
\hline
\forall S, T : \mathbb{P} \mathbb{N}_1 \bullet \text{uniSet}(S, T) = \{x : \mathbb{N}_1 \mid x \in S \vee x \in T\} \\
\hline
\end{array}$$

Those usages were modified as follows:

$$\begin{array}{l}
\text{used} = \text{empty} \\
\text{alloc} = \text{empty} \\
\text{alloc} = (\text{empty} \wedge (\text{used}, \mathbb{N}_1) \in \text{notEqual}) \Rightarrow (\exists n : \mathbb{N}_1 \bullet \\
(n, \text{used}) \in \text{nonMember1} \wedge \text{alloc}' = \{n\} \wedge \text{used}' = \text{uniSet}(\text{used}, \{n\})) \\
((\text{alloc}, \text{empty}) \in \text{notEqual} \vee \text{used} = \mathbb{N}_1) \Rightarrow (\text{alloc}' = \text{alloc} \wedge \text{used}' = \text{used}) \\
(\text{alloc}, \text{empty}) \in \text{notEqual} \Rightarrow n! \in \text{alloc} \wedge \text{alloc}' = \text{empty} \wedge \text{used}' = \text{used} \\
\text{alloc} = \text{empty}[\mathbb{N}_1] \Rightarrow n! = 0 \wedge \text{alloc}' = \text{alloc} \wedge \text{used}' = \text{used}
\end{array}$$

After the same modification as performed on previous experiment which is on the `uniSet` function, the modified SAL file could be verified by the SAL model checker. The total execution time was 0.0 second with no theorem. The SAL simulator could simulate also this modified SAL file.

Following is a section discussing our evaluation on this system.

8.3 Evaluation of Generic Constants Redefinition

As discussed above, there are 23 experiments over this system. Those experiments showed above are to present redefinition processes over a variety of generic constant declarations. Let us now move to an evaluation of this system by answering above questions on following sub-sections.

8.3.1 Evaluation of the #1 Question

The first question was set up to assess whether this system is able to redefine all generic constant definitions. In order to answer this question, discussions on several Z specifications from our above experiments are given here.

Based on our careful examination, the same generic constant definitions on usages which are greater than 1 were redefined to the equivalent axiomatic definitions which are different on their generic parameters. For example, an output of the Z specification `bbook.tex` from Section 9.2.1 on page 224, which is `output_bbook.tex`, will be discussed as follows.

There were two usages on the same generic constant definition `domain`. This generic constant name is a total function which will return a domain of this function. This domain was specified also as a function, but a partial function.

$$\begin{array}{l} \hline [X, Y] \hline \hline \text{domain} : (X \rightarrow Y) \rightarrow \mathbb{P} X \\ \hline \forall R : X \rightarrow Y \bullet \text{domain} R = \{x : X \mid \exists y : Y \bullet x \mapsto y \in R\} \\ \hline \end{array}$$

The first usage, which can be seen in the previous section, has `birthday` as its first actual parameter, and `known` as its second actual one. `birthday` is a partial function from `NAME` to `DATE`, whereas `known` is a set of `NAME`. Thus, `X` and its occurrences will be replaced with `NAME`, and `Y` and its occurrences will be replaced with `DATE`. As a result, the domain will be a set of `NAME` which match the type of `known`.

On the other hand, the second usage passed `house` as its parameter. `house` is also a partial function, but this time it is from `NAME` to `ADDRESS`. Thus, `X` will be actualised to `NAME` and `Y` will be actualised to `ADDRESS`. Since this is the second usage on the same generic constant as the first one, but with different actual types, name of the generic constant on this usage will be added with an index. At this case, it is `domain1`. The aim of this index is to differentiate between both these names as Z2SAL does not allow the same name for different functions.

Our system generated the same result as above description. It is shown in the associated experiment on the above section.

Another interesting example is `output_fFirst.tex` from Section 9.2.8 on page 233. In the same generic constant definition box, two generic constant names were specified. They are shown as follows:

$$\begin{array}{l} \hline [X, Y] \hline \hline \text{firstF} : X \times Y \rightarrow X; \text{secondF} : X \times Y \rightarrow Y \\ \hline \forall x : X; y : Y \bullet \text{firstF}(x, y) = x \wedge \text{secondF}(x, y) = y \\ \hline \end{array}$$

Several usages on these generic constant names are shown in the following schema:

<i>Find</i>
$a? : NAME; b? : NAME; a! : NAME$ $b! : \mathbb{N}; c? : \mathbb{N}; c!, d! : \mathbb{N}$ $\exists State$
$b! = secondF(number, c?)$ $a! = firstF(a?, b?)$ $c! = firstF(c?, number)$ $d! = secondF(2, 4)$

number is an instance of natural number, "N".

Based on these usages, our system generated axiomatic definition boxes which can be seen in the previous section. Above usages were also modified.

As can be seen from the outcome and comparing them with those usages, our system could actualise the same **X** on the same generic constant definition box but different generic constant names to the same type of parameter.

Complete examinations of usages of the same generic constant names are given in Table 8.3 on page 213. Although three generic parameters were specified in **Z** specifications, there is no **Z** specification on our experiments has three generic parameters on its generic constant definition. Table 8.3 on page 213 gives summaries about generic constant names whose their usages are greater than one.

However, as can be seen from Table 8.3 on page 213, there are several names which just have one usage, which is indicated by only "#1" and it is not accompanied by the second and greater numbers. It is a case of usages of the same generic constant names, but they are passed by the same types of generic parameters. For this case, our system does not redefine this generic constant name to one axiomatic definition and all the usages have the same names without any index.

The rests of our **Z** specification, whose usages on the same generic constant names is just one, are presented in Table 8.4 on page 214. Our system could generate **Z** specification whose axiomatic definitions have such types of parameters.

Therefore, based on these 23 experiments which could be redefined by our system, to some extent all these evidences prove that our system is able to redefine all generic constants correctly. The outcomes of our systems have also been checked manually to prove its correctness and reliability.

This proof is true and applicable to examples that were used for our experiments. It is possible that other **Z** specifications especially which have more complex types of generic constants or more complex usages of generic constant cannot be redefined by our system, but it is beyond the ability of

Table 8.3: Frequent Usages of Generic Constant (GC) Names

Name (output_*.tex)	GC Names	Actual Types		
		X	Y	Z
bbook	#1: domain	NAME	DATE	
	#2: domain1	NAME	ADDRESS	
fEmpty	#1: empty	\mathbb{N}		
	#2: empty1	\mathbb{Z}		
fEmptyImpl	#1: empty	\mathbb{N}		
	#2: empty1	\mathbb{Z}		
fFirst	#1: firstF	\mathbb{N}	\mathbb{N}	
	#2: firstF1	NAME	NAME	
	#3: firstF2	\mathbb{Z}	\mathbb{Z}	
	#1: secondF	\mathbb{N}	\mathbb{N}	
	#2: secondF1	NAME	NAME	
	#3: secondF2	\mathbb{Z}	\mathbb{Z}	
fHead	#1: headSeq	NAME		
	#2: headSeq1	\mathbb{N}		
	#3: headSeq2	\mathbb{Z}		
fHeadFunc	#1: headSeq	\mathbb{N}		
	#2: headSeq1	NAME		
fMaxComSubSeq	#1: allSubseq	\mathbb{N}		
	#1: commonSubseq	\mathbb{N}		
fMaxComSubSeq_1	#1: allSubseq	\mathbb{N}		
	#1: commonSubseq	\mathbb{N}		
fMaxComSubSeq_orig	#1: allSubseq	\mathbb{N}		
	#1: commonSubseq	\mathbb{N}		
fMonoSeq	#1: monoSequence	ALPHABET		
	#2: monoSequence1	\mathbb{Z}		
fMonoSeq_1	#1: monoSequence	ALPHABET		
	#2: monoSequence1	\mathbb{Z}		
fUniqSeq	#1: monoSequence	\mathbb{N}		
	#2: monoSequence1	\mathbb{Z}		
	#1: monoSequenceOne	\mathbb{N}		
	#2: monoSequenceOne1	\mathbb{Z}		
	#1: uniqSequence	\mathbb{N}		
	#2: uniqSequence1	\mathbb{Z}		
fUniq1Seq	#1: uniqSequence1	\mathbb{N}		
	#2: uniqSequence1	\mathbb{Z}		
fUniq2Seq	#1: uniqSequence1	\mathbb{N}		
	#2: uniqSequence1	\mathbb{Z}		
tn	#1: disjoint	PHONE		
tnImpl	#1: disjoint	PHONE		
fSet	#1: empty	\mathbb{N}_1		
	#2: notEqual	$\mathbb{P} \mathbb{N}_1$		

Table 8.4: Summary of a Usage of Generic Constant (GC) Names

Name (<code>output_*.tex</code>)	GC Names	Actual Types		
		X	Y	Z
bbook_map	mapRel	NAME	DATE	
bbook_map_uni	mapRel	NAME	DATE	
	uniSet	NAME \times DATE		
bbook_uni	uniSet	NAME \times DATE		
fDomRan	domainSet	GUEST	HOTELROOM	
	rangeSet	GUEST	HOTELROOM	
fMaxComSubSeq	subseqOf	\mathbb{N}		
	maxCommonSubseq	\mathbb{N}		
fMaxComSubSeq_1	subseqOf	\mathbb{N}		
	maxCommonSubseq	\mathbb{N}		
fMaxComSubSeq_origin	subseqOf	\mathbb{N}		
	maxCommonSubseq	\mathbb{N}		
fMonoSeq	monoSequenceOne	\mathbb{Z}		
fMonoSeq_1	monoSequenceOne	\mathbb{Z}		
fSwap	swap1	NAME		
	swap2	NAME	\mathbb{N}	
fFileStorage	after	0 .. 255		
	shift	0 .. 255		
fSet	uniSet	\mathbb{N}_1		
	nonMember	\mathbb{N}_1		

our system. The current performances of our system are sufficient to show a redefinition process over generic constant definitions found in Z specifications. The issue of more complex types of generic constants or more complex usages of these generic constants can be set up as future works.

8.3.2 Evaluation of the #2 Question

To assess the second question, previous experiments have provided proofs to answer this question. Thus, a user is encouraged to refer to associated section to get detail explanations.

Table 8.5 on page 216 gives us summaries of our experiments. These summaries record either Z specifications or generated SAL files. A term of 'modified SAL function' means there is a modification which was applied to a function on a generated SAL file. Thus, this modification relates to user defined functions. This modification was performed manually on associated SAL file.

On the other hand, a term of 'modified other parts of SAL file' means a manual modification has been performed manually on other parts on associated SAL file, not including the function. A term of 'OK' means there is no

modification performed on associated SAL file in order to make it was able to be executed.

Based on Table 8.5 on page 216, only one experiment which is the "22nd" one or `fFileStorage.tex` failed to be translated by Z2SAL. The reasons can be seen in the associated section.

To solve the problem found on `fFileStorage.tex`, a variable, which is a generic constant whose type was suspected as the source of an error generated by Z2SAL, was deleted from this Z specification. The associated generic constant definitions, the `writeSS` schema, and the `zero` axiomatic definition were also deleted.

However, Z2SAL still could not translate this Z specification. There was no error file generated and just a message:

Working on output_fFileStorage_1

displayed on Z2SAL's GUI instead.

Based on our investigation, Z2SAL failed to translate this Z specification since there is a definition of "." as follows:

BYTE == 0..255

Moreover, `BYTE` is used on many places on this Z specification.

Previously, there was a usage of a range of numbers on the Chapter 3. At first, Z2SAL raised an error, but Z2SAL could translate the associated Z specification file after the error was reported to Z2SAL's team. The range of numbers was used on this Z specification as follows:

count : 0..maxFiles

in which `maxFiles` was specified as follows:

<i>maxFiles</i> : \mathbb{N}_1	
<i>maxFiles</i> = 3	

Referring to the previous experience as given above, `BYTE` was modified then as follows:

BYTE == 0..maxByte

<i>maxByte</i> : \mathbb{N}	
<i>maxByte</i> = 255	

However, another error message was generated by Z2SAL and without any error file. The displayed message is:

Table 8.5: Summaries of Checked Files

Z Specification	Details
bbook.tex	Modified user defined function on SAL
bbook_map.tex	Modified user defined function on SAL
bbook_uni.tex	Modified user defined function on SAL and modified other parts of SAL file
bbook_map_uni.tex	Modified user defined function on SAL and modified other parts of SAL file
fDomRan.tex	Modified user defined function on SAL
fEmpty.tex	OK
fEmptyImpl.tex	OK
fFirst.tex	Modified user defined function on SAL
fHead.tex	Modified user defined function on SAL
fHeadFunc.tex	Modified user defined function on SAL and cannot be simulated: The set of initial states is empty
fMaxComSubSeq.tex	Modified other parts of SAL file and cannot be simulated: Out of memory error
fMaxComSubSeq_1.tex	Modified other parts of SAL file and cannot be simulated: Out of memory error
fMaxComSubSeq_orig.tex	Modified other parts of SAL file and cannot be simulated: Out of memory error
fMonoSeq.tex	OK and longer time to be simulated
fMonoSeq_1.tex	OK and longer time to be simulated
fSwap.tex	Modified user defined function on SAL
fUniqSeq.tex	OK and cannot be simulated because out of memory error
fUniq1Seq.tex	OK and cannot be simulated because out of memory error
fUniq2Seq.tex	OK and cannot be simulated because out of memory error
tn.tex	Modified other parts of SAL file and cannot be simulated because out of memory error
tnImpl.tex	Modified other parts of SAL file and cannot be simulated because out of memory error
fFileStorage.tex	Cannot be translated by Z2SAL
fSet.tex	Modified user defined function on SAL and modified other parts of SAL file

The same message was displayed though the number 255 has been changed to a small number such as 3. Since then, this Z specification was left unresolved.

Thus, the rests of Z specifications could be translated by Z2SAL. Several of their SAL files could be executed directly by the SAL tool without any modification. Examples are the 6th, 7th, 14th, and 15th experiments or their names of Z specifications are `fEmpty.tex`, `fEmptyImpl.tex`, `fMonoSeq_1.tex`, and `fMonoSeq.tex` respectively. However, the last two examples required longer times for simulation. Other examples such as the 17th, 18th, and 19th experiments or `fUniqSeq.tex`, `fUniq1Seq.tex`, and `fUniq2Seq.tex` for their names respectively failed to be simulated since there were out of memory errors.

Other SAL files required alterations which were performed manually so that they could be executed by the SAL tool. These alterations were performed either on SAL functions or other parts of associated SAL files. More information about these can be read on associated section.

Therefore, the second question can be answered as follows. Yes, the outcomes of this system could be translated by Z2SAL and executed by the SAL tool with several limitations. As long as the Z specification does not contain a type whose range is a function, and a type of a range of numbers, these outcomes could be translated by Z2SAL and executed by the SAL tool. Furthermore, out of memory errors generated by the SAL simulator seems can be avoided if there is no sequence on such a SAL file especially abundant usages of sequences. Nested sets also seem hindered successful executions by the SAL tool.

8.3.3 Evaluation of the #3 Question

Z specifications used for our experiments have different sizes measured in kilobytes. These sizes are summarized in Table 8.6 on page 218. Sizes of the redefined specifications are recorded also on this table.

'input' on this table means a Z specification input file for our system. On the other hand, 'output' means a Z specification output file generated by our system after performing a redefinition process, 'N/A's in **SAL specifications** means an associated Z specification could not be translated by Z2SAL.

Referring to this table, almost all of our experiments have the same sizes of Z specifications before and after redefinition processes. It means that there

Table 8.6: Sizes of Z Specifications

Z Specifications (.tex)	Sizes in KB		
	input	output	SAL specifications
bbook	2	2	6
bbook_map	1	1	4
bbook_map_uni	1	2	5
bbook_uni	1	1	4
fDomRan	2	2	6
fEmpty	1	1	2
fEmptyImpl	1	1	2
fFirst	1	1	3
fHead	1	1	3
fHeadFunc	1	1	3
fMaxComSubSeq	2	2	4
fMaxComSubSeq_1	2	2	4
fMaxComSubSeq_orig	2	2	4
fMonoSeq	1	1	3
fMonoSeq_1	1	1	3
fSwap	1	1	2
fUniqSeq	1	2	5
fUniq1Seq	1	2	5
fUniq2Seq	1	2	5
tn	3	3	6
tnImpl	3	3	6
fFileStorage	2	2	N/A
fSet	2	2	5

were not many usages specified in these specifications. It can also mean that the generic constant definitions are not quite complex definitions.

On the other hand, a size of a SAL specification is roughly twice to four times of its Z specification. Sizes of SAL specifications shown in this table are original sizes producing by Z2SAL. As discussed above, several of these SAL specifications have been modified as required in order to be executed by the SAL tool successfully. Thus, their sizes can be different from the original ones.

There are only four experiments which their sizes of Z specification outputs were increased twice of their inputs. These specifications are `bbook_map_uni`, `fUniqSeq`, `fUniq1Seq` and `fUniq2Seq`.

Based on previous discussions on each experiment, the last three Z specifications above could not be simulated by the SAL simulator because of out of memory errors. However, these errors cannot be blamed for the increasing sizes of specifications. It is because there are other specifications which their sizes were not increased, but they were involved on the same errors as above.

Sizes of these specifications are greater than 1. However, it can coincide which is not influenced entirely only by sizes of specifications.

Other consideration here is a complexity of a declaration of a generic constant. The out of memory errors were involved on specifications which have either sequences, or sets of sets.

Thus, it seems that a Z specification, which does not have a sequence, a set of other set, or a range of numbers, can be executed successfully by the SAL tool. It argues also that a size of a generic constant definition and a number of usages relate to a generation of that error.

As a conclusion, our approach to redefine generic constants definitions scales to larger specifications. However, as the outcomes of our system will be translated by Z2SAL and executed by the SAL tool later, the large specification resulted by our system is possible to be a problem with both tools.

8.4 Conclusion

Based on our experiments, our system could redefine generic constant definitions on Z specification inputs to some extent. This system could also generate the outcomes which sometimes could be translated by Z2SAL and be executed by the SAL tool with some limitations. Our system cannot possibly cope with more complex types of generic constants, more complex predicates of a generic constant definition, or more complex usages of such a generic constant.

Nevertheless, having a generic constant definition redefined, Z2SAL will be benefited from a variety of Z specifications. Thus, coverage of Z2SAL can be also broadened.

As a conclusion, this system is worth to consider, especially if it can be extended so it can redefine more complex generic constant definitions specified in a Z specification. A more complex generic constant definition means several conditions. It can be a more complex type of a generic constant variable. It can also be a more complex predicate part of this definition.

Another important issue to consider is our approach on translating functions on SAL files. Although our approach on this SAL translation is still a manual work, this approach can be considered also to automate in future. This automation can either be added to the Z2SAL system or an extension to our system. However, it seems the first option is easier to implement.

Related to the out of memory error, it requires further works. Our easy solution that was applied to this issue is to change sizes of several related variables smaller. Although there is high performance computing machines on our university, there is no chance to benefit from those machines. Another

potential solution to this problem is to apply abstraction on associated SAL file.

Chapter 9

A Schema Calculus Expansion

This chapter discusses another type of our support for model checking Z specification which is support for schema calculus. This support is a work in our pre-processing tool as given in Fig. 3.1. The chapter consists of several sections. It begins with setting up questions for an evaluation on this system. These questions represent a few scales to measure performances of our system. It is followed by our experiments on this system. These experiments present expansions of schema calculus performed by our system. An evaluation based on these questions is discussed on the next section. A conclusion ends this chapter.

9.1 Setting up Questions for an Evaluation

In line with our objective mentioned on an earlier chapter, a system for expanding schema calculus has been implemented. Having this system, an evaluation is necessary to conduct to know its performances. In order to evaluate this system, several questions were set up. These questions are as follows:

- Can this system expand all schema calculus definitions correctly?
- Can the outcome of this system be translated by Z2SAL and then be executed by the SAL tool?
- Does the approach scale to larger specifications?

Before an evaluation can be performed, several experiments with this system are discussed on the following section. These experiments test the ability of our system to expand schema calculus in Z specifications.

9.2 Experiments with the Schema Calculus Definitions

Several experiments which were performed for our support on schema calculus will be presented in this section. These experiments are summarized in Table 9.1 on page 222 where 'hs' means a horizontal schema. The horizontal schema is represented by using a pair of "[" and "]"".

The order of our experiments is based on schema operators specified in a schema calculus definition. A simple schema operator will be introduced

Table 9.1: Details of Several Experiments with the Expansion System

Number of Schema Operators	Z Specification (.tex)	Details
1	expandingschema_1 (9.2.1 on page 224)	" \forall "
	expandingschema_2 (9.2.2 on page 227)	" \wedge "
	expandingschema_4 (9.2.4 on page 229)	" \wedge "
		" \forall "
	expandingschema_8 (9.2.8 on page 233)	" \forall "
	expandingsch2_4 (9.2.9 on page 234)	" \neg "
	expandingsch3_1 (9.2.10 on page 235)	" \Rightarrow "
	expandingsch4_1 (9.2.13 on page 237)	" \Leftrightarrow "
	expandingsch6_1 (9.2.17 on page 239)	" \setminus "
	expandingsch6_2 (9.2.18 on page 240)	" \setminus "
	expandingsch5_1 (9.2.15 on page 238)	" $/$ "
	expandingsch5_2 (9.2.16 on page 239)	" $/, /$ "
	expandingsch7_1 (9.2.19 on page 240)	" $\overset{0}{9}$ "
	expandingschema_3 (9.2.3 on page 228)	" $\overset{0}{9}$ "
	expandingsch8_1 (9.2.20 on page 241)	" \forall "
	expandingsch8_2 (9.2.21 on page 241)	" \forall "
expandingsch8_6 (9.2.23 on page 242)	" \exists "	
2	expandingschema_4 (9.2.4 on page 229)	" \forall, \forall "
	expandingsch3_2 (9.2.11 on page 236)	" \wedge, \Rightarrow "
	expandingsch3_4 (9.2.12 on page 236)	" \Rightarrow, \wedge "
	expandingsch4_2 (9.2.14 on page 238)	" \wedge, \Leftrightarrow "
	expandingsch8_3 (9.2.22 on page 242)	" \forall, \wedge "
3	expandingschema_5 (9.2.5 on page 231)	" \wedge, \neg, \wedge "
1 and hs	expandingschema_6 (9.2.6 on page 232)	" $\wedge, [,]$ "
	expandingschema_8 (9.2.8 on page 233)	" $\wedge, [,]$ "
2 and hs	expandingschema_7 (9.2.7 on page 233)	" $\neg, \wedge, [,]$ "
	expandingschema_8 (9.2.8 on page 233)	" $\neg, \wedge, [,]$ "

at the first experiment. Conjugation and disjunction are simpler than other operators.

Afterwards, other schema operators will be introduced as well as more complex schema calculus definitions. A more complex schema calculus definition is specified by using a combination of schema operators. It is also a more complex schema calculus definition if there are a combination of operators and a horizontal schema. Our experiments also experienced with a combination of operators and a horizontal schema which declared predicates.

On the other hand, experiments on a library system will begin our presentations of these experiments since this specification is quite a complete specification. There are other specification systems used for our experiments. They will be introduced as follows before they are used for the experiments.

Thus, our experiments as follows begin by showing how individual operators are expanded correctly. These operators are: " \vee ", " \wedge ", " \neg ", " \Rightarrow ", " \Leftrightarrow ", " \setminus ", " $/$ ", " \circ ", " \forall " and " \exists ".

Afterwards, our experiments also show how when they are combined they are still expanded correctly. These combinations of operators are: " \vee " and " \forall ", " \wedge " and " \Rightarrow ", " \Rightarrow " and " \wedge ", " \wedge " and " \Leftrightarrow ", " \forall " and " \wedge ", " \wedge ", " \neg " and " \wedge ", " \neg ", and " \wedge " and a horizontal schema.

Let us now move to discussions on these experiments. It begins with introductions of specification systems in general.

Specifications which were used for an Experiment 1 to an Experiment 8 were taken from [56]. It is a library system. This system has a state and an initialization schema as follows:

<p style="margin: 0;"><i>Library</i></p> <hr style="border: 0.5px solid black;"/> <p style="margin: 0;"><i>stock</i> : <i>COPY</i> \rightarrow <i>BOOK</i>; <i>issued</i> : <i>COPY</i> \leftrightarrow <i>READER</i> <i>shelved</i> : \mathbb{F} <i>COPY</i>; <i>readers</i> : \mathbb{F} <i>READER</i></p> <hr style="border: 0.5px solid black;"/> <p style="margin: 0;">$\forall x : \text{COPY}; y1, y2 : \text{READER} \bullet$ $(x \mapsto y1) \in \text{issued} \wedge (x \mapsto y2) \in \text{issued} \Rightarrow y1 = y2$ <i>shelved</i> \cup <i>dom issued</i> = <i>dom stock</i> <i>shelved</i> \cap <i>dom issued</i> = \emptyset <i>ran issued</i> \subseteq <i>readers</i> $\forall r : \text{readers} \bullet \#(\text{issued} \triangleright \{r\}) \leq \text{maxloans}$</p> <hr style="border: 0.5px solid black;"/>
<p style="margin: 0;"><i>InitLibrary</i></p> <hr style="border: 0.5px solid black;"/> <p style="margin: 0;"><i>Library'</i></p> <hr style="border: 0.5px solid black;"/> <p style="margin: 0;"><i>shelved'</i> = \emptyset <i>readers'</i> = \emptyset</p> <hr style="border: 0.5px solid black;"/>

On the other hand, Experiment 9 was taken from [49]. This is a simple car park system. The state schema and the initialization schema are as follows:

<i>CarsPark</i>
$count : \mathbb{N}; maximum : \mathbb{N}$
$count \leq maximum$

<i>InitCarsPark</i>
<i>CarsPark</i>
$count = 0$ $maximum = 3$

Experiment 10 to Experiment 14 and Experiment 23 were taken from [75]. This system regards with bookings for performances on a concert hall. The state schema and the initialization schema for these experiments are as follows:

<i>BoxOffice</i>
$seating : \mathbb{P} Seat$ $sold : Seat \rightarrow Customer$
$dom\ sold \subseteq seating$

<i>InitBoxOffice</i>
<i>BoxOffice'</i>
$sold' = \emptyset$ $seating' = initial_allocation$

The rest of experiments were taken from [2].

<i>Calculator</i>
$store : MEMORY \rightarrow \mathbb{Z}$ $display : \mathbb{Z}$ $arg2 : \mathbb{Z}$

<i>Init</i>
<i>Calculator</i>
$\forall m : MEMORY \bullet store(m) = 0$ $display = 0$ $arg2 = 0$

Above are the state and initialization schemas of this specification. This specification is a system of a four function calculator.

Each experiment is described on a separate sub-section. Results and discussions are also given on each sub-section.

Let us now move to each experiment. The first sequence of experiments is on the library system.

9.2.1 Experiment 1: expandingschema_1.tex

There is one schema calculus definition in this specification. It uses one schema operator, "V".

$$AddCopy \hat{=} AddKnownTitle \vee AddNewTitle$$

Associated operational schemas are as follows:

<i>AddKnownTitle</i>
$\Delta Library; b? : BOOK; rep! : Report$
$b? \in ran\ stock$ $\exists c : COPY \mid c \notin dom\ stock \bullet$ $(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\})$ $issued' = issued$ $readers' = readers$ $rep! = FurtherCopyAdded$

The above schema is to add a copy of a known book. It means that the book is available in the library. In other words, several books with this title exist in the library, but not a book whose copy is intended to add.

The below schema adds a new book to the library.

<i>AddNewTitle</i>
$\Delta Library; b? : BOOK; rep! : Report$
$ \begin{aligned} &b? \notin \text{ran } stock \\ &\exists c : COPY \mid c \notin \text{dom } stock \bullet \\ &(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\}) \\ &issued' = issued \\ &readers' = readers \\ &rep! = NewTitleAdded \end{aligned} $

A new book means neither its title nor its copy exists in the library.

The above schema calculus will create a new operational schema to this specification, namely **AddCopy**. As given by that definition, there are two cases of adding a copy of a book to the library system. The first case is a copy of an existing book. The second case is a copy of a new book.

The expanded **AddCopy**, which was generated by our system, is as follows:

<i>AddCopy</i>
$\Delta Library; b? : BOOK; rep! : Report$
$ \begin{aligned} &(b? \in \text{ran } stock \wedge \\ &(\exists c : COPY \mid c \notin \text{dom } stock \bullet \\ &(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\})) \wedge \\ &issued' = issued \wedge \\ &readers' = readers \wedge \\ &rep! = FurtherCopyAdded) \\ \vee \\ &(b? \notin \text{ran } stock \wedge \\ &(\exists c : COPY \mid c \notin \text{dom } stock \bullet \\ &(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\})) \wedge \\ &issued' = issued \wedge \\ &readers' = readers \wedge \\ &rep! = NewTitleAdded) \end{aligned} $

If the predicates are rearranged as follows:

$$\begin{aligned}
&((\exists c : COPY \mid c \notin \text{dom } stock \bullet \\
&(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\})) \wedge \\
&issued' = issued \wedge \\
&readers' = readers \wedge \\
&b? \in \text{ran } stock \wedge \\
&rep! = FurtherCopyAdded) \\
\vee \\
&((\exists c : COPY \mid c \notin \text{dom } stock \bullet \\
&(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\})) \wedge \\
&issued' = issued \wedge \\
&readers' = readers \wedge \\
&b? \notin \text{ran } stock \wedge \\
&rep! = NewTitleAdded)
\end{aligned}$$

then they form a pattern of a propositional logic formula as follows:

$$(p \wedge q) \vee (p \wedge r)$$

p represents the first, second, and third conjuncts before and after "∨", whereas q represents the four and fifth conjuncts, and r represents the last two conjuncts.

Based on the algebra of logical equivalences as shown also in [56], the above pattern has an equivalence form as follows:

$$p \wedge (q \vee r)$$

It is a distributivity law on AND.

Thus, our new operational schema can become as follows:

$\frac{\text{AddCopy}}{\Delta\text{Library}; b? : \text{BOOK}; \text{rep!} : \text{Report}}$ $(\exists c : \text{COPY} \mid c \notin \text{dom stock} \bullet$ $(\text{stock}' = \text{stock} \oplus \{c \mapsto b?\} \wedge \text{shelved}' = \text{shelved} \cup \{c\})) \wedge$ $\text{issued}' = \text{issued} \wedge$ $\text{readers}' = \text{readers} \wedge$ $((b? \in \text{ran stock} \wedge$ $\text{rep!} = \text{FurtherCopyAdded})$ \vee $(b? \notin \text{ran stock} \wedge$ $\text{rep!} = \text{NewTitleAdded}))$

The last conjunct can be simplified further since its form is as follows:

$$(p \wedge q) \vee (\neg p \wedge r)$$

p represents " $b? \in \text{ran stock}$ ", q represents " $\text{rep!} = \text{FurtherCopyAdded}$ ", and r represents " $\text{rep!} = \text{NewTitleAdded}$ ".

There is an equivalence that can be used to simplify it as follows:

$$(p \wedge q) \vee (\neg p \wedge r) \Leftrightarrow (p \Rightarrow q) \wedge (\neg p \Rightarrow r)$$

Thus, the simplified schema is as follows:

$\frac{\text{AddCopy}}{\Delta\text{Library}; b? : \text{BOOK}; \text{rep!} : \text{Report}}$ $\exists c : \text{COPY} \mid c \notin \text{dom stock} \bullet$ $(\text{stock}' = \text{stock} \oplus \{c \mapsto b?\} \wedge \text{shelved}' = \text{shelved} \cup \{c\})$ $\text{issued}' = \text{issued}$ $\text{readers}' = \text{readers}$ $b? \in \text{ran stock} \Rightarrow \text{rep!} = \text{FurtherCopyAdded}$ $b? \notin \text{ran stock} \Rightarrow \text{rep!} = \text{NewTitleAdded}$

However, this simplification was performed manually since our system has not been equipped with this capability. The simplified version of a schema

might require less time to be translated by Z2SAL as well as executed by the SAL tool.

Z2SAL could translate this expanded specification as well as the simplified one. The generated SAL file could also be verified by the SAL model checker in 0.063 second and 0.031 second total execution times without any theorem. The former time is dedicated for a non-simplified version and the latter time is for a simplified one. It could also be simulated by the SAL simulator.

The below theorem was added to this SAL file:

th1: theorem State |- G(shelved = set{COPY;}! empty);

Increasing total execution times occurred. The non-simplified SAL file required 0.92 second and the simplified one required 0.81 second.

9.2.2 Experiment 2: expandingschema_2.tex

There is one schema calculus definition, which uses one schema operator, " \wedge ". It is shown as follows:

$$AddCopy \hat{=} EnterNewCopy \wedge AddCopyReport$$

The generated schema has a quite similar function as the one from previous experiment. However, in this experiment, both schemas must exist to specify the AddCopy schema.

In contrast to the reference book, both schemas used in this specification have quite different declarations. It is since Z2SAL does not allow a re-declaration of state variables. Thus, both schemas were modified and it turned out that the AddCopyReport schema had a quite similar design to the EnterNewCopy schema.

Both schemas are as follows:

$$\begin{array}{l} \text{---} \\ \text{EnterNewCopy} \\ \Delta Library; b? : BOOK \\ \text{---} \\ \exists c : COPY \mid c \notin \text{dom } stock \bullet (stock' = stock \oplus \{c \mapsto b?\} \wedge \\ \text{shelved}' = \text{shelved} \cup \{c\}) \\ \text{issued}' = \text{issued} \\ \text{readers}' = \text{readers} \end{array}$$

The above schema performs an operation to enter a new copy to the library. On the other hand, the below schema is to release a report relating to an entered book either it is a new book or not.

$$\begin{array}{l} \text{---} \\ \text{AddCopyReport} \\ \exists Library; b? : BOOK; rep! : Report \\ \text{---} \\ b? \in \text{ran } stock \Rightarrow rep! = \text{FurtherCopyAdded} \\ b? \notin \text{ran } stock \Rightarrow rep! = \text{NewTitleAdded} \end{array}$$

The expanded schema was generated as follows:

<i>AddCopy</i>
Δ Library; $b? : BOOK$; $rep! : Report$
$((\exists c : COPY \mid c \notin \text{dom } stock \bullet$ $(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\})) \wedge$ $issued' = issued \wedge$ $readers' = readers)$ \wedge $(stock' = stock \wedge$ $issued' = issued \wedge$ $shelved' = shelved \wedge$ $readers' = readers \wedge$ $b? \in \text{ran } stock \Rightarrow rep! = FurtherCopyAdded \wedge$ $b? \notin \text{ran } stock \Rightarrow rep! = NewTitleAdded)$

The generated SAL file was verified by the SAL model checker in 0.062 second. It was without any theorem. If the same theorem as found on the previous experiment was added, the total execution time will be 0.78 second. The SAL simulator was also able to simulate this SAL file.

9.2.3 Experiment 3: expandingschema_3.tex

A schema calculus definition on this specification uses the schema composition operator, ” \circ ”. It is shown as follows:

$$Donate \cong EnterNewCopy \circ RegisterReader$$

The **EnterNewCopy** schema has been given on the previous experiment.

Another schema is as follows:

<i>RegisterReader</i>
Δ Library; $b? : BOOK$; $r? : READER$; $rep! : Report$
$r? \notin readers \Rightarrow (readers' = readers \cup \{r?\} \wedge rep! = Ok)$ $r? \in readers \Rightarrow (readers' = readers \wedge rep! = ReaderAlreadyRegistered)$ $stock' = stock$ $issued' = issued$ $shelved' = shelved$

The above schema will register a reader either a new reader or not.

Our system generated a new schema as follows:

<i>Donate</i>
Δ Library; $b? : BOOK$; $r? : READER$; $rep! : Report$
$\exists c : COPY \mid c \notin \text{dom } stock \bullet$ $(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\}) \wedge$ $r? \notin readers \Rightarrow (readers' = readers \cup \{r?\} \wedge rep! = Ok) \wedge$ $r? \in readers \Rightarrow (readers' = readers \wedge rep! = ReaderAlreadyRegistered) \wedge$ $issued' = issued$

The schema has been simplified to become as presented above. This simplification was performed by our system.

The generated SAL file could be verified by the SAL model checker. It required 0.03 second and 0.733 second for verifying 0 and 1 theorem. The SAL simulator could also simulate this SAL file.

9.2.4 Experiment 4: expandingschema_4.tex

This specification has three schema calculus definitions specified in one non-box definition.

$$\begin{aligned}
NormalIssue &\hat{=} Issue \wedge Success \\
IssueError &\hat{=} AlreadyIssued \vee NotRegistered \vee HasMaxLoans \\
TotalIssue &\hat{=} NormalIssue \vee IssueError
\end{aligned}$$

Schema operators used here are " \wedge " and " \vee ". Associated operational schemas are as follows:

$ \begin{array}{l} \textit{Issue} \\ \hline \Delta\textit{Library} \\ c? : \textit{COPY}; r? : \textit{READER} \\ \hline c? \in \textit{shelved} \\ r? \in \textit{readers} \\ \#(\textit{issued} \triangleright \{r?\}) < \textit{maxloans} \\ \textit{issued}' = \textit{issued} \oplus \{c? \mapsto r?\} \\ \textit{stock}' = \textit{stock} \\ \textit{readers}' = \textit{readers} \end{array} $	$ \begin{array}{l} \textit{Success} \\ \hline rep! : \textit{Report} \\ \hline rep! = \textit{Ok} \end{array} $
$ \begin{array}{l} \textit{NotRegistered} \\ \hline \exists\textit{Library} \\ r? : \textit{READER}; rep! : \textit{Report} \\ \hline r? \notin \textit{readers} \\ rep! = \textit{ReaderNotRegistered} \end{array} $	$ \begin{array}{l} \textit{AlreadyIssued} \\ \hline \exists\textit{Library} \\ c? : \textit{COPY}; rep! : \textit{Report} \\ \hline c? \in \textit{dom issued} \\ rep! = \textit{CopyAlreadyIssued} \end{array} $
$ \begin{array}{l} \textit{HasMaxLoans} \\ \hline \exists\textit{Library}; r? : \textit{READER}; rep! : \textit{Report} \\ \hline r? \in \textit{readers} \\ \#(\textit{issued} \triangleright \{r?\}) = \textit{maxloans} \\ rep! = \textit{ReaderHasMaxLoans} \end{array} $	

Several of those schemas have been modified to get them to obey conditions that were defined by Z2SAL, such as does not declare any state variable, and just has one state schema in a specification.

The first schema calculus created a schema as follows:

<i>NormalIssue</i>
$\Delta \text{Library}; c? : \text{COPY}; r? : \text{READER}; \text{rep!} : \text{Report}$
$ \begin{aligned} &(c? \in \text{shelved} \wedge \\ &r? \in \text{readers} \wedge \\ &\#(\text{issued} \triangleright \{r?\}) < \text{maxloans} \wedge \\ &\text{issued}' = \text{issued} \oplus \{c? \mapsto r?\} \wedge \\ &\text{stock}' = \text{stock} \wedge \\ &\text{readers}' = \text{readers}) \\ &\wedge \\ &(\text{rep!} = \text{Ok}) \end{aligned} $

This schema performs a successful issue of a copy of a book to a reader.

The second schema calculus produced the new operational schema as follows:

<i>IssueError</i>
$\Xi \text{Library}; c? : \text{COPY}; \text{rep!} : \text{Report}; r? : \text{READER}$
$ \begin{aligned} &(c? \in \text{dom issued} \wedge \text{rep!} = \text{CopyAlreadyIssued}) \vee \\ &(r? \notin \text{readers} \wedge \text{rep!} = \text{ReaderNotRegistered}) \vee \\ &(r? \in \text{readers} \wedge \#(\text{issued} \triangleright \{r?\}) = \text{maxloans} \wedge \\ &\text{rep!} = \text{ReaderHasMaxLoans}) \end{aligned} $

The above schema performs a fail in issuing a copy of a book to a reader. This fail can be the issue has been performed, the reader was not registered, or a number of issues of such the reader exceed the maximum loans.

The third schema calculus was created from both created schemas above. This schema is shown as follows:

<i>TotalIssue</i>
$\Delta \text{Library}; c? : \text{COPY}; r? : \text{READER}; \text{rep!} : \text{Report}$
$ \begin{aligned} &((c? \in \text{shelved} \wedge \\ &r? \in \text{readers} \wedge \\ &\#(\text{issued} \triangleright \{r?\}) < \text{maxloans} \wedge \\ &\text{issued}' = \text{issued} \oplus \{c? \mapsto r?\} \wedge \\ &\text{stock}' = \text{stock} \wedge \\ &\text{readers}' = \text{readers}) \\ &\wedge \\ &(\text{rep!} = \text{Ok})) \\ &\vee \\ &((\text{stock}' = \text{stock} \wedge \\ &\text{issued}' = \text{issued} \wedge \\ &\text{shelved}' = \text{shelved} \wedge \\ &\text{readers}' = \text{readers} \wedge \\ &c? \in \text{dom issued} \wedge \\ &\text{rep!} = \text{CopyAlreadyIssued}) \\ &\vee \\ &(\text{r?} \notin \text{readers} \wedge \text{rep!} = \text{ReaderNotRegistered}) \\ &\vee (\text{r?} \in \text{readers} \wedge \#(\text{issued} \triangleright \{r?\}) = \text{maxloans} \wedge \\ &\text{rep!} = \text{ReaderHasMaxLoans})) \end{aligned} $

The above schema describes that an issue can be a normal issue or an error one.

All above schemas were generated by our system. The generated SAL file could be verified by the SAL model checker. It required 0.016 second for no theorem and 2.044 seconds to verify one theorem. The SAL simulator could also simulate this SAL file.

9.2.5 Experiment 5: expandingschema_5.tex

There is one schema calculus definition in this specification. It uses three schema operators as follows:

$$\text{RemoveCopy} \cong \text{InStock} \wedge \neg \text{OnLoan} \wedge \text{FromStock}$$

Those three operational schemas are as follows:

$\frac{\text{InStock} \quad \Xi \text{Library} \quad c? : \text{COPY}}{c? \in \text{dom stock}}$	$\frac{\text{OnLoan} \quad \Xi \text{Library} \quad c? : \text{COPY}}{c? \in \text{dom issued}}$
$\frac{\text{FromStock} \quad \Delta \text{Library}; c? : \text{COPY}}{\text{stock}' = \{c?\} \triangleleft \text{stock} \quad \text{readers}' = \text{readers} \quad \text{issued}' = \text{issued}}$	

These three schemas were specified horizontally in [56]. Furthermore, re-declarations state variables and references to schemas other than the state schema have been removed from these schemas.

Following is the new schema created from the above schema calculus definition which was generated by our system:

$\frac{\text{RemoveCopy} \quad \Delta \text{Library}; c? : \text{COPY}}{\begin{aligned} & (c? \in \text{dom stock}) \wedge ((\forall x : \text{COPY}; y1, y2 : \text{READER} \bullet \\ & \neg ((x \mapsto y1) \in \text{issued} \wedge (x \mapsto y2) \in \text{issued} \Rightarrow y1 = y2)) \vee \\ & (\forall x : \text{COPY}; y1, y2 : \text{READER} \bullet \\ & \neg ((x \mapsto y1) \in \text{issued}' \wedge (x \mapsto y2) \in \text{issued}' \Rightarrow y1 = y2)) \vee \\ & \neg (\text{shelved} \cup \text{dom issued} = \text{dom stock}) \vee \\ & \neg (\text{shelved}' \cup \text{dom issued}' = \text{dom stock}') \vee \\ & \neg (\text{shelved} \cap \text{dom issued} = \emptyset) \vee \\ & \neg (\text{shelved}' \cap \text{dom issued}' = \emptyset) \vee \\ & \neg (\text{ran issued} \subseteq \text{readers}) \vee \\ & \neg (\text{ran issued}' \subseteq \text{readers}') \vee \\ & (\forall r : \text{readers} \bullet \neg (\#\text{issued} \triangleright \{r\} \leq \text{maxloans})) \vee \\ & (\forall r : \text{readers}' \bullet \neg (\#\text{issued}' \triangleright \{r\} \leq \text{maxloans})) \vee \\ & \neg (c? \in \text{dom issued}) \\ & \wedge \\ & (\text{stock}' = \{c?\} \triangleleft \text{stock} \wedge \\ & \text{readers}' = \text{readers} \wedge \text{issued}' = \text{issued}) \end{aligned}}$
--

The above schema performs the removal of a copy of a book from this library. A copy of books can be removed from a stock of books belonging to this library so long this copy is in the stock and it is not issued to any reader.

Z2SAL was able to translate this expanded schema. The generated SAL file without theorem could be verified by the SAL model checker in 0.031 second. If the same theorem as previous experiment was added, the SAL model checker required 1.654 seconds to execute this SAL file. The SAL simulator could also simulate this SAL file.

9.2.6 Experiment 6: expandingschema_6.tex

The schema calculus definition on this specification is as follows:

$$OnLoanError \cong OnLoan \wedge \left[rep! : Report \mid rep! = CopyOnLoan \right]$$

This new schema performs operations from the `OnLoan` schema which is then added with a new operation to display an error report. The error report informs a reader that a copy of a book has already been in an issue. In other words, this copy is on loan to other reader.

The right hand side of conjunction operator in the above schema calculus definition is similar to the ordinary horizontal schema that Z2SAL supports. However, in the above schema calculus definition, this horizontal schema is operated by the conjunction operator to another operational schema, `OnLoan`. This schema calculus as a whole cannot be translated by Z2SAL.

The `OnLoan` schema is shown as follows:

$$\frac{\frac{OnLoan}{\exists Library; c? : COPY}}{c? \in \text{dom } issued}$$

The created schema which was generated by our system is as follows:

$$\frac{\frac{OnLoanError}{\exists Library; c? : COPY; rep! : Report}}{(c? \in \text{dom } issued) \wedge (rep! = CopyOnLoan)}$$

The expanded specification could be translated by Z2SAL. The generated SAL file could be verified by the SAL model checker. This SAL file without theorem required 0.031 second and 0.686 second to be verified by the SAL model checker. It could be simulated by the SAL simulator.

9.2.7 Experiment 7: expandingschema_7.tex

The schema calculus definition on this specification is quite similar with the previous experiment. However, this time, the previous schema is negated before it is conjunct with a new variable and a new predicate.

$$\text{NotInStockError} \hat{=} \neg \text{InStock} \wedge \left[\text{rep!} : \text{Report} \mid \text{rep!} = \text{CopyNotInStock} \right]$$

The **InStock** schema can be seen in the previous experiment.

The generated schema is as follows:

$\begin{array}{l} \text{NotInStockError} \\ \text{stock} : \mathbb{P}(\text{COPY} \times \text{BOOK}); \text{stock}' : \mathbb{P}(\text{COPY} \times \text{BOOK}) \\ \text{issued} : \text{COPY} \leftrightarrow \text{READER}; \text{issued}' : \text{COPY} \leftrightarrow \text{READER} \\ \text{shelved} : \mathbb{F} \text{COPY}; \text{shelved}' : \mathbb{F} \text{COPY}; \text{readers} : \mathbb{F} \text{READER} \\ \text{readers}' : \mathbb{F} \text{READER}; c? : \text{COPY}; \text{rep!} : \text{Report} \\ \\ ((\forall x : \text{COPY}; y1, y2 : \text{READER} \bullet \\ \neg ((x \mapsto y1) \in \text{issued} \wedge (x \mapsto y2) \in \text{issued} \Rightarrow y1 = y2)) \vee \\ (\forall x : \text{COPY}; y1, y2 : \text{READER} \bullet \\ \neg ((x \mapsto y1) \in \text{issued}' \wedge (x \mapsto y2) \in \text{issued}' \Rightarrow y1 = y2)) \vee \\ \neg (\text{shelved} \cup \text{dom issued} = \text{dom stock}) \vee \\ \neg (\text{shelved}' \cup \text{dom issued}' = \text{dom stock}') \vee \\ \neg (\text{shelved} \cap \text{dom issued} = \emptyset) \vee \\ \neg (\text{shelved}' \cap \text{dom issued}' = \emptyset) \vee \\ \neg (\text{ran issued} \subseteq \text{readers}) \vee \\ \neg (\text{ran issued}' \subseteq \text{readers}') \vee \\ (\forall r : \text{readers} \bullet \neg (\#\{\text{issued} \triangleright \{r\}\} \leq \text{maxloans})) \vee \\ (\forall r : \text{readers}' \bullet \neg (\#\{\text{issued}' \triangleright \{r\}\} \leq \text{maxloans})) \vee \\ \neg (c? \in \text{dom stock}) \vee \\ \neg (\text{stock} \in (\text{COPY} \rightarrow \text{BOOK})) \vee \\ \neg (\text{stock}' \in (\text{COPY} \rightarrow \text{BOOK}))) \\ \wedge \\ (\text{rep!} = \text{CopyNotInStock}) \end{array}$
--

However, this expanded specification could not be translated by Z2SAL. It is since there are several re-declared state variables on the created schema. These state variables in this expanded schema could not be collapsed to the reference of the state schema since the associated predicates were different; they have been negated.

9.2.8 Experiment 8: expandingschema_8.tex

This specification has several schema calculus definitions, though the first two definitions have been specified in the previous experiments. Another definition was specified uses other definitions.

$$\begin{array}{l} \text{OnLoanError} \hat{=} \text{OnLoan} \wedge \left[\text{rep!} : \text{Report} \mid \text{rep!} = \text{CopyOnLoan} \right] \\ \text{NotInStockError} \hat{=} \neg \text{InStock} \wedge \left[\text{rep!} : \text{Report} \mid \text{rep!} = \text{CopyNotInStock} \right] \\ \text{RemoveErrors} \hat{=} \text{OnLoanError} \vee \text{NotInStockError} \end{array}$$

The created schemas for the first two definitions can be seen in the previous experiments. The last created definition is as follows:

<i>RemoveErrors</i>
$\exists \text{Library}; c? : \text{COPY}; \text{rep!} : \text{Report}$
$ \begin{aligned} & ((c? \in \text{dom issued}) \\ & \wedge \\ & (\text{rep!} = \text{CopyOnLoan})) \\ & \vee (((\forall x : \text{COPY}; y1, y2 : \text{READER} \bullet \\ & \neg (((x \mapsto y1) \in \text{issued}) \wedge ((x \mapsto y2) \in \text{issued}) \Rightarrow y1 = y2)) \vee \\ & (\forall x : \text{COPY}; y1, y2 : \text{READER} \bullet \\ & \neg (((x \mapsto y1) \in \text{issued}') \wedge ((x \mapsto y2) \in \text{issued}') \Rightarrow y1 = y2)) \vee \\ & \neg (\text{shelved} \cup \text{dom issued} = \text{dom stock}) \vee \\ & \neg (\text{shelved}' \cup \text{dom issued}' = \text{dom stock}') \vee \\ & \neg (\text{shelved} \cap \text{dom issued} = \emptyset) \vee \\ & \neg (\text{shelved}' \cap \text{dom issued}' = \emptyset) \vee \\ & \neg (\text{ran issued} \subseteq \text{readers}) \vee \\ & \neg (\text{ran issued}' \subseteq \text{readers}') \vee \\ & (\forall r : \text{readers} \bullet \neg (\#\text{issued} \triangleright \{r\}) \leq \text{maxloans})) \vee \\ & (\forall r : \text{readers}' \bullet \neg (\#\text{issued}' \triangleright \{r\}) \leq \text{maxloans})) \vee \\ & \neg (c? \in \text{dom stock}) \vee \\ & \neg (\text{stock} \in (\text{COPY} \mapsto \text{BOOK})) \vee \\ & \neg (\text{stock}' \in (\text{COPY} \mapsto \text{BOOK}))) \\ & \wedge \\ & (\text{rep!} = \text{CopyNotInStock}) \end{aligned} $

However, Z2SAL could not translate the expanded specification for the same reason as previous experiment.

9.2.9 Experiment 9: expandingsch2_4.tex

There is one operational schema used in a schema calculus definition. The operational schema and the schema calculus definition as follows:

<i>Enters</i>
$\Delta \text{CarsPark}$
$ \begin{aligned} & \text{count} < \text{maximum} \\ & \text{count}' = \text{count} + 1 \\ & \text{maximum}' = \text{maximum} \end{aligned} $

$$\text{NotEntered} \hat{=} (\neg \text{Enters})$$

The above schema calculus definition was specified by us.

There is one schema operator specified in this schema calculus definition. It is a negation operator.

Our system generated an expanded schema as follows:

<i>NotEntered</i>
$count : \mathbb{Z}; count' : \mathbb{Z}; maximum : \mathbb{Z}; maximum' : \mathbb{Z}$
$(\neg (count \leq maximum) \vee$ $\neg (count' \leq maximum') \vee$ $\neg (count < maximum) \vee$ $\neg (count' = count + 1) \vee$ $\neg (maximum' = maximum) \vee$ $\neg (count \in (\mathbb{N})) \vee$ $\neg (count' \in (\mathbb{N})) \vee$ $\neg (maximum \in (\mathbb{N})) \vee$ $\neg (maximum' \in (\mathbb{N})))$

However, this expanded specification could not be translated by Z2SAL because of redeclaring state or global variables. Due to negated predicates, state variables in this expanded schema could not collapse to a reference of the state schema.

9.2.10 Experiment 10: expandingsch3_1.tex

Two operational schemas that were used in the schema calculus definitions are as follows:

<i>NotPossible</i>	<i>Failure</i>
$\exists BoxOffice; s? : Seat$	$r! : Response$
$c? : Customer$	
$s? \mapsto c? \notin sold$	$r! = sorry$

The schema calculus definition was specified by us as follows:

$$Return \cong Failure \Rightarrow NotPossible$$

Our system created the **Return** expanded schema by using the equivalence form of an implication operator. This method was used at first for the sake of easiness since the equivalence form uses simpler operators. However, it turned out later that our system failed to expand this operator in a case of a more complex schema calculus definition.

The created schema is as follows:

<i>Return</i>
$\exists BoxOffice; r! : Response; s? : Seat; c? : Customer$
$(\neg (r! = sorry))$ \vee $(s? \mapsto c? \notin sold)$

The generated SAL file could be verified by the SAL model checker. Total execution time was 0.015 second for executing the SAL file without any theorem. The SAL simulator could also simulate this SAL file.

9.2.11 Experiment 11: expandingsch3_2.tex

There was one schema calculus definition specified as follows:

$$Return \hat{=} (Return0 \wedge Failure) \Rightarrow NotPossible$$

Failure and **NotPossible** schemas can be seen in the previous experiment. The **Return0** schema is as follows:

$\frac{Return0}{\Delta BoxOffice; s? : Seat; c? : Customer}$
$\begin{aligned} s? \mapsto c? \in sold \\ sold' = sold \setminus \{s? \mapsto c?\} \\ seating' = seating \end{aligned}$

The expanded specification is as follows:

$\frac{Return}{\exists BoxOffice; s? : Seat; c? : Customer; r! : Response}$
$\begin{aligned} &(\neg (\text{dom } sold \subseteq seating) \vee \\ &\neg (\text{dom } sold' \subseteq seating') \vee \\ &\neg (s? \mapsto c? \in sold) \vee \\ &\neg (sold' = sold \setminus \{s? \mapsto c?\}) \vee \\ &\neg (seating' = seating) \vee \\ &\neg (sold \in (Seat \rightarrow Customer)) \vee \\ &\neg (sold' \in (Seat \rightarrow Customer))) \\ &\vee \\ &(\neg (r! = sorry)) \\ &\vee \\ &(seating' = seating \wedge \\ &sold' = sold \wedge \\ &s? \mapsto c? \notin sold) \end{aligned}$

The generated SAL file with no theorem required 0.032 second to be executed by the SAL model checker. It could be simulated by the SAL simulator.

9.2.12 Experiment 12: expandingsch3_4.tex

A schema calculus definition which was specified for this experiment has the same operational schemas as the previous experiment. Differences are on an order of operator and usages of brackets.

The schema calculus definition was specified by us is as follows:

$$Return \hat{=} Return0 \Rightarrow NotPossible \wedge Failure$$

The " \wedge " operator is tighter than the " \Rightarrow " operator. Thus, the **Return** schema will be conjuncted with the **NotPossible** schema. Our system generated an expanded schema as follows:

<i>Return</i>
$\exists \text{BoxOffice}; s? : \text{Seat}; c? : \text{Customer}; r! : \text{Response}$
$(\neg (\text{dom } \text{sold} \subseteq \text{seating}) \vee$ $\neg (\text{dom } \text{sold}' \subseteq \text{seating}') \vee$ $\neg (s? \mapsto c? \in \text{sold}) \vee$ $\neg (\text{sold}' = \text{sold} \setminus \{s? \mapsto c?\}) \vee$ $\neg (\text{seating}' = \text{seating}) \vee$ $\neg (\text{sold} \in (\text{Seat} \leftrightarrow \text{Customer})) \vee$ $\neg (\text{sold}' \in (\text{Seat} \leftrightarrow \text{Customer})))$ $\vee (\text{seating}' = \text{seating} \wedge \text{sold}' = \text{sold} \wedge s? \mapsto c? \notin \text{sold})$ $\wedge (r! = \text{sorry})$

The expanded schema could be translated by Z2SAL. The generated SAL file with no theorem could be verified by the SAL model checker in 0.016 second. It could also be simulated by the SAL simulator.

9.2.13 Experiment 13: expandingsch4_1.tex

A schema calculus definition which was specified by us is as follows:

$$\text{Return} \cong \text{Return0} \Leftrightarrow \text{Failure}$$

It will be expanded as follows by our system:

<i>Return</i>
$\Delta \text{BoxOffice}; s? : \text{Seat}; c? : \text{Customer}; r! : \text{Response}$
$((\neg (\text{dom } \text{sold} \subseteq \text{seating}) \vee$ $\neg (\text{dom } \text{sold}' \subseteq \text{seating}') \vee$ $\neg (s? \mapsto c? \in \text{sold}) \vee$ $\neg (\text{sold}' = \text{sold} \setminus \{s? \mapsto c?\}) \vee$ $\neg (\text{seating}' = \text{seating}) \vee$ $\neg (\text{sold} \in (\text{Seat} \leftrightarrow \text{Customer})) \vee$ $\neg (\text{sold}' \in (\text{Seat} \leftrightarrow \text{Customer})))$ \vee $(r! = \text{sorry}))$ \wedge $((\neg (r! = \text{sorry}))$ \vee $(s? \mapsto c? \in \text{sold} \wedge$ $\text{sold}' = \text{sold} \setminus \{s? \mapsto c?\} \wedge$ $\text{seating}' = \text{seating}))$

As can be seen from the above schema, in expanding " \Leftrightarrow " schema operator, this operator was replaced by its equivalence operators. This has been explained on previous chapter. However, on more complex schema calculus definition, our system still cannot expand it.

The generated SAL file was executed in 0.015 second by the SAL model checker. It could also be simulated by the SAL simulator.

9.2.14 Experiment 14: expandingsch4_2.tex

The only schema definition which was specified by us is as follows:

$$Return \triangleq (NotPossible \wedge Success) \Leftrightarrow Return0$$

The **Success** schema is as follows:

$\frac{Success}{r! : Response}$
$r! = okay$

The expanded schema which was specified by our system is as follows:

$\frac{Return}{\Delta BoxOffice; s? : Seat; c? : Customer; r! : Response}$
$((\neg (\text{dom } sold \subseteq seating) \vee$ $\neg (\text{dom } sold' \subseteq seating') \vee$ $\neg (s? \mapsto c? \notin sold)) \vee$ $\vee (\neg (r! = okay)) \vee$ $(s? \mapsto c? \in sold \wedge$ $sold' = sold \setminus \{s? \mapsto c?\} \wedge seating' = seating))$ \wedge $((\neg (\text{dom } sold' \subseteq seating') \vee$ $\neg (s? \mapsto c? \in sold) \vee$ $\neg (sold' = sold \setminus \{s? \mapsto c?\}) \vee$ $\neg (seating' = seating)) \vee$ $((s? \mapsto c? \notin sold) \wedge (r! = okay)))$

Z2SAL could generate the SAL file from the expanded specification above. The SAL model checker could verify this SAL file without any theorem in 0.031 second. This SAL file could also be simulated by the SAL simulator.

9.2.15 Experiment 15: expandingsch5_1.tex

There is one schema calculus definition specified by us in this specification. The schema calculus definition uses a schema renaming operator. This operator was applied to the state schema as follows:

$$CalculatorI \triangleq Calculator[argument/arg2]$$

Our system generated an expanded schema as follows:

$\frac{CalculatorI}{store : \mathbb{P}(MEMORY \times \mathbb{Z}); display : \mathbb{Z}; argument : \mathbb{Z}}$
$store \in (MEMORY \rightarrow \mathbb{Z})$

However, the generated expanded specification could not be translated by Z2SAL. Redeclaring state or global variables was a source of the error. The renamed state variable made all state variables declared in this expanded schema were not collapsed to a reference of the state schema.

9.2.16 Experiment 16: expandingsch5_2.tex

An operational schema which was used in a schema calculus definition specified in this experiment is as follows:

$\frac{\text{Add}}{\Delta \text{Calculator}}$
$\begin{array}{l} \text{store}' = \text{store} \\ \text{display}' = \text{display} + \text{arg2} \end{array}$

The schema calculus definition is as follows:

$$\text{AddI} \cong \text{Add}[\text{argument}/\text{arg2}, \text{screen}/\text{display}]$$

As can be seen from the above definition, there are two variables which will be renamed from the **Add** schema.

Our system generated an expanded schema as follows:

$\frac{\text{AddI}}{\text{store} : \mathbb{P}(\text{MEMORY} \times \mathbb{Z}); \text{store}' : \mathbb{P}(\text{MEMORY} \times \mathbb{Z})}$
$\text{screen} : \mathbb{Z}; \text{display}' : \mathbb{Z}; \text{argument} : \mathbb{Z}; \text{arg2}' : \mathbb{Z}$
$\begin{array}{l} \text{store}' = \text{store} \wedge \\ \text{display}' = \text{screen} + \text{argument} \wedge \\ \text{store} \in (\text{MEMORY} \rightarrow \mathbb{Z}) \wedge \\ \text{store}' \in (\text{MEMORY} \rightarrow \mathbb{Z}) \end{array}$

This expanded specification could not be translated by Z2SAL because of the same error as previous experiment.

9.2.17 Experiment 17: expandingsch6_1.tex

This example uses the same operational schema as previous experiment. However, an operator used here is " \setminus ". Furthermore, the state schema used in this specification has been modified a bit by adding one state variable, **arg21**.

A schema calculus definition specified by us is as follows:

$$\text{AddI} \cong \text{Add} \setminus (\text{arg2})$$

This definition will hide the **arg2** variable, if any, from the **Add** schema.

$\frac{\text{AddI}}{\text{store} : \mathbb{P}(\text{MEMORY} \times \mathbb{Z}); \text{store}' : \mathbb{P}(\text{MEMORY} \times \mathbb{Z})}$
$\text{display} : \mathbb{Z}; \text{display}' : \mathbb{Z}; \text{arg2}' : \mathbb{Z}; \text{arg21} : \mathbb{Z}; \text{arg21}' : \mathbb{Z}$
$\begin{array}{l} \exists \text{arg2} : \mathbb{Z} \bullet \\ \text{store}' = \text{store} \wedge \text{display}' = \text{display} + \text{arg21} \wedge \\ \text{store} \in (\text{MEMORY} \rightarrow \mathbb{Z}) \wedge \text{store}' \in (\text{MEMORY} \rightarrow \mathbb{Z}) \end{array}$

The above schema from associated Z specification could not be translated by Z2SAL. The reason is redeclaring state or global variables. State variables declared in this schema could not be collapsed to a reference of the state schema since there is one hidden state variable.

9.2.18 Experiment 18: expandingsch6_2.tex

There are two hidden variables specified in this schema calculus definition.

$$AddI \hat{=} Add \setminus (arg2, arg21')$$

Our system generated an expanded schema as follows:

$\frac{AddI}{store : \mathbb{P}(MEMORY \times \mathbb{Z}); store' : \mathbb{P}(MEMORY \times \mathbb{Z})}$ $display : \mathbb{Z}; display' : \mathbb{Z}; arg2' : \mathbb{Z}; arg21 : \mathbb{Z}$
$\exists arg2 : \mathbb{Z}; arg21' : \mathbb{Z} \bullet$ $store' = store \wedge display' = display + arg2 \wedge$ $store \in (MEMORY \rightarrow \mathbb{Z}) \wedge store' \in (MEMORY \rightarrow \mathbb{Z})$

However, this expanded specification could not be translated by Z2SAL. The reason is the same as previous experiment.

9.2.19 Experiment 19: expandingsch7_1.tex

This specification uses the same state schema as the one used for the experiment 15. There is an operational schema that has not been used in previous experiments from the same state and initialization schemas. This operational schema is as follows:

$\frac{Enter}{\Delta Calculator; value? : \mathbb{Z}}$
$store' = store$ $display' = value?$ $arg2' = display$

There is one schema calculus definition specified in this specification as shown below:

$$Composition \hat{=} Enter \circ Add$$

One schema operator in this definition is the "o" schema composition.

Our system generated an expanded schema as follows:

$\frac{Composition}{\Delta Calculator; value? : \mathbb{Z}}$
$store' = store \wedge display' = value? + display$

This expanded specification, which looks like that after an automatic simplification by our system, could be translated by Z2SAL. It required 0.031 second to be verified by the SAL model checker on the SAL file with no theorem. It could also be simulated by the SAL simulator.

9.2.20 Experiment 20: expandingsch8_1.tex

One schema calculus definition was specified in this specification.

$$\text{Subtract} \triangleq \forall \text{arg2} : \mathbb{Z} \mid \text{arg2} < 0 \bullet \text{Add}$$

This specification uses the universal quantifier operator, "∀". The Add schema, which was applied to that operator, can be seen in the previous experiment.

Our system generated an expanded schema as follows:

$\begin{array}{l} \text{Subtract} \\ \text{store} : \mathbb{P}(\text{MEMORY} \times \mathbb{Z}); \text{store}' : \mathbb{P}(\text{MEMORY} \times \mathbb{Z}) \\ \text{display} : \mathbb{Z}; \text{display}' : \mathbb{Z}; \text{arg2}' : \mathbb{Z} \end{array}$
$\begin{array}{l} \forall \text{arg2} : \mathbb{Z} \mid \text{arg2} < 0 \bullet \\ \text{store}' = \text{store} \wedge \text{display}' = \text{display} + \text{arg2} \wedge \\ \text{store} \in (\text{MEMORY} \rightarrow \mathbb{Z}) \wedge \text{store}' \in (\text{MEMORY} \rightarrow \mathbb{Z}) \end{array}$

However, the expanded specification could not be translated by Z2SAL. The error is redeclaring state or global variables. One hidden state variable enforces all these state variables to be listed as their existences before.

9.2.21 Experiment 21: expandingsch8_2.tex

There is one schema calculus definition specified in this specification, the same as previous experiment. However, in this experiment there are two state variables will be hidden. This definition is given as follows:

$$\text{Subtract} \triangleq \forall \text{arg2}, \text{display}' : \mathbb{Z} \mid \text{arg2} < 0 \bullet \text{Add}$$

It has one schema operator which is "∀". This operator is applied to the Add schema.

Our system generated an expanded schema as follows:

$\begin{array}{l} \text{Subtract} \\ \text{store} : \mathbb{P}(\text{MEMORY} \times \mathbb{Z}); \text{store}' : \mathbb{P}(\text{MEMORY} \times \mathbb{Z}) \\ \text{display} : \mathbb{Z}; \text{arg2}' : \mathbb{Z} \end{array}$
$\begin{array}{l} \forall \text{arg2}, \text{display}' : \mathbb{Z} \mid \text{arg2} < 0 \bullet \\ \text{store}' = \text{store} \wedge \text{display}' = \text{display} + \text{arg2} \wedge \\ \text{store} \in (\text{MEMORY} \rightarrow \mathbb{Z}) \wedge \text{store}' \in (\text{MEMORY} \rightarrow \mathbb{Z}) \end{array}$

This expanded specification could not be translated by Z2SAL. The error is redeclaring state or global variables. Two hidden state variables make un-collapsed state variables.

9.2.22 Experiment 22: expandingsch8_3.tex

One schema calculus definition was specified in this specification. It is as shown below:

$$\text{Subtract} \hat{=} \forall \text{arg2}, \text{display}' : \mathbb{Z} \mid \text{arg2} < 0 \bullet \text{Enter} \wedge \text{Add}$$

The above definition uses the universal quantifier, "∀", and conjunction operators.

At the first, both **Enter** and **Add** schemas will be operated by using "∧". A schema output of this conjunction will be operated further by using the quantifier as an operator.

$\begin{array}{l} \text{Subtract} \\ \text{store} : \text{MEMORY} \rightarrow \mathbb{Z}; \text{store}' : \text{MEMORY} \rightarrow \mathbb{Z} \\ \text{display} : \mathbb{Z}; \text{arg2}' : \mathbb{Z}; \text{value}' : \mathbb{Z} \end{array}$
$\begin{array}{l} \forall \text{arg2}, \text{display}' : \mathbb{Z} \mid \text{arg2} < 0 \bullet \\ (\text{store}' = \text{store} \wedge \text{display}' = \text{value}' \wedge \text{arg2}' = \text{display}) \\ \wedge \\ (\text{store}' = \text{store} \wedge \text{display}' = \text{display} + \text{arg2}) \end{array}$

Above is an expanded schema generated by our system. All state variables declared on this new schema could not be collapsed to a reference of a state schema. It is since two state variables, **arg2** and **display'**, have been hidden. As a result, the expanded specification could not be translated by Z2SAL.

9.2.23 Experiment 23: expandingsch8_6.tex

There is one schema calculus definition specified in this specification. This definition uses the existential quantifier, "∃", as a schema operator. It is shown as follows:

$$\text{AnonymousReturn} \hat{=} \exists c? : \text{Customer} \bullet \text{Return0}$$

Our system generated an expanded schema as follows:

$\begin{array}{l} \text{AnonymousReturn} \\ \text{seating} : \mathbb{P} \text{Seat}; \text{seating}' : \mathbb{P} \text{Seat} \\ \text{sold} : \mathbb{P}(\text{Seat} \times \text{Customer}); \text{sold}' : \mathbb{P}(\text{Seat} \times \text{Customer}) \\ s? : \text{Seat} \end{array}$
$\begin{array}{l} \exists c? : \text{Customer} \bullet \text{dom sold} \subseteq \text{seating} \wedge \\ \text{dom sold}' \subseteq \text{seating}' \wedge s? \mapsto c? \in \text{sold} \wedge \\ \text{sold}' = \text{sold} \setminus \{s? \mapsto c?\} \wedge \text{seating}' = \text{seating} \wedge \\ \text{sold} \in (\text{Seat} \mapsto \text{Customer}) \wedge \text{sold}' \in (\text{Seat} \mapsto \text{Customer}) \end{array}$

However, the expanded specification could not be translated by Z2SAL. The error relates to redeclaring state or global variables. It is since state variables could not be collapsed to a reference to a state schema. The existential quantifier hides several variables which in this case are state variables.

Table 9.2: Several Experiments with the Expansion System

Z Specification (.tex)	Number of Schema Calculus Definitions	Verification time in secs	
		Non-simplified	Simplified
expandingschema_1	1	0.063	0.031
expandingschema_2	1	0.062	
expandingschema_3	1	0.03	
		0.733	
expandingschema_5	1	0.031	
		1.654	
expandingschema_6	1	0.031	
		0.686	
expandingschema_7	1	N/A	
expandingsch2_4	1	N/A	
expandingsch3_1	1	0.015	
expandingsch3_2	1	0.032	
expandingsch3_4	1	0.016	
expandingsch4_1	1	0.015	
expandingsch4_2	1	0.031	
expandingsch5_1	1	N/A	
expandingsch5_2	1	N/A	
expandingsch6_1	1	N/A	
expandingsch6_2	1	N/A	
expandingsch7_1	1	0.031	
expandingsch8_1	1	N/A	
expandingsch8_2	1	N/A	
expandingsch8_3	1	N/A	
expandingsch8_6	1	N/A	
expandingschema_8	3	N/A	
expandingschema_4	3	0.016	
		2.044	

9.3 Evaluation of Schema Calculus Expansion

This section discusses evaluation on our expansion system. This section is divided into three sub-sections in which each section answers one of our questions. The first section as follows discusses the first question.

9.3.1 Evaluation of the #1 Question

Our first question wants to assess whether our system can expand all schema calculus definition correctly. As part of our evaluation on this issue, Table 9.1 on page 222 or Table 9.2 on page 243 shows several of our experiments with this system.

All these experiments were successfully expanded by our system correctly.

Please refer to each experiment above to get explanations of each expansion.

As mentioned above that not all our Z specifications containing schema calculus definitions were displayed in Table 9.1 on page 222 or Table 9.2 on page 243. It is because several other specifications contain schema calculus definitions which were specified by us. Thus, these definitions are less interesting than schema calculus definitions used for above experiments. Another reason is a few other specifications still could not be expanded by our system to correct expanded specifications.

From our 76 Z specifications in total including experiments above, there are two Z specifications which fell into that category. Both of these specifications were failed to expand by our system. These two specifications will be discussed as follows.

Failed Experiments

Both specifications which fail to be expanded by our system are discussed in this sub-section. Each of them will be introduced in a separate paragraph.

expandingsch7_2.tex The first specification is `expandingsch7_2`. The state and initialization schemas of this specification has been given above which is the booking system on a concert hall.

A schema calculus definition specified in this specification by us is given as follows:

$$Return \cong Purchase0 \circ Return0$$

The `Return0` schema has also been given earlier. The `Purchase0` schema is as follows:

$ \begin{array}{l} \textit{Purchase0} \\ s? : \textit{Seat}; c? : \textit{Customer}; \Delta \textit{BoxOffice} \\ \hline s? \in \textit{seating} \setminus \text{dom } \textit{sold} \\ \textit{sold}' = \textit{sold} \cup \{s? \mapsto c?\} \\ \textit{seating}' = \textit{seating} \end{array} $
--

The above schema calculus definition uses a schema composition operator. There are two other specifications in our experiments which have this operator. However, both of these specifications could be expanded correctly by our system. It is because they contain simpler predicates than schemas specified in the above schema calculus definition. Our system only can identify an equal operator as a connector of a left and right hand side operands. In contrast to both schemas, they have "∈" instead.

Our system generated an expanded specification as follows:

<i>Return</i>
$\Delta\text{BoxOffice}; s? : \text{Seat}; c? : \text{Customer}$
$s? \in \text{seating} \setminus \text{dom sold} \wedge \text{seating}' = \text{seating}$

Inevitably, the above schema is not correct as the expanded one from the above schema calculus definition. The first predicate is not a correct predicate.

If a manual work is applied to that definition, the expanded schema is as follows:

<i>Return</i>
$\Delta\text{BoxOffice}; s? : \text{Seat}; c? : \text{Customer}$
$s? \mapsto c? \in \text{sold} \cup \{s? \mapsto c?\} \wedge$ $\text{sold}' = \text{sold} \cup \{s? \mapsto c?\} \setminus \{s? \mapsto c?\} \wedge$ $\text{seating}' = \text{seating}$

The first predicate is true so that this line can be deleted. The second line can be simplified by applying the operator "\". The final schema is as follows:

<i>Return</i>
$\Delta\text{BoxOffice}; s? : \text{Seat}; c? : \text{Customer}$
$\text{sold}' = \text{sold} \wedge$ $\text{seating}' = \text{seating}$

Based on this specification, our system could not simplify the predicates to the correct simplified ones. The first line of predicate requires simplification works which are not easy to implement. The automation of the above simplification can be put as a future work.

expandingsch3_10.tex This specification unfortunately could not be run by our system. The Java compiler raised an error running this specification. This specification also has the same state and initialization schemas as the above example. However, its schema calculus definition is rather complex.

$$\text{Return} \hat{=} (\neg \text{Return0} \vee \neg \text{Success}) \Rightarrow (\text{NotPossible} \wedge \text{Failure})$$

As can be seen in the above definition, negation, conjunction, disjunction, and implication were specified in it. The above schema calculus was specified by us.

In order to be able to locate the error, the definition from the operator "\Rightarrow" to the right has been deleted. Thus, the definition was as follows:

$$\text{Return} \hat{=} (\neg \text{Return0} \vee \neg \text{Success})$$

This simple definition could be expanded correctly by our system.

However, if this definition was changed as follows:

$$Return \cong (\neg Return0 \vee \neg Success) \Rightarrow NotPossible$$

It could be expanded by our system, but the expanded specification is not the correct one. Several lines from its predicate part are not correct.

Thus, it seems that there are bugs with our way to implement an implication operator. At the current, our method to expand this operator is to use its equivalent form. This form has been described on previous chapter.

In addition to both specifications, there are several specifications in our experiments which could not also be expanded by our system. However, this time it is not an error on our code; it is as expected. It originated from the incorrect *Z* specification: those schemas have unmatched variables.

Thus, our system generated outputs of these specifications as the same as its inputs in which schema calculus definitions still exist. All these specifications will be discussed as follows.

Impossible Specifications to be Expanded

This sub-section discusses our experiments which are impossible to be expanded. As a result, the same specification as the specification input was generated by our system in which the schema calculus definitions have not been expanded. Each of this experiment is given on a separate paragraph.

expandingsch1_20.tex This specification has the same state and initialization as both specifications above. One schema calculus specified is as follows:

$$Return \cong (Return0 \wedge Success) \vee (NotPossible \wedge Failure)$$

All of these schemas have been given earlier in this chapter.

However, there is a bit modification on the **NotPossible** schema. The modified one is as follows:

$\frac{NotPossible}{\exists BoxOffice; s? : Seat1; c? : Customer}$
$s? \mapsto c? \notin sold$

It is supposed to be the type for **s?** is an instance of **Seat**.

In other schema, **Return0**, specified in the above schema calculus definition, this variable was still specified as an instance of **Seat**. Thus, there is one common variable which has different types. These incompatible types were the source of the problem so that our system refused to generate an expanded specification of this specification input.

expandingsch6_3.tex This specification was taken from [2]. Its state and initialization schemas have been given earlier in this chapter.

A schema calculus definition was specified as follows:

$$AddI \cong Add \setminus (arg22)$$

This definition wants to hide the `arg22` variable from the `Add` schema and specify this schema as the new schema named `AddI`.

However, this variable is not available in the `Add` schema. Thus, the above definition cannot be expanded. Our system generated the same specification as the input `Z` file.

expandingsch6_4.tex This specification is quite similar to the specification above. A difference is on a schema calculus definition in which here there are two variables which are required to hide. One of these variables is the same as the hidden variable of the above specification. Another one is a variable, `arg2`, which is indeed declared in the operated schema. However, the unavailable variable made this schema calculus definition not be expanded. As a result, the same specification as the specification input was generated by our system.

expandingsch8_4.tex A specification used here is also similar to the previous specification. However, this time a different schema operator was used in the schema calculus definition which is a universal quantifier, " \forall ". The definition is as follows:

$$Subtract \cong \forall displaying : \mathbb{Z} \mid arg2 < 0 \bullet Add$$

As can be seen above, the same operational schema was operated.

However, this schema calculus definition could not also be expanded. A variable which is required to hide, `displaying`, is also not declared in the `Add` schema. Our system generated the specification input as an output of this expansion process.

Based on discussions on both sub-sections and previous discussions, our system cannot expand all schema calculus definition specified in `Z` specifications on our experiments. Although only two of these specifications could not be expanded, these fails cause great concerns. One of these concerns is to revise our method to expand an implication operator. Another one is to implement a better automatic simplification of a predicate part.

9.3.2 Evaluation of the #2 Question

As can be seen in Table 9.2 on page 243, several Z specifications do not have verification times. It means that these specifications could not be translated by Z2SAL. Furthermore, such specifications contain negation, hiding or renaming operator in their schema calculus. As also be discussed above, Z2SAL generated errors of redeclaring state or global variables.

However, all our expanded specifications from the above experiments which could be translated by Z2SAL, these specifications could also be verified and simulated by the SAL tool. It is because our Z specification inputs are quite simple Z specifications which do not have user-defined functions.

If there is a user-defined function specified in our input, the same problem as on our experiments with the generic constant definition system can be experienced. It is because the current method of a translation for such a function adopted by Z2SAL is quite different from a method adopted by the SAL language.

However, errors of incompatible types for a function application were also experienced in other specifications of our experiments with this system. Although these errors do not originate from user defined functions, several functions declared in schemas seem sources of these errors. These specifications, which were not discussed on above experiments and were not put in Table 9.1 on page 222 or Table 9.2 on page 243, will be discussed later.

An out of memory error can also be a problem if our Z specification input is rather a complex Z specification. It is because a schema expansion creates a new operational schema. One operational schema represents a state in a transition system of either the Z language or the SAL language.

For example, let us look at the generated SAL of one of previous examples. Its name is `output_expandingsch3_4.sal`. This SAL file was generated from the `output_expandingsch3_4.tex` Z specification which has five operational schemas. Each of these operational schemas was translated as a guarded transition. All guarded transitions were specified asynchronously by Z2SAL. An asynchronous system means that this system is only deadlocked if all its components are [18].

If the only schema calculus definition was deleted from the Z specification, the generated SAL file could be executed by the SAL model checker on 0.015 second. The time difference between this SAL file and the SAL with an expanded schema is just 0.001 second; it is almost the same time taken. It is because this specification though it has been expanded is still quite a simple Z specification.

Tables 9.3 on page 249 and 9.4 on page 250 show us other experiments conducted in our system.

Table 9.3: Other Experiments with the Expansion System

Z Specification (.tex)	Details	Verification time in secs	
		Non-simplified	Simplified
expandingsch1_1	" \wedge "	0.0	
expandingsch1_2	" \wedge "	0.016	
expandingsch1_3	" \wedge, \wedge, \wedge "	0.0	
expandingsch1_4	" \wedge, \wedge "	0.015	
expandingsch1_5	" \vee, \wedge "	0.015	
expandingsch1_6	" \vee, \wedge "	0.0	
expandingsch1_7	" \wedge, \vee "	0.015	
expandingsch1_8	" \wedge, \vee "	0.0	
expandingsch1_9	" \wedge "	0.0	
expandingsch1_10	" \vee, \vee, \vee "	0.0	
expandingsch1_11	" \wedge, \vee, \wedge "	0.016	
expandingsch1_12	" \wedge, \vee, \wedge "	0.015	
expandingsch1_13	" \wedge, \vee, \wedge "	0.016	
expandingsch1_14	" \wedge, \vee, \wedge "	0.016	
expandingsch1_15	" \wedge "	0.016	
expandingsch1_16	" \wedge "	0.016	
expandingsch1_17	" \wedge "	0.016	
expandingsch1_18	" \wedge "	0.031	
expandingsch1_19	" \wedge, \vee, \wedge "	0.032	
expandingsch1_20	" \wedge, \vee, \wedge "	N/A	
expandingsch1_21	" \wedge, \vee, \wedge "	0.03	
expandingsch1_22	" \wedge, \vee "	0.031	
expandingsch1_23	" \wedge, \vee "	0.032	
expandingsch1_24	" \wedge, \vee "	0.031	
expandingsch1_25	" \wedge, \vee "	0.016	
expandingsch1_26	" \wedge "	0.015	
expandingsch1_27	" \wedge "	0.031	
expandingsch1_28	" \vee, \vee, \vee "	0.031	
expandingsch1_29	" \vee, \vee, \vee "	0.031	
expandingsch1_30	" \vee, \vee, \vee "	0.047	
expandingsch1_31	" \vee, \wedge, \vee "	0.0	
expandingsch1_32	" \vee, \vee, \wedge "	0.015	
expandingsch2_1	" \neg "	N/A	
expandingsch2_2	" \neg, \wedge "	0.032	
expandingsch2_3	" \neg "	N/A	
expandingsch2_5	" \neg, \wedge "	N/A	
expandingsch2_6	" \neg, \wedge "	N/A	

Table 9.4: Other Experiments with the Expansion System (continued)

Z Specification (.tex)	Details	Verification time in secs	
		Non-simplified	Simplified
expandingsch2_7	" \wedge, \neg "	0.0	
expandingsch2_8	" \wedge, \neg "	0.0	
expandingsch2_9	" \neg, \wedge, \neg "	N/A	
expandingsch3_3	" \wedge, \Rightarrow "	0.016	
expandingsch3_5	" \Rightarrow, \wedge "	0.015	
expandingsch3_6	" \Rightarrow, \wedge "	0.031	
expandingsch3_7	" $\Rightarrow, \vee, \Rightarrow$ "	N/A	
expandingsch3_8	" $\wedge, \Rightarrow, \wedge$ "	0.015	
expandingsch3_9	" $\wedge, \Rightarrow, \wedge, \Rightarrow, \wedge$ "	0.015	
expandingsch8_5	" \forall "	N/A	
expandingschema_9	" \wedge, \neg, \wedge " " $\wedge, [,]$ " " $\neg, \wedge, [,]$ " " \vee " " \wedge, \vee "	N/A	

Experiments, but have not been discussed above, which had problems in translating or verifying them will be discussed on separate sub-sections as follows.

Case In-Sensitive in SAL

The SAL model checker generated an error during a verification of a generated SAL file either of `expandingsch1_26.tex`, `expandingsch1_27.tex`, `expandingsch1_28.tex`, `expandingsch1_29.tex`, or `expandingsch1_30.tex`. It is because a value or a member of a enumeration type variable was declared. This value has the same name as one of keywords of SAL though this variable has different case of letters from the keyword. Another error in this SAL file is a variable which has the same name as that value though also in different case of letters.

All these specifications were taken from [31], but have been modified in several places to be able to be translated by Z2SAL. One of modifications is to have one state schema. A state and initialization schemas are given as follows:

<i>Flexi</i>
<i>Standard_Hours, Flexitime_Hours : Time → Period</i> <i>worked : Ident → Period; in : Ident → Time</i>
$\text{dom } in \subseteq \text{dom } worked$

<i>InitFlexi</i>
<i>Flexi</i>
$in = \emptyset$ $worked = \emptyset$

Referring to the above discussion, the involved variable is **in**. This variable is indeed one of SAL keywords. Let us look at definitions and abbreviations specified in these specifications which are given as follows:

Time == \mathbb{N}
Period == $\mathbb{P} \text{ Time}$
Response ::= *In* | *Out* | *Balance* | *IdUnKnown*
RelMinutes == \mathbb{Z}

Another source of error is **In**.

A schema calculus definition specified in `expandingsch1_26.tex` is as follows:

$$ClockIn \hat{=} ClockIn_0 \wedge Worked$$

Both operational schemas on this specification are as follows:

<i>ClockIn_0</i>
$\Delta Flexi; ident? : Ident; t? : Time; ind! : Response$
$ident? \in \text{dom } worked$ $Standard_Hours' = Standard_Hours$ $Flexitime_Hours' = Flexitime_Hours$ $ident? \notin \text{dom } in$ $t? \in Flexitime_Hours(t?)$ $in' = in \cup \{ident? \mapsto t?\}$ $worked' = worked$ $ind! = In$

<i>Worked</i>
$\Delta Flexi; ident? : Ident$ $t? : Time; ind! : Response; cr! : RelMinutes$
$ident? \in \text{dom } worked$ $Standard_Hours' = Standard_Hours$ $Flexitime_Hours' = Flexitime_Hours$ $cr! = \#(worked'(ident?) \cap Flexitime_Hours(t?)) - \#\{t : Standard_Hours(t?) \mid t < t?\}$

On the other hand, a schema calculus definition on `expandingsch1_28.tex` is as follows:

$$ClockOut \cong ClockOut_0 \wedge Worked$$

The `ClockOut_0` operational schema is as follows:

$\frac{ClockOut_0}{\Delta Flexi; ident? : Ident; t? : Time; ind! : Response}$
$\begin{aligned} & ident? \in \text{dom } worked \\ & Standard_Hours' = Standard_Hours \\ & Flexitime_Hours' = Flexitime_Hours \\ & ident? \in \text{dom } in \\ & worked' = worked \oplus \{ident? \mapsto (worked(ident?) \cup (in(ident?) \dots (t? - 1)))\} \\ & ind! = Out \end{aligned}$

A schema calculus definition on the `expandingsch1_28.tex` operational schema is given as follows:

$$InsertKey \cong ClockIn \vee ClockOut \vee ReadOut \vee UnKnown$$

The first two schemas mentioned on the definition are created schemas which were expanded from the two previous schema calculus definitions. Manually both these schemas were added to this specification by copying from outcomes of previous specifications. Other schemas are as follows:

$\frac{ReadOut}{\Delta Flexi; ident? : Ident; t? : Time}$
$\begin{aligned} & ind! : Response; cr! : RelMinutes \\ & ident? \in \text{dom } worked \\ & Standard_Hours' = Standard_Hours \\ & Flexitime_Hours' = Flexitime_Hours \\ & cr! = \#(worked'(ident?) \cap Flexitime_Hours(t?)) - \#\{t : Standard_Hours(t?) \mid t < t?\} \\ & ident? \notin \text{dom } in \\ & t? \notin Flexitime_Hours(t?) \\ & ind! = Balance \\ & worked' = worked \\ & in' = in \end{aligned}$

$\frac{UnKnown}{\exists Flexi; ident? : Ident; ind! : Response}$
$\begin{aligned} & ident? \notin \text{dom } worked \\ & ind! = IdUnKnown \end{aligned}$

The `expandingsch1_29.tex` specification has a similar schema calculus definition to previous definition. The difference is on the order of the operational schemas. In this specification, such a definition is as follows:

$$InsertKey \cong ReadOut \vee UnKnown \vee ClockIn \vee ClockOut$$

As well as the `expandingsch1_30.tex` specification, this specification has a similar schema calculus definition as two previous specifications. The order

of the operational schemas differs from two earlier specifications. It can be seen as follows:

$$\text{InsertKey} \hat{=} \text{ReadOut} \vee \text{ClockIn} \vee \text{UnKnown} \vee \text{ClockOut}$$

Based on these experiments, it seems that the SAL language accepts not case sensitive letters of variables. It means that words are still the same though these words have a variety of cases of its letters. On the other hand, the Z language is case sensitive.

A way to fix these problems is to change an involved value or variable into different names which are not members of the SAL keywords. The first specification above could be verified by the SAL model checker afterwards. However, other specifications still could not be verified. Another problem to such specifications will be discussed as follows.

A Range of Numbers

This problem was reported by Z2SAL during a translation of one of either `expandingsch1_27.tex`, `expandingsch1_28.tex`, `expandingsch1_29.tex`, or `expandingsch1_30.tex`. An error message produced by Z2SAL is as follows:

The conversion from Z to SAL failed because null cannot resolved to a set of constants.

This error related to the `ClockOut_0` operational schema. The suspicious line is as follows:

$$\text{worked}' = \text{worked} \oplus \{ \text{ident?} \mapsto (\text{worked}(\text{ident?}) \cup (\text{in}(\text{ident?}) \dots (t? - 1))) \}$$

Our reason for this involved schema and line is based on our experiments. If such a line was deleted, the above error did not appear. This specification could be translated by Z2SAL.

This line was modified then as follows:

$$\text{in}(\text{ident?}) < (t? - 1) \Rightarrow \text{worked}' = \text{worked} \oplus \{ \text{ident?} \mapsto (\text{worked}(\text{ident?}) \cup (\text{in}(\text{ident?}) \dots (t? - 1))) \}$$

Z2SAL still failed to translate it.

Other experiment is to modify it to a line as follows:

$$\text{worked}' = \text{worked} \oplus \{ \text{ident?} \mapsto (\text{worked}(\text{ident?}) \cup \{(\text{in}(\text{ident?}))\}) \}$$

Fortunately this specification could be translated by Z2SAL.

It seems that there is a problem of a range of numbers. This is also proven by modifying the above line as follows:

$$\text{worked}' = \text{worked} \oplus \{ \text{ident?} \mapsto (\text{worked}(\text{ident?}) \cup (1 \dots 2)) \}$$

Having this line of predicate, this specification could be translated by Z2SAL. However, its generated SAL file could not be verified by the SAL model checker. This problem will be discussed as follows.

A Mismatch in the Function Application

The generated SAL files of either `expandingsch1_27.tex`, `expandingsch1_28.tex`, `expandingsch1_29.tex`, or `expandingsch1_30.tex` have another error relating to a type mismatch in the function application. The actual error message from `output_expandingsch1_27.sal` is as follows:

```
Error: [Context: output_expandingsch1_27, line(93), column(19)]:
Type mismatch in the function application.
Expected type:
[set{output_expandingsch1_27!B_Time}!Set,
set{output_expandingsch1_27!B_Time}!Set]
Actual type:
[output_expandingsch1_27!Set_B_Time,
set{output_expandingsch1_27!Range1_2}!Set]
```

Involved lines of the SAL file are as follows:

```
89 worked' =
90 function {Ident, set {B_Time;} ! Set; set {B_Time;} !
91 singleton(5)} !
92 insert(worked, (ident?, set {B_Time;} !
93 union(worked(ident?), set {Range1_2;} ! full))) AND
```

This error says that there was a type mismatch in the `union` function. The actual and expected types of this function are not compatible, as can be seen above. The second parameter of this function is formed from a range of numbers.

Since this error has been solved yet, a simpler predicate for the involved line was used in which there is no a range of numbers. The associated line is as given earlier. The SAL model checker could verify this SAL file.

Redeclaring State or Global Variables

This error occurs during a translation by Z2SAL. Several Z specifications fell into this category of error. Usually these specifications have 'N/A's for their verification times. These specifications are `expandingsch2_1.tex`, `expandingsch2_3.tex`, `expandingsch2_5.tex`, `expandingsch2_6.tex`, `expandingsch2_9.tex`, `expandingsch3_7.tex`, `expandingsch8_5.tex`, and `expandingschema_9.tex`.

Schema operators specified in their schema calculus definitions which can originate such a problem are negation, implication and hiding by using a universal quantifier. This problem is hard to fix. A potential solution is to let a Z specification have many state schemas. This solution relates to

a Z2SAL upgrade. However, this solution is only applicable to a negation operator. It is inapplicable to hiding or renaming operator.

This error prevents Z2SAL to generate a SAL file from a Z specification input. Thus, there is no SAL file which is required to be verified by the SAL model checker.

Therefore, there is a situation in which Z specifications generated by our system could not be translated by Z2SAL. Redeclaring state variables was found as the situation. Furthermore, it was also experienced several SAL files generated by Z2SAL could not be verified by the SAL model checker either. In this case, the associated SAL files contain a translation of a range of numbers and a type mismatch in the function application.

9.3.3 Evaluation of the #3 Question

Tables 9.5 on page 256 and 9.6 on page 257 show us sizes of our Z specifications on these experiments. As can be seen from these three tables, a range of sizes of our Z specification inputs is between 1 and 3 kilobytes, otherwise the ranges are 1 to 8 and 1 to 14 for Z specification outputs and SAL specifications respectively. Sizes of SAL specifications shown in this table are original sizes producing by Z2SAL. As discussed above, a few of these SAL specifications have been modified as required in order to be executed by the SAL tool successfully or have been simplified to their compact form of predicates. Thus, their sizes can be different from the original ones.

'input' means a Z specification input file for our system. On the other hand, 'output' means a Z specification output file generated by our system after performing an expansion process, 'N/A's in output means an associated Z specification input could not be expanded by our system either because of errors on the input file or because of bugs on our system, 'N/A's in SAL specifications means an associated Z specification could not be translated by Z2SAL. It can also be seen that a 'N/A' in input makes this Z specification is not possible to be further processed.

One issue that is important to consider is by having many schema calculus definitions, both a Z specification and a SAL specification are also getting big in sizes. Another important issue is that a size of a SAL specification is roughly twice to four times of its Z specification.

As a conclusion, our approach to expand schema calculus definitions scales to larger specifications. However, as the outcomes of our system will be translated by Z2SAL and executed by the SAL tool later, the large specification resulted by our system is possible to be a problem with both tools.

Table 9.5: Sizes of Z Specifications

Z Specifications (.tex)	Sizes in KB		
	input	output	SAL specifications
expandingschema_1	2	2	7
expandingschema_2	2	2	6
expandingschema_3	2	2	6
expandingschema_4	2	3	10
expandingschema_5	2	3	9
expandingschema_6	2	2	4
expandingschema_7	2	3	N/A
expandingschema_8	3	5	N/A
expandingsch2_4	1	N/A	N/A
expandingsch3_1	1	1	3
expandingsch3_2	2	2	4
expandingsch3_4	2	2	4
expandingsch4_1	2	2	4
expandingsch4_2	2	2	5
expandingsch5_1	1	1	N/A
expandingsch5_2	1	1	N/A
expandingsch6_1	1	1	N/A
expandingsch6_2	2	2	N/A
expandingsch7_1	1	1	2
expandingsch8_1	2	2	N/A
expandingsch8_2	2	2	N/A
expandingsch8_3	2	2	N/A
expandingsch8_6	1	2	N/A
expandingsch1_1	1	1	3
expandingsch1_2	1	1	3
expandingsch1_3	1	1	3
expandingsch1_4	1	1	3
expandingsch1_5	1	1	3
expandingsch1_6	1	1	3
expandingsch1_7	1	1	3
expandingsch1_8	1	1	3
expandingsch1_9	1	1	3
expandingsch1_10	1	1	3
expandingsch1_11	1	2	3
expandingsch1_12	1	2	4
expandingsch1_13	1	2	4
expandingsch1_14	1	2	3
expandingsch1_15	1	1	3
expandingsch1_16	1	1	3
expandingsch1_17	1	1	3
expandingsch1_18	1	1	3
expandingsch1_19	2	2	4
expandingsch1_20	2	N/A	N/A
expandingsch1_21	2	2	4

Table 9.6: Sizes of Z Specifications (continued)

Z Specification (.tex)	Sizes in KB		
	input	output	SAL specifications
expandingsch1_22	2	2	4
expandingsch1_23	2	2	4
expandingsch1_24	2	2	4
expandingsch1_25	2	2	5
expandingsch1_26	2	3	7
expandingsch1_27	2	3	7
expandingsch1_28	3	5	14
expandingsch1_29	3	5	14
expandingsch1_30	3	5	14
expandingsch1_31	1	1	3
expandingsch1_32	1	1	3
expandingsch2_1	1	1	N/A
expandingsch2_2	1	1	3
expandingsch2_3	1	2	N/A
expandingsch2_5	1	2	N/A
expandingsch2_6	1	2	N/A
expandingsch2_7	1	1	3
expandingsch2_8	1	1	3
expandingsch2_9	1	2	N/A
expandingsch3_3	2	2	4
expandingsch3_5	2	2	4
expandingsch3_6	2	2	4
expandingsch3_7	2	2	N/A
expandingsch3_8	2	2	4
expandingsch3_9	2	2	5
expandingsch3_10	2	N/A	N/A
expandingsch6_3	2	N/A	N/A
expandingsch6_4	2	N/A	N/A
expandingsch7_2	2	N/A	N/A
expandingsch8_4	2	N/A	N/A
expandingsch8_5	1	2	N/A
expandingschema_9	3	8	N/A

9.4 Conclusion

Based on our experiments, our system could expand schema calculus definitions on Z specification inputs to some extent. This system could also generate expanded schemas which sometimes could be translated by Z2SAL. Fortunately, almost all generated SAL files could be executed by the SAL tool. It was found that the SAL language is not a case sensitive language. Another finding is that it seems there is a bug on a translation of a range of

numbers on Z2SAL. This finding convinces us to such a bug since our other experiments with Z2SAL also found this.

Being able to expand a schema calculus definition, Z2SAL experiences a big specification. However, it cannot a huge specification. The huge specification requires more time to be translated by Z2SAL and to be executed by the SAL tool. Furthermore, it risks experiencing an out of memory error generated by the SAL tool. Fortunately, there is no of our experiments fell into this error. It is because our Z specifications are simple systems.

Redeclaring state or global variables is the error which usually occurs in translating an expanded specification. Solution to this problem can be set as a future work. One solution is to have many state schemas. Thus, one state schema is specified to have just a variable part. Another state schema has a predicate part. Having these state schemas, a user can collapse state variables easily without a bother on negated predicates of other state schema.

Chapter 10

Conclusion and Future Work

Our thesis aims to enhance the ability of Z2SAL to translate generic constants and schema calculus specified in Z specification inputs. It has been highlighted in the above discussion the difficulties associated with translating generic constants and schema calculus which are specified in Z specifications. Previous chapters have also discussed the objectives and contribution of this thesis as well as the implementation of our system, experiments and evaluations with the system.

This is the final chapter of this thesis which will conclude previous chapters and offer several works for enhancing our work. The first discussion is a brief summary of this thesis. Let us move to the discussion.

10.1 Thesis Summary

As mentioned above that our objectives as well as our contribution have been discussed on previous chapter, specifically in Section 1.2 on page 15. There were two objectives stated in that section.

The first objective is to implement a tool which will redefine a generic constant definition to an equivalent axiomatic definition based on usages of this generic constant. The second one is to implement a tool to construct a new schema by expanding other schemas, in which they are connected by schema operators.

These tools were implemented in a system which is called support for model checking Z specifications. Having this system, the applicability of model checking Z specifications can be broadening. Indeed, it is our expected contribution.

A description and discussions on our system were structured into 9 chapters excluding this chapter. These chapters are summarized briefly as follows.

Chapter 1 on page 12 also outlined the thesis structure and publications made of this thesis in addition to the contents mentioned above. Chapter 2 on page 20 described main topics relate to our thesis. There were five sections discussed on this chapter. They are formal method, the Z notation, model checking, the SAL tool and Z2SAL. Chapter 3 on page 74 provided us with our proposal on translation of embedded theorems in Z specifications. Chapter 4 on page 88 detailed the implementation of our Z scanner and parser. A small portion of relevant code was also given to accompany the discussion. On the other hand, Chapter 5 on page 111 described the implementation of our system of a redefinition of generic constants and the Chapter 6 on page 130 described the implementation of our system of an expansion of schema calculus. Several groups of code were given as well as captured screen-shots of our system when running such code. Chapter 7 on page 171 discussed how to integrate those four separate systems into a support for model checking Z specifications system. Chapter 8 on page 179 and Chapter 9 on page 221 provided several experiments on redefinition and expansion systems respectively as well as evaluations of both systems.

The following section discusses a main contribution offered by our research.

10.2 The Main Contribution of Our Research

As mentioned above, our main contribution is expected to broaden the applicability of model checking Z specifications and to enrich the literature of support for model checking Z specifications. Our contribution consists of small pieces of contributions as follows:

- Proposing a method of translating theorems embedded in Z specifications. This proposal has been implemented in one version of Z2SAL by its researchers. This proposal was available in Chapter 3 on page 74.
- Implement a system of a redefinition of generic constants. This system will redefine generic constant definitions to axiomatic definitions. The discussion on this work can be seen in Chapter 8 on page 179. This implementation included also implementation of Z scanner and Z parser.
- Propose a new translation of SAL function and constant. This proposal has been implemented manually by us in our experiments. However, it has not been adopted by Z2SAL as this proposal sometimes still cannot solve problems relating to SAL function. This proposal was discussed on Section 5.5 on page 126, but in other sections it was also explained.

- Implement a system of an expansion of schema calculus. This system will expand schema calculus definition to new schemas. The discussion on this work can be seen in Chapter 9 on page 221. This implementation included also implementation of Z scanner and Z parser.

10.3 Relating Research Outcomes to Research Objectives

As mentioned earlier, our research objectives are to implement a system which is able to redefine generic constants and to expand schema calculus specified in Z specifications. In order to achieve these aims, such a system has been implemented. Please see Chapters 4 on page 88, 5 on page 111, 6 on page 130, and 7 on page 171 for details of this system. This system does not translate both aspects of Z language to the SAL language as Z2SAL does, instead it will pre-process the Z specification by redefining the generic constant or expanding schema calculus such that the pre-processed Z specification is similar to the general Z specification which can be translated easily by Z2SAL.

Furthermore, an evaluation which was based on several defined questions was then conducted on this system to assess its performance. This evaluation obtained valuable information from our experiments on this system in order to test these questions. These evaluation were discussed on Sections 8.1 on page 179, 8.3 on page 210, 9.1 on page 221, and 9.3 on page 243. On the other hand, our experiments were discussed on Sections 8.2 on page 180, and 9.2 on page 222.

Based on this evaluation, our system, to some extent, is able to fulfil our aims. Several limitations of a predicate simplification, implementations of implication and bi-implication operators, and a complex schema calculus definition restricted our expansion system to expand all our Z specifications correctly. In addition, usages of simple types of generic constants and simple usage of such generic constants made our redefinition system succeed to redefine all of our Z specifications. However, more complex types of generic constants can possibly cause our system to fail a redefinition of such generic constants.

Since the outcomes of our system will further processed by Z2SAL and the SAL tool, a successful redefinition or expansion is also defined by a success on a translation by Z2SAL and an execution by the SAL tool. Relating to these requirements and previous discussions, Z2SAL and the SAL tool could not always run our inputs. Redefining state variables, a range of numbers,

an incompatible type of function application, and an out of memory error are problems which were sources of unsuccessful either translations by Z2SAL or executions by the SAL tool.

Despite above problems, overall, our system can meet our objectives. Furthermore, pre-processed Z specifications yield Z specifications which are possible to be translated by Z2SAL as opposed to original Z specifications. Based on our literature review, these original Z specifications cannot be translated by Z2SAL. The current Z2SAL generated an error message, implying that is on generic constant or schema calculus.

Identified or predicted limitations as mentioned above can be set up as future works which will be discussed follows.

10.4 Future Work

Discussions on previous chapters have also recommended several works for future. As explained above, these future works can be limitations which either were identified or were predicted during our experiments. These future works can also be devoted for our system or the Z2SAL translator. The latter one is only our recommendations which can be considered by Z2SAL researchers.

At first, works for enhancing our system will be discussed. Afterwards, recommendations for Z2SAL are described.

Experiments with more complex types of generic constants and complex predicates of generic constant definitions are our first recommendation for the redefinition system. The complexity relating to a type of generic constant is to use several allowed Z tags to specify the type of the generic constant inputs and output. More complex predicates which use these generic constants are another future work for this system. This complexity can be interpreted in either the large number of usages or the way the generic constant is used. These were discussed extensively on Section 8.3.1 on page 211.

A better simplification of predicates is our first recommendation for the expansion system. This simplification requires extensive works to automate manual simplifications which ease to perform. Better implementations of the implication and bi-implication operators are also important to be performed. Section 9.3.1 on page 243 discussed these works.

From Z2SAL side, it is recommended to allow a Z specification has many state schemas. It is also important to upgrade a translation of the tag of a range of numbers, "..". Support for function in a range of another function will also benefit Z2SAL. These can be seen in Sections 8.3.2 on page 214, and 9.3.2 on page 248. Another worth recommendation is a revision to a

SAL translation of a user-defined function or constant (see Sections 8.3.1 on page 211, 8.3.2 on page 214, 8.3.3 on page 217, and 9.3.2 on page 248).

On the other hand, out of memory errors can be solved by resizing variables in SAL specifications to smaller appropriate sizes. If these problems persist, experiment these SAL specifications with high performance computing machines. Another solution is to apply abstraction on Z specifications. These problems are discussed on Sections 8.3.3 on page 217, and 9.3.3 on page 255.

10.5 Finally

Finally, all these discussions aim to introduce our research summarized in this thesis. Furthermore, the implementation of the system is a practical application to achieve our objectives. It is then supported by experimenting with this system which is empirical studies to obtain expected results. In the end, an evaluation was performed in order to assess the reliability and performance of our system. Due to time limitations, this research has to conclude in this thesis. Weaknesses of our system and other system have been offered as future works.

Bibliography

- [1] R. Arthan and R. B. Jones. The Story of ProofPower. Technical report, February 2005. Available at <http://www.rbjones.com/rbjpub/pp/doc,2005>.
- [2] R. Barden, S. Stepney, and D. Cooper. *Z in Practice*. Prentice-Hall, Inc., 1995.
- [3] M. Ben-Ari. *Mathematical Logic for Computer Science*. Springer, 2012.
- [4] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, Jun 2000. NASA Langley Research Center.
- [5] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Computer Aided Verification*, pages 319–331. Springer, 1998.
- [6] S. Bensalem, Y. Lakhnech, and S. Owre. Invest: A Tool for the Verification of Invariants. In *Computer Aided Verification*, pages 505–510. Springer, 1998.
- [7] D. Bjørner. *Software Engineering 1*, volume 1. Springer-Verlag, Berlin Heidelberg, 2006.
- [8] C. Bolton. Using the Alloy Analyzer to Verify Data Refinement in Z. *Electronic Notes in Theoretical Computer Science*, 137(2):23–44, 2005.
- [9] A. D. Brucker, F. Rittinger, and B. Wolff. Hol-Z 2.0. *Journal of Universal Computer Science*, 9(2):152–172, 2003.
- [10] R. Bussöw. *Model Checking Combined Z and Statechart Specifications*. PhD thesis, PhD thesis, Fakultät IV - Elektrotechnik und Informatik, Technische Universität Berlin, 2003.

- [11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169. Springer, 2000.
- [12] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.
- [13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT press, 1999.
- [14] M. Cristiá, P. Albertengo, and P. R. Monetti. Fastest: A Model-based Testing Tool for the Z Notation. *PTD-SEFM, Consiglio Nazionale della Ricerche, Pisa, Italy*, pages 3–8, 2010.
- [15] D. Dams, R. Gerth, and O. Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):253–291, 1997.
- [16] C. Daws and S. Tripakis. Model Checking of Real-time Reachability Properties using Abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 313–329. Springer, 1998.
- [17] L. de Moura. SAL: Tutorial. *Computer Science Laboratory, SRI International*, 2004.
- [18] L. de Moura, S. Owre, and N. Shankar. The SAL Language Manual. *Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep. CSL-01-01*, 2003.
- [19] H. Deitel and P. J. Deitel. *Java : How to Program (5th (International) ed.)*. Upper Saddle River, NJ: Prentice-Hall, 2003.
- [20] J. Derrick, S. North, and A. J. Simons. Z2SAL-building a Model Checker for Z. In *Abstract State Machines, B and Z*, pages 280–293. Springer, 2008.
- [21] J. Derrick, S. North, and A. J. Simons. Z2SAL: A Translation-based Model Checker for Z. *Formal Aspects of Computing*, 23(1):43–71, 2011.
- [22] J. Derrick, S. North, and T. Simons. Issues in Implementing a Model Checker for Z. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 678–696. Springer Berlin Heidelberg, 2006.

- [23] J. Dingel and T. Filkorn. Model Checking for Infinite State Systems using Data Abstraction, Assumption-commitment Style Reasoning and Theorem Proving. In *Computer Aided Verification*, pages 54–69. Springer, 1995.
- [24] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z Specification Language: Version 1, 1991.
- [25] R. Duke and G. Smith. Temporal Logic and Z Specifications. *Australian Computer Journal*, 21(2):62–66, 1989.
- [26] L. Freitas. *Model Checking Circus*. PhD thesis, PhD thesis, Department of Computer Science, University of York, 2005.
- [27] S. Graf. Characterization of a Sequentially Consistent Memory and Verification of a Cache Memory by Abstraction. *Distributed Computing*, 12(2-3):75–90, 1999.
- [28] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification*, pages 72–83. Springer, 1997.
- [29] A. Hall. Integrating Z into Large Projects Tools and Techniques. In *ABZ2008 Conference*. Citeseer, 2008.
- [30] K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681. Springer, 1996.
- [31] I. Hayes and B. Flinn. *Specification Case Studies*. Prentice-Hall International London, 1987.
- [32] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.
- [33] T. Hurka. BYACC/J, 2008.
- [34] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.
- [35] D. Jackson. Abstract Model Checking of Infinite Specifications. In *FME'94: Industrial Benefit of Formal Methods*, pages 519–531. Springer, 1994.
- [36] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

- [37] D. Jackson. *Software Abstractions*. The MIT Press, 2006.
- [38] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1996.
- [39] S. C. Johnson. Yacc: Yet Another Compiler-Compiler, 2015. [Online; accessed 10-May-2017].
- [40] D. Jordan. CADiZ-computer Aided Design in Z. In *VDM'91 Formal Software Development Methods*, pages 685–686. Springer, 1991.
- [41] B. W. Kernighan, M. E. Lesk, and J. F. Ossanna. Document Preparation. *Bell Sys. Tech. J*, 57(6):2115–2135, 1978.
- [42] P. King. Printing Z and Object-Z LaTeX Documents. *Department of Computer Science, University of Queensland, May*, 393:404–410, 1990.
- [43] G. Klein. JFlex - The Fast Scanner Generator for Java, 2015.
- [44] L. Lamport. *LaTeX: A Document*, volume 14. pub-AW, 1994.
- [45] J. Levine, T. Mason, and D. Brown. *Lex & YACC*. O'Reilly & Associates, Inc., 1992.
- [46] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, and D. Probst. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
- [47] D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, PhD thesis, School of Computer Science, Carnegie Mellon University, 1993.
- [48] P. Malik, L. Groves, and C. Lenihan. Translating Z to Alloy. In *Abstract State Machines, Alloy, B and Z*, pages 377–390. Springer, 2010.
- [49] T. Marris. *Z Notes*, 2007.
- [50] A. P. Martin. Proposal: Community Z Tools Project (CZT), Sept. 2001.
- [51] S. Merz. Model Checking: A Tutorial Overview. In *Modeling and Verification of Parallel Processes*, pages 3–38. Springer, 2001.
- [52] A. Mota and A. Sampaio. Model-checking CSP-Z: Strategy, Tool Support and Industrial Application. *Science of Computer Programming*, 40(1):59 – 96, 2001.

- [53] O. Müller and T. Nipkow. Combining Model Checking and Deduction for I/O-automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–16. Springer, 1995.
- [54] R. Pelánek. *Reduction and Abstraction Techniques for Model Checking*. PhD thesis, PhD thesis, Faculty of Informatics, Masaryk University, Brno, 2006.
- [55] D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In *Integrated Formal Methods*, pages 480–500. Springer, 2007.
- [56] B. Potter, D. Till, and J. Sinclair. *An Introduction to Formal Specification and Z*. Prentice Hall PTR, 1996.
- [57] D. Rann, J. Turner, and J. Whitworth. *Z: A Beginner’s Guide*, volume 2. CRC Press, 1994.
- [58] A. J. D. Reis. *Compiler Construction Using Java, JavaCC, and Yacc*. Wiley-IEEE Press, 2012.
- [59] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International, 1997.
- [60] M. Saaltink. The Z/Eves System. In *ZUM’97: The Z Formal Specification Notation*, pages 72–85. Springer, 1997.
- [61] M. Saaltink et al. The Z/EVES 2.0 Users Guide. *ORA Canada*, pages 31–32, 1999.
- [62] A. Simons. The Z2SAL User Guide, 2012.
- [63] M. Siregar. Support for Model Checking Z Specifications. In *2016 IEEE 17th International Conference on Information Reuse and Integration (IRI)*, pages 241–248, July 2016.
- [64] M. U. Siregar and J. Derrick. An Investigation into the Use of Abstraction in Model Checking Z Specification. In *Proceedings of the 9th Annual South-East European Doctoral Student Conference*, pages 330–345. SEERC, Sep 2014.
- [65] M. U. Siregar and J. Derrick. Using Abstraction in Model Checking Z Specifications. In *The University of Sheffield Engineering Symposium Conference Proceeding*. The University of Sheffield, Jun 2014.

- [66] M. U. Siregar, J. Derrick, S. North, and A. J. H. Simons. Experiences using Z2SAL. In *2014 International Conference on Advanced Computer Science and Information System*, pages 225–231, Oct 2014.
- [67] G. Smith and L. Wildman. Model Checking Z Specifications using SAL. In *ZB 2005: Formal Specification and Development in Z and B*, pages 85–103. Springer, 2005.
- [68] G. Smith and K. Winter. Proving Temporal Properties of Z Specifications using Abstraction. In *ZB 2003: Formal Specification and Development in Z and B*, pages 260–279. Springer, 2003.
- [69] J. M. Spivey. *The Z Notation*, volume 1992. Prentice Hall New York, 1989.
- [70] J. M. Spivey. The Fuzz Manual. *Computing Science Consultancy*, 34, 1992.
- [71] M. Utting. *Data Structures for Z Testing Tools*. Department of Computer Science, University of Waikato, 2001.
- [72] A. Vakili and N. A. Day. *Temporal Logic Model Checking in Alloy*, pages 150–163. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [73] M. M. West. *Issues in Validation and Executability of Formal Specifications in the Z Notation*. PhD thesis, University of Leeds, 2002.
- [74] Wikibooks. LaTeX — Wikibooks, The Free Textbook Project, 2016. [Online; accessed 20-December-2016].
- [75] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., 1996.

Appendix A

The SAL File of Unique Allocator Specification

A.1 The SAL File of The Original Specification

```
uniqueAllocator.mod : CONTEXT = BEGIN
NZNAT : TYPE = [1..2];
NAT : TYPE = [0..2];
Counter_NZNAT : CONTEXT = count2 {NZNAT; 1, 2};
State : MODULE =
  BEGIN
    LOCAL used : set {NZNAT;} ! Set
    LOCAL alloc : set {NZNAT;} ! Set
    LOCAL sentNum : NAT
    LOCAL invariant_ : BOOLEAN
    DEFINITION
      invariant_ = (
        Counter_NZNAT ! size?(alloc) <= 1)AND
    INITIALIZATION [
      used = set {NZNAT;} ! empty AND
      alloc = set {NZNAT;} ! empty AND
      sentNum = 0 AND
      invariant_
    ->
  ]
  TRANSITION [
    Request :
      ((alloc = set {NZNAT;} ! empty AND
        used /= set {NZNAT;} ! full) => (EXISTS (q_ : NZNAT) : NOT
        set {NZNAT;} ! contains?(used, q_) AND
        alloc' = set {NZNAT;} ! singleton(q_) AND
        used' = set {NZNAT;} ! insert(used, q_)))AND
      ((alloc /= set {NZNAT;} ! empty OR
        used = set {NZNAT;} ! full) =>
        (alloc' = alloc AND used' = used)) AND
      sentNum' = 0 AND
      invariant_'
    ->
```



```

        used' IN {x : set {NZNAT;} ! Set|TRUE};
        alloc' IN {x : set {NZNAT;} ! Set|TRUE}
        sentNum_' IN {x : NAT | TRUE}
    []
    Send :
        (alloc /= set {NZNAT;} ! empty =>
         (set {NZNAT;} ! contains?(alloc ,sentNum_') AND
          alloc' = set {NZNAT;} ! empty AND used' = used)) AND
        (alloc = set {NZNAT;} ! empty =>
         (sentNum_' = 0 AND alloc' = alloc AND used' = used)) AND
        invariant..'
    -->
        used' IN {x : set {NZNAT;} ! Set|TRUE};
        alloc' IN {x : set {NZNAT;} ! Set|TRUE};
        sentNum_' IN {x : NAT | TRUE}
    ]
END;
th1: theorem State |- G((sentNum /= 2) OR X(G(sentNum /= 2)));
END

```

A.2 The SAL File of The Fifth Abstract Model

```

uniqueAllocatorAbs_simpl5 : CONTEXT = BEGIN
SState : TYPE = DATATYPE
    s3 ,
    s7 ,
    s9 ,
    s10 ,
END;
State : MODULE =
    BEGIN
        LOCAL s : SState
        INITIALIZATION [
            s = s9
            -->
        ]
        TRANSITION [
            ARequest :
                s = s3 => s' = s3 AND
                s = s7 => s' = s10 AND
                s = s9 => s' = s3 AND
                s = s10 => s' = s10
            -->
                s' IN {x : SState | TRUE}
        []
            ASend :
                s = s3 => s' = s7 AND
                s = s7 => s' = s10 AND
                s = s9 => s' = s9 AND
                s = s10 => s' = s1
            -->
                s' IN {x : SState | TRUE}
        ]
    END;
th1: theorem State |- G((s /= s7) OR X(G(s /= s7)));
END

```

Appendix B

The Counter-Example of The Fourth Abstraction

```
$ sal-smc uniqueAllocatorAbs_simpl4
Counterexample for 'th1' located at [Context: uniqueAllocatorAbs_simpl4, line(35), column(2)]:
=====
Path
=====
Step 0:
--- System Variables (assignments) ---
s = s4
-----
Transition Information:
(module instance at [Context: uniqueAllocatorAbs_simpl4, line(35), column(15)]
(label ARequest
transition at [Context: uniqueAllocatorAbs_simpl4, line(18), column(11)]))
-----
Step 1:
--- System Variables (assignments) ---
s = s3
-----
Transition Information:
(module instance at [Context: uniqueAllocatorAbs_simpl4, line(35), column(15)]
(label ASend
transition at [Context: uniqueAllocatorAbs_simpl4, line(25), column(11)]))
-----
Step 2:
--- System Variables (assignments) ---
s = s7
```

```

-----
Transition Information:
(module instance at [Context: uniqueAllocatorAbs_simpl4, line(35), column(15)]
(label ARequest
transition at [Context: uniqueAllocatorAbs_simpl4, line(18), column(11)]))
-----

Step 3:
--- System Variables (assignments) ---
s = s4
-----

Transition Information:
(module instance at [Context: uniqueAllocatorAbs_simpl4, line(35), column(15)]
(label ARequest
transition at [Context: uniqueAllocatorAbs_simpl4, line(18), column(11)]))
-----

Step 4:
--- System Variables (assignments) ---
s = s3
-----

Transition Information:
(module instance at [Context: uniqueAllocatorAbs_simpl4, line(35), column(15)]
(label ASend
transition at [Context: uniqueAllocatorAbs_simpl4, line(25), column(11)]))
-----

Step 5:
--- System Variables (assignments) ---
s = s7
-----

Transition Information:
(module instance at [Context: uniqueAllocatorAbs_simpl4, line(35), column(15)]
(label ASend
transition at [Context: uniqueAllocatorAbs_simpl4, line(25), column(11)]))
-----

Step 6:
--- System Variables (assignments) ---
s = s4
-----

Summary: The assertion 'th1' located at [Context:
uniqueAllocatorAbs_simpl4, line(35), column(2)] is invalid.

```

Appendix C

Full Z Specifications from Related Chapter

C.1 shop.tex

This specification is taken from [57].

[*ITEM*]

$RESPONSE ::= price_changed \mid item_ordered \mid item_sold \mid$
 $not_in_stock \mid delivery_done \mid no_price$

<i>Shop</i>
$stock : ITEM \rightarrow \mathbb{N}$ $cost : ITEM \rightarrow \mathbb{N}$ $order : ITEM \rightarrow \mathbb{N}$
$dom\ order = dom\ stock$ $dom\ cost = dom\ order \cup dom\ stock$

<i>InitShop</i>
<i>Shop'</i>
$stock' = \emptyset$ $cost' = \emptyset$ $order' = \emptyset$

<i>PriceChange</i>
$\Delta Shop; item? : ITEM$ $new_price? : \mathbb{N}; reply! : RESPONSE$
$cost' = cost \oplus \{item? \mapsto new_price?\}$ $reply! = price_changed$ $stock' = stock$ $order' = order$

SellItem

$\Delta Shop; item? : ITEM$
 $price! : \mathbb{N}; reply! : RESPONSE$

$item? \in \text{dom } cost$
 $item? \in \text{dom } stock$
 $stock(item?) \geq 1$
 $cost(item?) = price!$
 $stock' = stock \oplus \{item? \mapsto (stock(item?) - 1)\}$
 $reply! = item_sold$
 $order' = order$
 $cost' = cost$

SellItemNoItem

$\exists Shop; item? : ITEM; reply! : RESPONSE$

$(item? \notin \text{dom } stock \vee stock(item?) = 0)$
 $reply! = not_in_stock$

SellItemNoPrice

$\exists Shop; item? : ITEM; reply! : RESPONSE$

$item? \notin \text{dom } cost$
 $reply! = no_price$

OrderNewItem

$\Delta Shop; item? : ITEM$
 $order_level? : \mathbb{N}_1; cost? : \mathbb{N}; reply! : RESPONSE$

$item? \notin \text{dom } stock$
 $order' = order \cup \{item? \mapsto order_level?\}$
 $stock' = stock \cup \{item? \mapsto 0\}$
 $cost' = cost \cup \{item? \mapsto cost?\}$
 $reply! = item_ordered$

OrderItem

$\Delta Shop; item? : ITEM$
 $order_level? : \mathbb{N}_1; reply! : RESPONSE$

$item? \in \text{dom } stock$
 $order' = order \oplus \{item? \mapsto (order(item?) + order_level?)\}$
 $reply! = item_ordered$
 $stock' = stock$
 $cost' = cost$

<i>Delivery</i>
$\Delta Shop; delivery? : ITEM \rightarrow \mathbb{N}_1$ $reject! : ITEM \rightarrow \mathbb{N}_1; reply! : RESPONSE$
$\forall item : ITEM \bullet \exists no : \mathbb{N}_1 \bullet$ $((item \mapsto no) \in delivery? \wedge$ $(item \in \text{dom } order \wedge$ $(no = order(item) \vee no < order(item) \wedge$ $order' = order \oplus \{item \mapsto order(item) - no\} \wedge$ $stock' = stock \oplus \{item \mapsto stock(item) + no\}) \vee$ $(no > order(item) \wedge$ $order' = order \oplus \{item \mapsto 0\} \wedge$ $(item \mapsto (no - order(item))) \in reject! \wedge$ $stock' = stock \oplus \{item \mapsto stock(item) + order(item)\})) \vee$ $(item \notin \text{dom } order \wedge$ $(item \mapsto no) \in delivery? \wedge$ $(item \mapsto no) \in reject! \wedge$ $order' = order \wedge$ $stock' = stock) \vee$ $(item \notin \text{dom } delivery? \wedge$ $order' = order) \wedge$ $reply! = delivery_done \wedge cost' = cost$

C.2 telephonenetwork.tex

This specification is taken from [31, p. 31-34].

[PHONE]

$CON == \mathbb{P} PHONE$

$Status ::= Yes \mid No$

TN

$reqs, cons : \mathbb{P} CON$

$cons \subseteq reqs \wedge$
 $\forall c1, c2 : cons \bullet$
 $(c1 \neq c2) \Rightarrow (c1 \cap c2 = \emptyset)$

Init

TN'

$reqs' = \emptyset$
 $cons' = \emptyset$

efficientTN

TN

$\neg (\exists cons0 : \mathbb{P} CON \bullet$
 $(cons \subset cons0) \wedge$
 $(cons0 \subseteq reqs) \wedge$
 $\forall c1, c2 : cons0 \bullet$
 $(c1 \neq c2) \Rightarrow (c1 \cap c2 = \emptyset))$

DeltaTN

TN; TN'; ph? : PHONE

$\neg (\exists cons0 : \mathbb{P} CON \bullet (cons \subset cons0) \wedge (cons0 \subseteq reqs) \wedge$
 $\forall c1, c2 : cons0 \bullet (c1 \neq c2) \Rightarrow (c1 \cap c2 = \emptyset))$
 $\neg (\exists cons0 : \mathbb{P} CON \bullet (cons' \subset cons0) \wedge (cons0 \subseteq reqs') \wedge$
 $\forall c1, c2 : cons0 \bullet (c1 \neq c2) \Rightarrow (c1 \cap c2 = \emptyset))$
 $\neg (\exists cons1 : \mathbb{P} CON \bullet (cons \setminus cons1) \subset (cons' \setminus cons1) \wedge$
 $\neg (\exists cons0 : \mathbb{P} CON \bullet (cons1 \subset cons0) \wedge (cons0 \subseteq reqs) \wedge$
 $\forall c1, c2 : cons0 \bullet (c1 \neq c2) \Rightarrow (c1 \cap c2 = \emptyset)))$

<p><i>Call</i></p> <p>ΔTN <i>dialled?</i> : PHONE <i>ph?</i> : PHONE</p> <hr/> <p>$reqs' = reqs \cup \{\{ph?, dialled?\}\}$ $\neg (\exists cons1 : \mathbb{P} CON \bullet$ $(cons \setminus cons1) \subset (cons \setminus cons') \wedge$ $\neg (\exists cons0 : \mathbb{P} CON \bullet$ $cons1 \subset cons0 \wedge$ $cons0 \subseteq reqs \wedge$ $\forall c1, c2 : cons0 \bullet$ $(c1 \neq c2) \Rightarrow (c1 \cap c2 = \emptyset))$</p>

<p><i>HangUp</i></p> <p>ΔTN <i>ph?</i> : PHONE</p> <hr/> <p>$reqs' = reqs \setminus \{c : cons \mid ph? \in c\}$ $\neg (\exists cons1 : \mathbb{P} CON \bullet$ $(cons \setminus cons1) \subset (cons \setminus cons') \wedge$ $\neg (\exists cons0 : \mathbb{P} CON \bullet$ $cons1 \subset cons0 \wedge$ $cons0 \subseteq reqs \wedge$ $\forall c1, c2 : cons0 \bullet$ $(c1 \neq c2) \Rightarrow (c1 \cap c2 = \emptyset))$</p>

<p><i>Engaged</i></p> <p>ΔTN; <i>engaged!</i> : Status; <i>other!</i> : PHONE; <i>ph?</i> : PHONE</p> <hr/> <p>$reqs' = reqs$ $cons' = cons$ $(engaged! = Yes) \Rightarrow (\{ph?, other!\} \in cons)$ $(engaged! = No) \Rightarrow ph? \notin (\bigcup cons)$ $\neg (\exists cons1 : \mathbb{P} CON \bullet (cons \setminus cons1) \subset (cons \setminus cons') \wedge$ $\neg (\exists cons0 : \mathbb{P} CON \bullet cons1 \subset cons0 \wedge cons0 \subseteq reqs \wedge$ $\forall c1, c2 : cons0 \bullet (c1 \neq c2) \Rightarrow (c1 \cap c2 = \emptyset))$</p>
--

C.3 hotelspecguestcomps.tex

This specification is taken from [57, p. 55-57].

$HOTELROOM ::= Room1 \mid Room2 \mid Room3 \mid Room4 \mid Room5 \mid Room6 \mid Room7 \mid$
 $Room8 \mid Room9 \mid Room10 \mid Room11 \mid Room12 \mid Room13 \mid Room14 \mid Room15$

$RESPONSE ::= no_room_vacant \mid not_a_guest \mid success \mid wrong_number \mid add_to_tab_ok$

<p><i>Hotel</i></p> <p>$current_guest : \mathbb{P} GUEST$; $unoccupied_room : \mathbb{P} HOTELROOM$ $occupied_room : \mathbb{P} HOTELROOM$; $occupies : GUEST \leftrightarrow HOTELROOM$ $tab : HOTELROOM \leftrightarrow \mathbb{N}$</p> <hr/> <p>$current_guest = \text{dom } occupies$ $occupied_room = \text{ran } occupies$ $unoccupied_room = HOTELROOM \setminus occupied_room$</p>

<p><i>InitHotel</i></p> <p><i>Hotel'</i></p> <hr/> <p>$occupies' = \emptyset$ $tab' = \emptyset$ $occupied_room' = \emptyset$ $unoccupied_room' = HOTELROOM$</p>

<p><i>ArriveGuest</i></p> <p>$\Delta Hotel; \text{guest?} : GUEST$ $\text{room!} : HOTELROOM; \text{reply!} : RESPONSE$</p> <hr/> <p>$\text{room!} \in \text{unoccupied_room}$ $\text{occupies}' = \text{occupies} \cup \{\text{guest?} \mapsto \text{room!}\}$ $\text{tab}' = (\{\text{room!}\} \triangleleft \text{tab}) \cup \{\text{room!} \mapsto 0\}$ $\text{reply!} = \text{success}$</p>

<p><i>DepartGuest</i></p> <p>$\Delta Hotel; \text{guest?} : GUEST$ $\text{bill!} : \mathbb{N}; \text{reply!} : RESPONSE$</p> <hr/> <p>$\exists b : \mathbb{N} \bullet (\text{guest?} \in \text{current_guest} \wedge$ $(\text{guest?}, b) \in (\text{occupies} \circ \text{tab}) \wedge$ $b = \text{bill!} \wedge$ $\text{occupies}' = \{\text{guest?}\} \triangleleft \text{occupies} \wedge$ $\text{tab}' = \text{tab} \wedge$ $\text{reply!} = \text{success})$</p>

<p><i>ArriveError</i></p> <p>$\exists Hotel$ $\text{reply!} : RESPONSE$</p> <hr/> <p>$\text{unoccupied_room} = \emptyset$ $\text{reply!} = \text{no_room_vacant}$</p>
--

<p><i>DepartError</i></p> <p>$\exists Hotel$ $\text{guest?} : GUEST$ $\text{reply!} : RESPONSE$</p> <hr/> <p>$\text{guest?} \notin \text{current_guest}$ $\text{reply!} = \text{wrong_number}$</p>
--

<p><i>AddToTab</i></p> <p>$\Delta Hotel; \text{room?} : HOTELROOM$ $\text{charge?} : \mathbb{N}; \text{reply!} : RESPONSE$</p> <hr/> <p>$\exists n : \mathbb{N} \bullet (\text{room?}, n) \in \text{tab} \wedge$ $\text{room?} \in \text{occupied_room} \wedge$ $\text{tab}' = (\{\text{room?}\} \triangleleft \text{tab}) \cup \{\text{room?} \mapsto (\text{charge?} + n)\} \wedge$ $\text{occupies}' = \text{occupies} \wedge$ $\text{reply!} = \text{add_to_tab_ok}$</p>
--

<p><i>AddToTabError</i></p> <p>$\exists Hotel; \text{room?} : HOTELROOM$ $\text{charge?} : \mathbb{N}; \text{reply!} : RESPONSE$</p> <hr/> <p>$\text{room?} \notin \text{occupied_room}$ $\text{reply!} = \text{not_a_guest}$</p>
--

C.4 club_horz.txt

This specification is obtained from [57, p. 126-127].

[PERSON]

$MESSAGE ::= OK$

$Club \hat{=} [members : \mathbb{P} PERSON \mid \#members \leq 3]$

$Init \hat{=} [Club' \mid members' = \emptyset]$

$JoinOk \hat{=} [\Delta Club; name? : PERSON; reply! : MESSAGE \mid name? \notin members \wedge \#members < 3 \wedge members' = members \cup \{name?\} \wedge reply! = OK]$

$SumOfClub \hat{=} [\exists Club; sumC! : \mathbb{N} \mid sumC! = \#members]$

$LeaveClub \hat{=} [\Delta Club; name? : PERSON; sumC! : \mathbb{N} \mid name? \in members \wedge members' = members \setminus \{name?\} \wedge sumC! = \#members]$

C.5 counterMod4.tex

This specification was rewritten from its SAL version taken from [4].

$CounterMod4$ $count : \mathbb{N}$ <hr/> $count \leq 3$	$InitCounter$ $CounterMod4$ <hr/> $count = 0$
$Mod4$ $\Delta CounterMod4; count! : \mathbb{N}$ <hr/> $count' = (count + 1) \bmod 4$ $count! = count'$	

C.6 uniqueAllocator.tex

$Allocator$ $used : \mathbb{P} \mathbb{N}_1$ $alloc : \mathbb{F} \mathbb{N}_1$ <hr/> $\#alloc \leq 1$	$Init$ $Allocator$ <hr/> $used = \emptyset$ $alloc = \emptyset$
--	--

<i>Request</i> $\Delta \text{Allocator}$
$(\text{alloc} = \emptyset \wedge \text{used} \neq \mathbb{N}_1) \Rightarrow (\exists n : \mathbb{N}_1 \bullet n \notin \text{used} \wedge \text{alloc}' = \{n\} \wedge \text{used}' = \text{used} \cup \{n\})$ $(\text{alloc} \neq \emptyset \vee \text{used} = \mathbb{N}_1) \Rightarrow (\text{alloc}' = \text{alloc} \wedge \text{used}' = \text{used})$

<i>Send</i> $\Delta \text{Allocator}; n! : \mathbb{N}$
$\text{alloc} \neq \emptyset \Rightarrow n! \in \text{alloc} \wedge \text{alloc}' = \emptyset \wedge \text{used}' = \text{used}$ $\text{alloc} = \emptyset \Rightarrow n! = 0 \wedge \text{alloc}' = \text{alloc} \wedge \text{used}' = \text{used}$

C.7 carspark.tex

<i>CarsPark</i> $\text{count} : \mathbb{N}$ $\text{maximum} : \mathbb{N}$
$\text{count} \leq \text{maximum}$

<i>InitCarsPark</i> CarsPark
$\text{count} = 0$ $\text{maximum} = 3$

<i>Enters</i> $\Delta \text{CarsPark}$ $\text{space!} : \mathbb{N}$
$\text{count} < \text{maximum}$ $\text{count}' = \text{count} + 1$ $\text{space}' = \text{maximum} - \text{count}'$ $\text{maximum}' = \text{maximum}$

<i>Leaves</i> $\Delta \text{CarsPark}$ $\text{space!} : \mathbb{N}$
$\text{count} > 0$ $\text{count}' = \text{count} - 1$ $\text{space}' = \text{maximum} - \text{count}'$ $\text{maximum}' = \text{maximum}$

C.8 birthdaybook.tex

[NAME, DATE]

REPORT ::= ok | already-known | not-known

<i>BirthdayBook</i> $\text{known} : \mathbb{P} \text{NAME}$ $\text{birthday} : \text{NAME} \mapsto \text{DATE}$
$\text{known} = \text{dom birthday}$

<i>InitBirthdayBook</i> BirthdayBook
$\text{known} = \emptyset$

<i>AddBirthday</i>
$\Delta \text{BirthdayBook}; \text{name?} : \text{NAME}; \text{date?} : \text{DATE}$
$\text{name?} \notin \text{known}$ $\text{birthday}' = \text{birthday} \cup \{\text{name?} \mapsto \text{date?}\}$

<i>FindBirthday</i>
$\exists \text{BirthdayBook}; \text{name?} : \text{NAME}; \text{date!} : \text{DATE}$
$\text{name?} \in \text{known}$ $\text{date!} = \text{birthday}(\text{name?})$

<i>Remind</i>
$\exists \text{BirthdayBook}; \text{today?} : \text{DATE}; \text{cards!} : \mathbb{P} \text{NAME}$
$\text{cards!} = \{n : \text{known} \mid \text{birthday}(n) = \text{today?}\}$

C.9 paperexample.txt

[PERSON, TITLE]

$\text{black}, \text{brown}, \text{grey} : \text{PERSON}$ $\text{filmA}, \text{filmB}, \text{filmC}, \text{filmD} : \text{TITLE}$
--

<i>State</i>
$\text{members} : \mathbb{P} \text{PERSON}$ $\text{rented} : \text{PERSON} \leftrightarrow \text{TITLE}$ $\text{stockLevel} : \text{TITLE} \rightarrow \mathbb{N}$
$\text{dom } \text{rented} \subseteq \text{members}$ $\text{ran } \text{rented} \subseteq \text{dom } \text{stockLevel}$

<i>Init</i>
<i>State'</i>
$\text{members}' = \emptyset$ $\text{stockLevel}' = \emptyset$

<i>RentVideo</i>
ΔState $p? : \text{PERSON}$ $t? : \text{TITLE}$
$p? \in \text{members}$ $t? \in \text{dom } \text{stockLevel}$ $\text{stockLevel}(t?) > \#(\text{rented} \triangleright \{t?\})$ $(p?, t?) \notin \text{rented}$ $\text{rented}' = \text{rented} \cup \{(p?, t?)\}$ $\text{stockLevel}' = \text{stockLevel}$ $\text{members}' = \text{members}$

<i>AddTitle</i>
ΔState $t? : \text{TITLE}$ $\text{level?} : \mathbb{N}$
$\text{stockLevel}' = \text{stockLevel} \oplus \{(t?, \text{level?})\}$ $\text{rented}' = \text{rented}$ $\text{members}' = \text{members}$

<p><i>DeleteTitle</i></p> <hr/> <p>$\Delta State$ $t? : TITLE$</p> <hr/> <p>$t? \notin \text{ran } \textit{rented}$ $t? \in \text{dom } \textit{stockLevel}$ $\textit{stockLevel}' = \{t?\} \triangleleft \textit{stockLevel}$ $\textit{rented}' = \textit{rented}$ $\textit{members}' = \textit{members}$</p>
--

<p><i>AddMember</i></p> <hr/> <p>$\Delta State$ $p? : PERSON$</p> <hr/> <p>$p? \notin \textit{members}$ $\textit{stockLevel}' = \textit{stockLevel}$ $\textit{rented}' = \textit{rented}$ $\textit{members}' = \textit{members} \cup \{p?\}$</p>
--

<p><i>CopiesOut</i></p> <hr/> <p>$\exists State; t? : TITLE; \textit{copies}' : \mathbb{N}$</p> <hr/> <p>$t? \in \text{dom } \textit{stockLevel}$ $\textit{copies}' = \#(\textit{rented} \triangleright \{t?\})$</p>
--

Appendix D

The States Animation of telephonenetwork.tex

```
State 1
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!1986 : PHONE): false)) = false
reqs((LAMBDA (arg!1986 : PHONE): true)) = false
cons((LAMBDA (arg!1987 : PHONE): false)) = false
cons((LAMBDA (arg!1987 : PHONE): true)) = false
engaged_ = Yes
other_ = PHONE_1
invariant_ = true
-----

State 2
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!1988 : PHONE): false)) = true
reqs((LAMBDA (arg!1988 : PHONE): true)) = false
cons((LAMBDA (arg!1989 : PHONE): false)) = false
cons((LAMBDA (arg!1989 : PHONE): true)) = false
engaged_ = No
other_ = PHONE_1
invariant_ = true
-----
```

```

State 3
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!1990 : PHONE): false)) = true
reqs((LAMBDA (arg!1990 : PHONE): true)) = false
cons((LAMBDA (arg!1991 : PHONE): false)) = false
cons((LAMBDA (arg!1991 : PHONE): true)) = false
engaged_ = Yes
other_ = PHONE_1
invariant_ = true
-----

```

```

State 4
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!1992 : PHONE): false)) = false
reqs((LAMBDA (arg!1992 : PHONE): true)) = true
cons((LAMBDA (arg!1993 : PHONE): false)) = false
cons((LAMBDA (arg!1993 : PHONE): true)) = false
engaged_ = Yes
other_ = PHONE_1
invariant_ = true
-----

```

```

State 5
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!1994 : PHONE): false)) = false
reqs((LAMBDA (arg!1994 : PHONE): true)) = true
cons((LAMBDA (arg!1995 : PHONE): false)) = false
cons((LAMBDA (arg!1995 : PHONE): true)) = true
engaged_ = Yes
other_ = PHONE_1
invariant_ = true
-----

```

```

State 6
--- Input Variables (assignments) ---

```

```

ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!1996 : PHONE): false)) = true
reqs((LAMBDA (arg!1996 : PHONE): true)) = true
cons((LAMBDA (arg!1997 : PHONE): false)) = false
cons((LAMBDA (arg!1997 : PHONE): true)) = true
engaged_ = No
other_ = PHONE_1
invariant_ = true
-----

State 7
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!1998 : PHONE): false)) = true
reqs((LAMBDA (arg!1998 : PHONE): true)) = true
cons((LAMBDA (arg!1999 : PHONE): false)) = true
cons((LAMBDA (arg!1999 : PHONE): true)) = true
engaged_ = No
other_ = PHONE_1
invariant_ = true
-----

State 8
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!2000 : PHONE): false)) = true
reqs((LAMBDA (arg!2000 : PHONE): true)) = true
cons((LAMBDA (arg!2001 : PHONE): false)) = true
cons((LAMBDA (arg!2001 : PHONE): true)) = false
engaged_ = Yes
other_ = PHONE_1
invariant_ = true
-----

State 9
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1

```

```

--- System Variables (assignments) ---
reqs((LAMBDA (arg!2002 : PHONE): false)) = true
reqs((LAMBDA (arg!2002 : PHONE): true)) = true
cons((LAMBDA (arg!2003 : PHONE): false)) = false
cons((LAMBDA (arg!2003 : PHONE): true)) = false
engaged_ = No
other_ = PHONE_1
invariant_ = true
-----

State 10
--- Input Variables (assignments) ---
ph? = PHONE_1
dialled? = PHONE_1
--- System Variables (assignments) ---
reqs((LAMBDA (arg!2004 : PHONE): false)) = true
reqs((LAMBDA (arg!2004 : PHONE): true)) = true
cons((LAMBDA (arg!2005 : PHONE): false)) = false
cons((LAMBDA (arg!2005 : PHONE): true)) = false
engaged_ = Yes
other_ = PHONE_1
invariant_ = true
-----

Only 10 of 18.0 states were displayed.
Remark: the command (display-curr-states <max>)
can be used to display up to <max> states.

```


Appendix E

JFlex Specification: Lexer.flex

```
%%
%byaccj
%class ScannerCl
%{
    int line = 0;
    private Parser yyparser;

    public ScannerCl(java.io.Reader r, Parser yyparser){
        this(r);
        this.yyparser = yyparser;
    }

    public int lineCount(){
        line = line + 1;
        return (line);
    }
}%
%%

"\\documentclass[12pt,zed]{" article"}"      {
}

"\\documentstyle[12pt,zed]{" article"}"      {
}

"\\documentstyle[12pt,oz]{" article"}"      {
}

"\\usepackage{z-eves}"      {
}

"\\begin{" document"}"      {
    yyparser.yylval = new ParserVal(yytext());
}

"\\end{" document"}"      {
    yyparser.yylval = new ParserVal(yytext());
    return 0;
}

"\\begin{" zed"}"          {
    yyparser.yylval = new ParserVal(yytext());
```

```

    return Parser.BZED;
}

"\end{"zed"}"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EZED;
}

"\begin{"axdef"}"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.BAXDEF;
}

"\end{"axdef"}"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EAXDEF;
}

"\begin{"gendef"}" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.BGENAXDEF;
}

"\end{"gendef"}"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EGENAXDEF;
}

"\begin{"schema"}" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.BSCHDEF;
}

"\end{"schema"}"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.ESCHDEF;
}

"\begin{"genschema"}" {
    return Parser.BGENSCHDEF;
}

"\end{"genschema"}" {
    return Parser.EGENSCHDEF;
}

"\where"          {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.WHERE;
}

"\ST" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.WHERE;
}

"\power"          {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.POWER;
}

"\pset" {

```

```

    yyparser.yylval = new ParserVal(yytext());
    return Parser.POWER;
}

"\\cross"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.CROSS;
}

"\\prod"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.CROSS;
}

"\\defs"    {
    return Parser.DEFS;
}

"::="      {
    return Parser.DDEF;
}

"\\ddef"    {
    return Parser.DDEF;
}

"\\in"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.IN;
}

"\\mem"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.IN;
}

"\\neq"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}

"\\nem"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}

"\\nmem"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}

"\\notin"   {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}

"\\subseteq" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}

"\\subset"  {

```

```

    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"<" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"\\leq" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"\\geq" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
">" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"\\prefix" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"\\suffix" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"\\inseq" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"\\inbag" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"\\ires" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}
"\\extract" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}
"\\subbageq" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}
"\\partition" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}

```

```

}

"\\rel"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.REL;
}

"\\fun"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\pfun"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\tfun"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\inj"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\pinj"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\tinj"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\surj"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\psur"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\psurj"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\bij"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"\\ffun"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}
}

```

```

"\\finj"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INGEN;
}

"["           {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

"]"          {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

"\\lsch"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.LSBRACK;
}

"\\rsch"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.RSBRACK;
}

"\\\\\\\\"    {
    return Parser.NL;
}

"\\also"     {
    return Parser.NL;
}

"\\backslash" {
    return Parser.NL;
}

"\\bigcap"   {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\dint"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\dinter"   {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\bigcup"   {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\duni"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

```

```

"\\dunion"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\bool"      {
}

"\\dcat"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\Delta"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.DELTA;
}

"\\dom"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\emptybag"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EMPTYBAG;
}

"\\emptyseq"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EMPTYSEQ;
}

"\\emptyset"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EMPTYSET;
}

"\\exi"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EXISTS;
}

"\\exists"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EXISTS;
}

"\\zexi"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EXISTS;
}

"\\exists_1"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EXIONE;
}

"\\exione"   {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EXIONE;
}

```

```

"\\false"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.FALSEITY;
}

"\\all"        {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.FORALL;
}

"\\forall"     {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.FORALL;
}

"\\zall"       {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.FORALL;
}

"\\fovr"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.INFUN;
}

"\\zovr"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.INFUN;
}

"\\zfor"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.REN;
}

"\\head"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.PREGEN;
}

"\\hide"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.HIDE;
}

"\\zhide"     {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.HIDE;
}

"\\iff"       {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.BIMPLIES;
}

"\\zeq"       {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.BIMPLIES;
}

"\\imp"       {

```



```

    yyparser.yylval = new ParserVal(yytext());
    return Parser.IMPLIES;
}

"\\implies" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.IMPLIES;
}

"\\zimp" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.IMPLIES;
}

"\\lambda" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.LAMBDA;
}

"\\and" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.AND;
}

"\\land" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.AND;
}

"\\zand" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.AND;
}

"\\wedge" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.AND;
}

"\\lang" {
    return Parser.LDATA;
}

"\\lbag" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.LBAG;
}

"\\ldata" {
    return Parser.LDATA;
}

"\\lim" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.LIMG;
}

"\\lnot" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.NOT;
}

```

```

"\\znot" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.NOT;
}

"\\or" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.OR;
}

"\\vee" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.OR;
}

"\\lor" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.OR;
}

"\\zor" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.OR;
}

"\\nat" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.UNSIGNEDNUMBER;
}

"\\natone" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.UNSIGNEDNUMBER;
}

"\\negate" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.NEGATE;
}

"\\num" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.UNSIGNEDNUMBER;
}

"\\integer" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.UNSIGNEDNUMBER;
}

"\\pipe" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PIPE;
}

"\\project" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PROJECT;
}

"\\zproject" {
    yyparser.yylval = new ParserVal(yytext());

```

```

    return Parser.PROJECT;
}

"\\ran" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\rang"
    return Parser.RDATA;
}

"\\rbag" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.RBAG;
}

"\\rdata" {
    return Parser.RDATA;
}

"\\ring" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.RIMG;
}

"\\semi" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.SEMI;
}

"\\zcmp" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.SEMI;
}

"\\sdef" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.SDEF;
}

"@" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.SPOT;
}

"\\bullet" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.SPOT;
}

"\\dot" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.SPOT;
}

"\\spot" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.SPOT;
}

"\\succ" {

```

{

```

    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\squash" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\supset" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}

"\\supseteq" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INREL;
}

"\\tail" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\theta" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.THETA;
}

"\\true" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.TRUTH;
}

"\\Xi" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.XI;
}

"\\psetone" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\id" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\fset" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\finset" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\fsetone" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

```

```

}

"\\seq"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.PREGEN;
}

"\\seqone"   {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.PREGEN;
}

"\\iseq"     {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.PREGEN;
}

"\\lseq"     {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.LSEQ;
}

"\\rseq"     {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.RSEQ;
}

"\\last"     {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.PREGEN;
}

"\\head"    {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.PREGEN;
}

"\\bag"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.PREGEN;
}

"\\map"      {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.INFUN;
}

"\\mapsto"   {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.INFUN;
}

"\\upto"     {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.INFUN;
}

"+"         {
  yyparser.yylval = new ParserVal(yytext());
  return Parser.INFUN;
}

```

```

"_" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\cup" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\uni" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\union" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\setminus" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\cat" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\uplus" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\uminus" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"*" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\div" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\mod" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\cap" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\int" {
    yyparser.yylval = new ParserVal(yytext());

```

```

    return Parser.INFUN;
}

"\\inter" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\sres" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\filter" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\comp" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\fcmp" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\cmp" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\circ" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\otimes" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\oplus" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\bagcount" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\dres" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\rres" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}
}

```

```

"\\ndres"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\dsub"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\nrres"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.INFUN;
}

"\\inv"       {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.POSTFUN;
}

"\\star"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.POSTFUN;
}

"\\plus"      {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.POSTFUN;
}

[A-Za-z]([a-zA-Z0-9]|\\|_)* {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.WORD;
}

[0-9]+        {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.UNSIGNEDNUMBER;
}

([!?] | _[0-9])+ {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.DECOR;
}

"|"          {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.BAR;
}

"\\bbar"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.BAR;
}

"\\cbar"     {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.BAR;
}

"\\zbar"     {

```



```

    yyparser.yylval = new ParserVal(yytext());
    return Parser.BAR;
}

"(" {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

")" {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

"{" {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

"}" {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

"" {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

"." {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

"\\eq" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EQ;
}

"\\Leftrightarrow" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EQ;
}

"=" {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EQ;
}

";" {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

":" {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

",," {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

```

```

}

"/"    {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.REN;
}

"^"    {
    yyparser.yylval = new ParserVal(yytext());
    return (int) yycharat(0);
}

[\\n]+ {
    lineCount();
}

[ \\r]+ {
}

"\\_"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.DUMMY;
}

"\\t1" {
}

[ \\t] {
}

"\\#"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.PREGEN;
}

"\\["  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.BTYPE;
}

"\\]"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.EPYTB;
}

"\\{"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.SET;
}

"\\}"  {
    yyparser.yylval = new ParserVal(yytext());
    return Parser.TES;
}

.      {
    String strErr = new String("Lexical_error_on_line:_"+
        (line+1)+"_:_" + yytext());
    System.err.println(strErr);
    System.exit(1);
}

```

Appendix F

BYACC/J Specification: Parser.y

```
%{
    import java.io.*;
    import java.util.*;
    import java.lang.Object;
}%

%token <sval> UNSIGNEDNUMBER WORD DECOR EMPTYSEQ IN EXISTS BAR INGEN
%token <sval> EMPTYSET POWER DELTA XI INFUN LSEQ RSEQ FORALL
%token <sval> CROSS PREGEN BIMPLIES IMPLIES AND NOT OR RBAG TRUTH
%token <sval> LSBRACK RSBRACK LIMG EQ REN HIDE DUMMY LBAG EMPTYBAG
%token <sval> RIMG REL FALSITY SEMI LAMBDA INREL EXIONE SPOT
%token BZED EZED BAXDEF EAXDEF BSCHDEF ESCHDEF BGENSCHDEF POSTIFUN
%token NL SDEF WHERE TES LDATA DDEF DEFS NEGATE PIPE SET THETA
%token PROJECT RDATA BTYPE EPYTB BGENAXDEF EGENAXDEF PIPE EGENSCHDEF

%type <sval> expr0 expr2.item expr3 expr4 '[' ']' expr2.word word.decor
%type <sval> expr word.tname.list sch.def tname.list expr.word
%type <sval> expr1 expr.list ident decl.part axiom.part schema.ref
%type <sval> expr2 definition.def schema.text '(' ';' pred1 rename.opt
%type <sval> decl.name decl.name.list rel.item rel.chain expr2.chain
%type <sval> decor sign zed.text body.schema def.lhs pred gen.actual.opt
%type <sval> gen.formals declaration basic.decl.word set.expr
%type <sval> rename.list ',' expr1.word body.schema.word schema.exp
%type <sval> rename.head.pred.sch basic.decl.let.def ')' '\\}'
%type <sval> abbrev.def ''' gen.ax.def word.schema.text '\\{'
%type <sval> word.decl.name expr4.word rel.chain.tail expr3.word
%type <sval> word.decl.name.list decl.part.word word.pred1
%type <sval> gen.actual.opt.rename.opt word.sq.bracks expr0.word
%type <sval> head.gen.actual.opt.rename.opt spot.tail
%type <sval> head.sign.gen.actual.opt.rename.opt ':'

%left      PIPE
%left      SEMI
%left      HIDE
%left      PROJECT
%left      BIMPLIES
%right     IMPLIES
%left      OR
%left      AND
```

```

%nonassoc      NOT
%nonassoc      '( '
%right         ') '

%right         INGEN REL
%left         INFUN
%left         CROSS
%left         ', '
%left         SPOT
%left         EQ IN INREL

%%
specification: section
  | /* empty */
  ;

section: paragraph
  {
    if (!err_found){
      System.out.println("Parsing_is_successful.");
    }
  }
  | section paragraph
  ;

paragraph: no.box
  | box
  ;

no.box: BTYPE gen.formals EPYTB
  | BZED zed.text EZED
  ;

box: BAXDEF body.schema.word EAXDEF
  | BGENAXDEF gen.ax.def EGENAXDEF
  | BGENSCHDEF gen.sch.def EGENSCHDEF
  | BSCHDEF sch.def ESCHDEF
  ;

zed.text: gen.formals
  | not.basic.type.list
  {
    if (schCal) schCal = false;
  }
  ;

body.schema: decl.part
  {
    if (genAxDef){
      genAxDef = false;
    }
    else if (schDef){
      schDef = false;
    }
  }
  | decl.part.word WHERE axiom.part
  ;

body.schema.word: body.schema
  | WORD
  {
    $$ = $1;
  }

```

```

        if (genAxDef){
            llGenAxDefVar.add($$);
        }
        else if (schDef){
            llVariables.add($$);
        }
    }
    ;

decl.part.word: decl.part
    {
        if (genAxDef){
            genAxDef = false;
        }
        else if (schDef){
            schDef = false;
        }
    }
| WORD
    {
        $$ = $1;
        if (genAxDef){
            llGenAxDefVar.add($$);
            genAxDef = false;
        }
        else if (schDef){
            llVariables.add($$);
            schDef = false;
        }
    }
    ;

gen.ax.def: '{' word.tname.list '}'
    {
        $$ = $2;
        genAxDef = true;
    }
    body.schema.word
    ;

gen.sch.def: '{' WORD '}' '{' word.tname.list '}' body.schema.word
    ;

word.tname.list: tname.list
    {
        $$ = $1;
    }
| WORD
    {
        $$ = $1;
    }
    ;

sch.def: '{' WORD '}'
    {
        $$ = $2;
        schDef = true;
    }
    body.schema.word
    ;

not.basic.type.list: not.basic.type

```

```

        | not.basic.type.list sep
        {
            if (schCal) separator = true;
        }
        not.basic.type
    ;

not.basic.type: definition.def
    | schema.def.horz
    | abbrev.def
    ;

definition.def: def.lhs DEFS expr.word
    | WORD gen.formals DEFS expr.word
    | let.def
    ;

def.lhs: PREGEN decor ident
    | ident REL decor ident
    | ident INGEN decor ident
    ;

ident: WORD decor
    {
        $$ = ($1+$2).trim();
        if (genAxDef){
            llGenAxDefVar.add($$);
        }
        else if (schDef) {
            llVariables.add($$);
        }
    }
    ;

decor: DECOR
    {
        $$ = $1;
    }
    | /* empty */
    {
        $$ = "";
    }
    ;

schema.def.horz: WORD SDEF
    {
        schCal = true;
        if (separator){
            llSchCal.add("separator");
            separator = false;
        }
    }
    schema.exp
    | WORD gen.formals SDEF
    {
        schCal = true;
        if (separator){
            llSchCal.add("separator");
            separator = false;
        }
    }
    schema.exp

```

```

;
abbrev.def: ident DDEF data.rhs.list
;
tname.list: word.tname.list ',' WORD
{
    $$ = $1 + $2 + $3;
}
;
decl.part: basic.decl
| basic.decl.word sep decl.part.word
;
axiom.part: pred sep axiom.part
| pred
;
sep: ';'
| NL
;
schema.exp: head.pred.sch schema.exp
| word.schema.exp1
;
data.rhs.list: data.rhs
| data.rhs.list BAR data.rhs
;
data.rhs: ident
| ident LDATA expr.word RDATA
;
expr0: LAMBDA spot.tail
{
    $$ = $1 + $2;
}
| expr
{
    $$ = $1;
}
;
expr: expr1
{
    $$ = $1;
}
;
expr1: expr1.word REL decor
{
    isRelation = true;
    if (genAxDef){
        hmTypeVar.put(llGenAxDefVar.getLast().toString(),"isRelation");
        llGenAxDefTypes.add($2);
        if (!$3.isEmpty())
            llGenAxDefTypes.add($3);
    }
    else if (schDef){
        hmTypeVar.put(llVariables.getLast().toString(),"isRelation");

```

```

        llTypes.add($2);
        if (!$3.isEmpty()) llTypes.add($3);
    }
}
expr1.word
| expr1.word INGEN decor
{
    isFunction = true;
    if (genAxDef){
        hmTypeVar.put(llGenAxDefVar.getLast().toString(),"isFunction");
        llGenAxDefTypes.add($2);
        if (!$3.isEmpty())
            llGenAxDefTypes.add($3);
    }
    else if (schDef){
        hmTypeVar.put(llVariables.getLast().toString(),"isFunction");
        llTypes.add($2);
        if (!$3.isEmpty()) llTypes.add($3);
    }
}
expr1.word
| expr2.chain
{
    $$ = $1;
    if (!isFunction){
        isRelation = true;
        if (genAxDef){
            hmTypeVar.put(llGenAxDefVar.getLast().toString(),"isRelation");
        }
        else if (schDef){
            hmTypeVar.put(llVariables.getLast().toString(),"isRelation");
        }
    }
}
| expr2
{
    $$ = $1;
    if (!isFunction && !isRelation){
        isConstant = true;
        if (genAxDef){
            hmTypeVar.put(llGenAxDefVar.getLast().toString(),"isConstant");
        }
        else if (schDef){
            hmTypeVar.put(llVariables.getLast().toString(),"isConstant");
        }
    }
}
;

expr1.word: expr1
{
    $$ = $1;
}
| WORD
{
    $$ = $1;
    if (genAxDef){
        llGenAxDefTypes.add($$);
    }
    else if (schDef){
        llTypes.add($$);
    }
}

```



```

    }
    ;
expr2.chain:  expr2.item
    {
        $$ = $1;
    }
|  expr2.chain CROSS
    {
        if (genAxDef){
            llGenAxDefTypes.add($2);
        }
        else if (schDef){
            llTypes.add($2);
        }
    }
    expr2.item
;

expr2:  expr2.word INFUN decor
    {
        if (genAxDef){
            llGenAxDefTypes.add($2);
            if (!$3.isEmpty())
                llGenAxDefTypes.add($3);
        }
        else if (schDef){
            llTypes.add($2);
            if (!$3.isEmpty()) llTypes.add($3);
        }
    }
    expr2.word
|  POWER expr4.word
    {
        $$ = $1 + "┌" + $2;

        if (genAxDef){
            llGenAxDefTypes.set(llGenAxDefTypes.
                size()-1,$1+ "┌"+llGenAxDefTypes.getLast().toString());
        }
        else if (schDef){
            llTypes.set(llTypes.size()-1,$1+ "┌"+llTypes.getLast().toString());
        }
    }
|  PREGEN decor expr4.word
    {
        $$ = $1 + "┌" + $2 + $3;

        if (genAxDef){
            if (!$2.isEmpty()) llGenAxDefTypes.
                set(llGenAxDefTypes.size()-1, $1+$2+"┌"+
                    llGenAxDefTypes.getLast().toString());
            else llGenAxDefTypes.set(llGenAxDefTypes.size()-1,$1+"┌"+
                llGenAxDefTypes.getLast().toString());
        }
        else if (schDef){
            if (!$2.isEmpty()) llTypes.
                set(llTypes.size()-1,$1+$2+"┌"+llTypes.getLast().toString());
            else llTypes.set(llTypes.size()-1,$1 +
                "┌"+ llTypes.getLast().toString());
        }
    }
}

```

```

| DUMMY decor expr4.word
{
  $$ = $1 + "_" + $2 + $3;
  if (genAxDef){
    if (!$2.isEmpty()) llGenAxDefTypes.
      set(llGenAxDefTypes.size()-1,$1+$2+"_"+llGenAxDefTypes.getLast().toString());
    else llGenAxDefTypes.set(llGenAxDefTypes.size()-1,$1+"_"+
      llGenAxDefTypes.getLast().toString());
  }
  else if (schDef){
    if (!$2.isEmpty()) llTypes.set(llTypes.size()-1,$1+$2+"_"+
      llTypes.getLast().toString());
    else llTypes.set(llTypes.size()-1,$1+
      "_" + llTypes.getLast().toString());
  }
}
| expr4.word LIMG
{
  if (genAxDef){
    llGenAxDefTypes.add($2);
  }
  else if (schDef){
    llTypes.add($2);
  }
}
expr0.word RIMG decor
{
  if (genAxDef){
    llGenAxDefTypes.add($5);
    if (!$6.isEmpty())
      llGenAxDefTypes.add($6);
  }
  else if (schDef){
    llTypes.add($5);
    if (!$6.isEmpty()) llTypes.add($6);
  }
}
| expr3
{
  $$ = $1;
}
;

expr2.word: expr2
| WORD
{
  if (genAxDef){
    llGenAxDefTypes.add($1);
  }
  else if (schDef) {
    llTypes.add($1);
  }
}
;

expr4.word: expr4
| WORD
{
  if (genAxDef){
    llGenAxDefTypes.add($1);
  }
  else if (schDef){

```

```

        llTypes.add($1);
    }
}
;

expr0.word: expr0
| WORD
{
    if (genAxDef){
        llGenAxDefTypes.add($1);
    }
    else if (schDef){
        llTypes.add($1);
    }
}
;

expr2.item: expr2.word CROSS
{
    if (genAxDef){
        llGenAxDefTypes.add($2);
    }
    else if (schDef){
        llTypes.add($2);
    }
}
expr2.word
;

expr4: word.decor
{
    $$ = $1;
}
| word.decor word.sq.bracks
{
    $$ = $1 + $2;
}
| word.decor gen.actual.opt
{
    $$ = $1 + $2;
}
| UNSIGNEDNUMBER
{
    $$ = $1;
    if (genAxDef){
        llGenAxDefTypes.add($1);
    }
    else if (schDef){
        llTypes.add($1);
    }
}
| SET TES
{
    $$ = "EMPTYSET";
}
| SET set.expr
| EMPTYSET
{
    $$ = $1;
}
| schema.ref
{

```

```

        $$ = $1;
    }
| LSEQ expr.word RSEQ
  {
    $$ = $1 + $2 + $3;
  }
| LSEQ expr.list RSEQ
  {
    $$ = $1 + $2 + $3;
  }
| EMPTYSEQ
  {
    $$ = $1;
  }
| LSEQ RSEQ
  {
    $$ = "EMPTYSEQ";
  }
| LBAG expr.word RBAG
  {
    $$ = $1 + $2 + $3;
  }
| LBAG expr.list RBAG
  {
    $$ = $1 + $2 + $3;
  }
| EMPTYBAG
  {
    $$ = $1;
  }
| LBAG RBAG
  {
    $$ = "EMPTYBAG";
  }
| '(' expr.word ')'
  {
    $$ = $1 + $2 + $3;
  }
| '(' expr.list ')'
  {
    $$ = $1 + $2 + $3;
  }
| theta.chain rename.opt
| theta.chain
| expr4.word '.' ident
| expr4.word POSTFUN decor
| '(' LAMBDA spot.tail ')'
;

expr3: expr3.word expr4.word
  {
    $$ = $1 + $2;
  }
| expr4
  {
    $$ = $1;
  }
;

expr3.word: expr3
  | WORD
  {

```

```

        $$ = $1;
        if (genAxDef){
            llGenAxDefTypes.add($$);
        }
        else if (schDef){
            llTypes.add($$);
        }
    }
;

schema.ref: head.gen.actual.opt.rename.opt
{
    $$ = $1;
}
| head.sign.gen.actual.opt.rename.opt
{
    $$ = $1;
}
| WORD gen.actual.opt.rename.opt
{
    $$ = $1 + "_" + $2;
    if (genAxDef){
        llGenAxDefVar.add($$);
    }
    else if (schDef){
        llVariables.add($$);
    }
}
;

head.gen.actual.opt.rename.opt: WORD ""
{
    $$ = $1 + $2;
    if (genAxDef){
        llGenAxDefVar.add($$);
    }
    else if (schDef){
        llVariables.add($$);
    }
}
| WORD ""
gen.actual.opt.rename.opt
{
    $$ = $1 + $2 + "_" + $3;
    if (genAxDef){
        llGenAxDefVar.add($$);
    }
    else if (schDef){
        llVariables.add($$);
    }
}
;

head.sign.gen.actual.opt.rename.opt: sign WORD
{
    $$ = $1 + "_" + $2;
    if (genAxDef){
        llGenAxDefVar.add($$);
    }
    else if (schDef){
        llVariables.add($$);
    }
}

```

```

    }
    | sign WORD gen.actual.opt.rename.opt
    {
        $$ = $1 + $2 + "_" + $3;
        if (genAxDef){
            llGenAxDefVar.add($$);
        }
        else if (schDef){
            llVariables.add($$);
        }
    }
    }
;

gen.actual.opt.rename.opt: word.sq.bracks
{
    $$ = $1;
}
| gen.actual.opt
{
    $$ = $1 ;
}
| rename.opt
{
    $$ = $1;
}
| word.sq.bracks rename.opt
{
    $$ = $1 + "_" + $2;
}
| gen.actual.opt rename.opt
{
    $$ = $1 + "_" + $2;
}
}
;

word.sq.bracks : '[' WORD ']'
{
    $$ = $1 + $2 + $3;
}
;

set.expr: word.schema.text
{
    $$ = $1;
}
TES
| spot.tail
{
    $$ = $1;
}
TES
| expr TES
{
    $$ = $1;
}
| expr.list TES
{
    $$ = $1;
}
}
;

spot.tail: word.schema.text SPOT expr.word

```

```

        {
            $$ = $1;
        }
;

expr.list: expr.word ',' expr.word
        {
            $$ = $1 + $2 + $3;
        }
| expr.list ',' expr.word
        {
            $$ = $1 + $2 + $3;
        }
;

gen.actual.opt: '[' expr ']'
        {
            $$ = $1 + $2 + $3;
        }
| '[' expr.list ']'
        {
            $$ = $1 + $2 + $3;
        }
;

theta.chain: THETA ident
;

rename.opt: '[' rename.list ']'
        {
            $$ = $1 + $2 + $3;
        }
;

rename.list: rename
        {
            $$ = $1;
        }
| rename.list ',' rename
        {
            $$ = $1 + $2 + $3;
        }
;

rename: WORD " " REN WORD " "
        {
            $$ = $1 + $2 + "_" + $3 + "_" + $4 + $5;
        }
| word.decl.name REN word.decl.name
        {
            $$ = $1 + "_" + $2 + "_" + $3;
        }
;

decl.name: word.decor
        {
            $$ = $1;
        }
;

word.decl.name: decl.name
        {

```

```

        $$ = $1;
    }
| WORD
  {
    $$ = $1;
    if (genAxDef){
      llGenAxDefVar.add($$);
    }
    else if (schDef){
      llVariables.add($$);
    }
  }
;

word.decor: WORD DECOR
  {
    $$ = ($1+$2).trim();
    if (genAxDef){
      llGenAxDefVar.add($$);
    }
    else if (schDef){
      llVariables.add($$);
    }
  }
;

sign: DELTA
  {
    $$ = $1;
  }
| XI
  {
    $$ = $1;
  }
;

schema.text: declaration
  {
    $$ = $1;
  }
| declaration BAR pred
  {
    $$ = $1 + "_" + $2 + "_" + $3;
  }
| WORD BAR pred
  {
    $$ = $1 + "_" + $2 + "_" + $3;
  }
;

declaration: basic.decl
  {
    $$ = $1;
  }
| declaration ';' basic.decl.word
  {
    $$ = $1 + "_" + $2 + "_" + $3;
  }
;

pred: head.pred.sch pred
  {

```



```

        $$ = $1 + "_" + $2;
    }
| pred1
  {
    $$ = $1;
  }
| WORD
  {
    $$ = $1;
  }
;

basic.decl.word: basic.decl
  {
    $$ = $1;
  }
| WORD
  {
    $$ = $1;
    if (genAxDef){
      llGenAxDefVar.add($$);
    }
    else if (schDef){
      llVariables.add($$);
    }
  }
;

basic.decl: schema.ref
  {
    String str;
    if (schDef){
      if (!hmVarDecl.containsKey(llVariables.getLast().toString())){
        hmVarDecl.put(llVariables.getLast().toString(), "null");
      }
    }
  }
| WORD "\"" ':' expr.word
  {
    String str;
    if (genAxDef){
      $$ = $1+" ";
      llGenAxDefVar.add($$);
      for (int i=0;i<llGenAxDefTypes.size();i++){
        if (hmGenAxDef.containsKey(
          llGenAxDefVar.getLast().toString())){
          str = hmGenAxDef.get(getLast().
            toString()).toString();
          hmGenAxDef.put(llGenAxDefVar.getLast().
            toString(), str + "," +
            llGenAxDefTypes.get(i).toString());
        }
        else hmGenAxDef.put(llGenAxDefVar.getLast().
          toString(), llGenAxDefTypes.get(i).toString());
      }
      llGenAxDefVar.removeAll(llGenAxDefVar);
      llGenAxDefTypes.removeAll(llGenAxDefTypes);
    }
    else if (schDef){
      $$ = $1+" ";
    }
  }

```

```

llVariables.add($$);
for (int i=0;i<llTypes.size();i++){
    if (hmVarDecl.containsKey(
        llVariables.getLast().toString())){
        str = hmVarDecl.get(llVariables.getLast().
            toString()).toString();
        hmVarDecl.put(llVariables.getLast().
            toString(), str + "," +
            llTypes.get(i).toString());
    }
    else hmVarDecl.put(llVariables.getLast().
        toString(), llTypes.get(i).toString());
}
llVariables.removeAll(llVariables);
llTypes.removeAll(llTypes);
}
else $$ = $1+"'" + "┘" + $3 + "┘" + $4;

isFunction = false;
isRelation = false;
isConstant = false;
}
| word.decl.name.list ':' expr.word
{
String str;
if (genAxDef){
    $$ = $1;
    for (int i=0;i<llGenAxDefTypes.size();i++){
        if (hmGenAxDef.containsKey(
            llGenAxDefVar.getLast().toString())){
            str = hmGenAxDef.get(llGenAxDefVar.getLast().
                toString()).toString();
            hmGenAxDef.put(llGenAxDefVar.getLast().
                toString(), str + "," +
                llGenAxDefTypes.get(i).toString());
        }
        else hmGenAxDef.put(llGenAxDefVar.getLast().
            toString(), llGenAxDefTypes.get(i).toString());
    }
    llGenAxDefVar.removeAll(llGenAxDefVar);
    llGenAxDefTypes.removeAll(llGenAxDefTypes);
}
else if (schDef){
    $$ = $1;
    for (int i=0;i<llTypes.size();i++){
        if (hmVarDecl.containsKey(
            llVariables.getLast().toString())){

            str = hmVarDecl.get(llVariables.getLast().
                toString()).toString();
            hmVarDecl.put(llVariables.getLast().
                toString(), str + "," +
                llTypes.get(i).toString());
        }
        else hmVarDecl.put(llVariables.getLast().
            toString(), llTypes.get(i).toString());
    }
    llVariables.removeAll(llVariables);
    llTypes.removeAll(llTypes);
}
else $$ = $1 + "┘" + $2 + "┘" + $3;
}

```

```

        isFunction = false;
        isRelation = false;
        isConstant = false;
    }
;

decl.name.list : decl.name
{
    $$ = $1;
}
| WORD "''", 'WORD "'
{
    $$ = ($1+$2+$3+$4+$5).trim();
    if (genAxDef){
        llGenAxDefVar.add($$);
    }
    else if (schDef){
        llVariables.add($$);
    }
}
| word.decl.name.list ', ' WORD "'
{
    $$ = ($1+$2+$3+$4).trim();
    if (genAxDef){
        llGenAxDefVar.set(llGenAxDefVar.size()-1,
            llGenAxDefVar.getLast().toString()+$$);
    }
    else if (schDef){
        llVariables.set(llVariables.size()-1,
            llVariables.getLast().toString()+$$);
    }
}
| WORD "''", 'word.decl.name
{
    $$ = ($1+$2+$3+$4).trim();
    if (genAxDef){
        llGenAxDefVar.set(llGenAxDefVar.size()-1,
            llGenAxDefVar.getLast().toString()+$$);
    }
    else if (schDef){
        llVariables.set(llVariables.size()-1,
            llVariables.getLast().toString()+$$);
    }
}
| word.decl.name.list ', ' word.decl.name
{
    $$ = $1+$2+$3;
    if (genAxDef){
        llGenAxDefVar.set(llGenAxDefVar.size()-2,$$);
        llGenAxDefVar.removeLast();
    }
    else if (schDef){
        llVariables.set(llVariables.size()-2,$$);
        llVariables.removeLast();
    }
}
;

word.decl.name.list : decl.name.list
| WORD
{
    $$ = $1;
}

```

```

                if (genAxDef){
                    llGenAxDefVar.add($$);
                }
                else if (schDef){
                    llVariables.add($$);
                }
            }
        }
;

let .def: ident DEFS expr .word
;

gen.formals: '[' word.tname.list ']'
{
    $$ = $2 ;
}
;

head.pred.sch: FORALL
{
    if (schCal){
        llSchCal.add($1);
    }
}
word.schema.text SPOT
{
    if (schCal){
        llSchCal.add($4);
    }
}
| EXISTS
{
    if (schCal){
        llSchCal.add($1);
    }
}
word.schema.text SPOT
{
    if (schCal){
        llSchCal.add($4);
    }
}
| EXIONE
{
    if (schCal){
        llSchCal.add($1);
    }
}
word.schema.text SPOT
{
    if (schCal){
        llSchCal.add($4);
    }
}
;

word.schema.text: schema.text
{
    if (schCal){
        llSchCal.add($1);
    }
}

```

```

        | WORD
        {
            if (schCal){
                llSchCal.add($1);
            }
        }
    ;

pred1: rel.chain
    | schema.ref
    | TRUTH
    {
        $$ = $1;
    }
    | FALSITY
    {
        $$ = $1;
    }
    | NOT word.pred1
    {
        $$ = $1 + "_" + $2;
    }
    | word.pred1 AND word.pred1
    {
        $$ = $1 + "_" + $2 + "_" + $3;
    }
    | word.pred1 OR word.pred1
    {
        $$ = $1 + "_" + $2 + "_" + $3;
    }
    | word.pred1 IMPLIES word.pred1
    {
        $$ = $1 + "_" + $2 + "_" + $3;
    }
    | word.pred1 BIMPLIES word.pred1
    {
        $$ = $1 + "_" + $2 + "_" + $3;
    }
    | '( ' pred ')'
    {
        $$ = $1 + $2 + $3;
    }
    ;

word.pred1: pred1
    {
        $$ = $1;
    }
    | WORD
    {
        $$ = $1;
    }
    ;

rel.chain: expr.word rel.chain.tail
    {
        $$ = $1 + "_" + $2;
    }
    ;

expr.word: expr
    {

```

```

        $$ = $1;
    }
| WORD
  {
    $$ = $1;
    if (genAxDef){
        llGenAxDefTypes.add($$);
    }
    else if (schDef){
        llTypes.add($$);
    }
  }
;

rel.item: EQ
  {
    $$ = $1;
  }
| IN
  {
    $$ = $1;
  }
| INREL decor
  {
    $$ = $1+$2;
  }
;

rel.chain.tail: rel.item expr.word
  {
    $$ = $1 + "⊂" + $2;
  }
| rel.item rel.chain
  {
    $$ = $1 + "⊂" + $2;
  }
;

schema.exp1: LSBRACK
  {
    llSchCal.add($1);
  }
word.schema.text RSBRACK
  {
    llSchCal.add($4);
  }
| schema.ref
  {
    llSchCal.add($1);
  }
| NOT word.schema.exp1
  {
    llSchCal.add($1);
  }
| word.schema.exp1 AND word.schema.exp1
  {
    llSchCal.add($2);
  }
| word.schema.exp1 OR word.schema.exp1
  {
    llSchCal.add($2);
  }

```

```

| word.schema.exp1 IMPLIES word.schema.exp1
| {
|     llSchCal.add($2);
| }
| word.schema.exp1 BIMPLIES word.schema.exp1
| {
|     llSchCal.add($2);
| }
| word.schema.exp1 PROJECT word.schema.exp1
| word.schema.exp1 HIDE '(' WORD "" ')',
| {
|     llSchCal.add($2);
|     llSchCal.add($3);
|     llSchCal.add($4 + $5);
|     llSchCal.add($6);
| }
| word.schema.exp1 HIDE '('
word.decl.name.list ')'
| {
|     llSchCal.add($2);
|     llSchCal.add($3);
|     llSchCal.add($4);
|     llSchCal.add($5);
| }
| word.schema.exp1 SEMI word.schema.exp1
| {
|     llSchCal.add($2);
| }
| word.schema.exp1 PIPE word.schema.exp1
| '(' schema.exp ')'
| {
|     llSchCal.add($1);
|     llSchCal.add($3);
| }
| }
;

word.schema.exp1: schema.exp1
| WORD
| {
|     llSchCal.add($1);
| }
;

%%
/* declare & initialize global variables */
LinkedList llMatched = new LinkedList();
boolean err_found = false;
static LinkedList llVariables = new LinkedList();
static LinkedList llTypes = new LinkedList();
static LinkedList llGenAxDefVar = new LinkedList();
static LinkedList llGenAxDefTypes = new LinkedList();
static LinkedList llSchCal = new LinkedList();

static HashMap hmTypeVar = new HashMap();
static HashMap hmGenAxDef = new HashMap();
static HashMap hmVarDecl = new HashMap();

static boolean genAxDef = false, schDef = false, isFunction = false;
static boolean isRelation = false, isConstant = false, schCal = false;
static boolean separator = false;

int countVar = 0;

```

```

/* a reference to the lexer */
private ScannerCl lexer;

/* interface to the lexer */
private int yylex(){
    int yyl_return = -1;
    try {
        yylval = new ParserVal(0);
        yyl_return = lexer.yyex();
    }

    catch (IOException e){
        System.err.println("I/O_Error");
    }

    return yyl_return;
}

public void yyerror (String error) {
    int lineN = lexer.line;
    error = error + "\n" + "Please_check_line:_" + String.valueOf(lineN) +
        "\n" + lexer.yytext() + "_with_length:_" + lexer.yylength();
    putError(error);
}

public Parser (Reader r) {
    lexer = new ScannerCl(r, this);
}

public void putError(String err) {
    System.err.println(err);
    err_found = true;
    if (err.startsWith("syntax_error")){
        System.exit(0);
    }
}
}

```


Appendix G

fHead.tex and output_fHead.tex

The below specification is fHead.tex.

$[NAME]$	$\models [X]$
	$headSeq : (\text{seq}_1 X) \rightarrow X$
	$\forall s : \text{seq}_1 X \bullet headSeq(s) = s(1)$
$State$	$Init$
$name : \text{seq}_1 NAME$	$State'$
$Head$	
$\exists State; a! : NAME; b? : \text{seq}_1 \mathbb{N}; b!, c! : \mathbb{N}$	
$a! = headSeq[\text{seq}_1 NAME](name) \wedge b! = headSeq(b?)$	
$c! = headSeq(\langle 5, 2, 4 \rangle)$	

The below specification is output_fHead.tex.

$[NAME]$	
$headSeq : (\text{seq}_1 NAME) \rightarrow NAME$	$headSeq1 : (\text{seq}_1 \mathbb{N}) \rightarrow \mathbb{N}$
$\forall s : \text{seq}_1 NAME \bullet$ $headSeq(s) = s(1)$	$\forall s : \text{seq}_1 \mathbb{N} \bullet$ $headSeq1(s) = s(1)$
$headSeq2 : (\text{seq}_1 \mathbb{Z}) \rightarrow \mathbb{Z}$	
$\forall s : \text{seq}_1 \mathbb{Z} \bullet headSeq2(s) = s(1)$	
$State$	$Init$
$name : \text{seq}_1 NAME$	$State'$
$Head$	
$\exists State; a! : NAME; b? : \text{seq}_1 \mathbb{N}; b!, c! : \mathbb{N}$	
$a! = headSeq(name) \wedge b! = headSeq1(b?)$	
$c! = headSeq2(\langle 5, 2, 4 \rangle)$	

Appendix H

expandingschema_2.tex and its expanded schema

The first specification is `expandingschema_2.tex`.

[*COPY*, *BOOK*, *READER*]

| *maxloans* : \mathbb{N}

Report ::= *ReaderAlreadyRegistered* | *Ok* | *FurtherCopyAdded* | *NewTitleAdded*

Library

stock : *COPY* \rightarrow *BOOK*; *issued* : *COPY* \leftrightarrow *READER*
shelved : \mathbb{F} *COPY*; *readers* : \mathbb{F} *READER*

$\forall x : \text{COPY}; y1, y2 : \text{READER} \bullet$
 $(x \mapsto y1) \in \text{issued} \wedge (x \mapsto y2) \in \text{issued} \Rightarrow y1 = y2$
 $\text{shelved} \cup \text{dom issued} = \text{dom stock}$
 $\text{shelved} \cap \text{dom issued} = \emptyset$
 $\text{ran issued} \subseteq \text{readers}$
 $\forall r : \text{readers} \bullet \#(\text{issued} \triangleright \{r\}) \leq \text{maxloans}$

InitLibrary

Library'

shelved' = \emptyset
readers' = \emptyset

EnterNewCopy

Δ *Library*; *b?* : *BOOK*

$\exists c : \text{COPY} \mid c \notin \text{dom stock} \bullet$
 $(\text{stock}' = \text{stock} \oplus \{c \mapsto b?\} \wedge \text{shelved}' = \text{shelved} \cup \{c\})$
issued' = *issued*
readers' = *readers*

$AddCopyReport$ $\exists Library; b? : BOOK; rep! : Report$
$b? \in \text{ran } stock \Rightarrow rep! = FurtherCopyAdded$ $b? \notin \text{ran } stock \Rightarrow rep! = NewTitleAdded$

$$AddCopy \cong EnterNewCopy \wedge AddCopyReport$$

Below is an expanded schema of the above specification which was generated by our system.

$AddCopy$ $\Delta Library; b? : BOOK; rep! : Report$
$((\exists c : COPY \mid c \notin \text{dom } stock \bullet$ $(stock' = stock \oplus \{c \mapsto b?\} \wedge shelved' = shelved \cup \{c\})) \wedge$ $issued' = issued \wedge$ $readers' = readers)$ \wedge $(stock' = stock \wedge$ $issued' = issued \wedge$ $shelved' = shelved \wedge$ $readers' = readers \wedge$ $b? \in \text{ran } stock \Rightarrow rep! = FurtherCopyAdded \wedge$ $b? \notin \text{ran } stock \Rightarrow rep! = NewTitleAdded)$

Appendix I

expandingsch2_4.tex and its expanded schema

Below is the `expandingsch2_4.tex` specification.

$Response ::= okay \mid sorry$

$CarsPark$
$count : \mathbb{N}; maximum : \mathbb{N}$
$count \leq maximum$

$InitCarsPark$
$CarsPark$
$count = 0$ $maximum = 3$

$Enters$
$\Delta CarsPark$
$count < maximum$ $count' = count + 1$ $maximum' = maximum$

$NotEntered \hat{=} (\neg Enters)$

Below is an expanded schema of the above specification.

$NotEntered$
$count : \mathbb{Z}$ $count' : \mathbb{Z}$ $maximum : \mathbb{Z}$ $maximum' : \mathbb{Z}$
$(\neg (count \leq maximum) \vee$ $\neg (count' \leq maximum') \vee$ $\neg (count < maximum) \vee$ $\neg (count' = count + 1) \vee$ $\neg (maximum' = maximum) \vee$ $\neg (count \in (\mathbb{N})) \vee$ $\neg (count' \in (\mathbb{N})) \vee$ $\neg (maximum \in (\mathbb{N})) \vee$ $\neg (maximum' \in (\mathbb{N})))$

Appendix J

expandingsch5_2.tex and its expanded schema

Below is the `expandingsch5_2.tex` specification. This specification is as follows:

[MEMORY]

<i>Calculator</i> $store : MEMORY \rightarrow \mathbb{Z}$ $display : \mathbb{Z}$ $arg2 : \mathbb{Z}$

<i>Init</i> <i>Calculator</i> $\forall m : MEMORY \bullet store(m) = 0$ $display = 0$ $arg2 = 0$
--

<i>Add</i> $\Delta Calculator$ $store' = store$ $display' = display + arg2$
--

$AddI \cong Add[argument/arg2, screen/display]$

On the other hand, the below schema is an expanded schema of the above specification.

<i>AddI</i> $store : \mathbb{P}(MEMORY \times \mathbb{Z})$ $store' : \mathbb{P}(MEMORY \times \mathbb{Z})$ $screen : \mathbb{Z}$ $display' : \mathbb{Z}$ $argument : \mathbb{Z}$ $arg2' : \mathbb{Z}$
$store' = store \wedge display' = screen + argument \wedge$ $store \in (MEMORY \rightarrow \mathbb{Z}) \wedge store' \in (MEMORY \rightarrow \mathbb{Z})$

Appendix K

expandingsch6_1.tex and its expanded schema

The below specification is `expandingsch6_1.tex`.

[MEMORY]

<i>Calculator</i> $store : MEMORY \rightarrow \mathbb{Z}$ $display : \mathbb{Z}$ $arg2, arg21 : \mathbb{Z}$
--

<i>Init</i> <i>Calculator</i> $\forall m : MEMORY \bullet store(m) = 0$ $display = 0$ $arg2 = 0$
--

<i>Enter</i> $\Delta Calculator$ $value? : \mathbb{Z}$ <hr/> $store' = store$ $display' = value?$ $arg2' = display$
--

<i>Add</i> $\Delta Calculator$ <hr/> $store' = store$ $display' = display + arg21$

$AddI \hat{=} Add \setminus (arg2)$

Below is a new schema which was created from the above specification generated by our system.

<i>AddI</i> $store : \mathbb{P}(MEMORY \times \mathbb{Z})$ $store' : \mathbb{P}(MEMORY \times \mathbb{Z})$ $display : \mathbb{Z}$ $display' : \mathbb{Z}$ $arg2' : \mathbb{Z}$ $arg21 : \mathbb{Z}$ $arg21' : \mathbb{Z}$ <hr/> $\exists arg2 : \mathbb{Z} \bullet$ $store' = store \wedge display' = display + arg21 \wedge$ $store \in (MEMORY \rightarrow \mathbb{Z}) \wedge store' \in (MEMORY \rightarrow \mathbb{Z})$

Appendix L

expandingsch7_1.tex and its expanded schema

The below specification is similar to the above `expandingsch5_2` specification, but one new operational schemas have been added here.

<i>Enter</i> Δ Calculator $value? : \mathbb{Z}$
$store' = store$ $display' = value?$ $arg2' = display$

$Composition \cong Enter \circ Add$

The expanded schema which is shown as follows was generated by our system.

<i>Composition</i> Δ Calculator $value? : \mathbb{Z}$
$store' = store \wedge display' = value? + display$