Generative frameworks for rigorous model-driven development

Nuno Amálio

Submitted for the Degree of Doctor of Philosophy

Department of Computer Science
University of York

August 2006

Abstract

Our increasing reliance on software systems requires reliable software. Mainstream software manufacture, however, is not rigorous and precise, and resulting software lacks the desired reliability. Formal methods take a rigorous and precise approach to software development, delivering reliable software, but they are widely recognised as being impractical. Although the situation is improving, the costs of achieving reliable software are still high and formal methods are only used when absolutely necessary.

This thesis investigates how reuse can contribute to improving the practicality of formal modelling methods, and how diagrammatic modelling notations and formal methods can be integrated for rigorous, but practical development of software systems.

The thesis proposes GeFoRME, an approach to building domain-specific frameworks based on templates that make a combined use of formal methods and diagrammatic modelling languages. To express templates, the thesis develops the language FTL, which generates sentences of some language upon instantiation and enables proof with templates of formal models.

The thesis then develops $UML + Z$ a framework for object-oriented modelling for general purpose sequential systems that combines UML and Z. FTL is used to build $UML + Z$'s catalogue of templates and meta-theorems, which generates Z models and simplifies their proofs of consistency. The thesis also develops an approach to formal analysis with snapshot diagrams.

Acknowledgements

It seems this it. This is the last bit I write in this thesis. It is now time to thank people and institutions.

I would like to thank Dr Fiona Polack and Professor Susan Stepney for guidance, encouragement, and useful feedback. They always showed enthusiasm and kept me motivated. I am sincerely grateful to them.

I would like to thank the Portuguese Foundation for Science and Technology that has funded me for over the last four years. They also provided funding to attend conferences. I would also like to thank the department of computer science of the university of York for the funding to attend international conferences, summer schools, and for the Gibbs-plessey award that took me to the Federal University of Pernambuco in Recife/Brazil for an academic visit.

In Recife, I am grateful to Professor Augusto Sampaio for the supervision and support while I was there. I had a very good time and I am very grateful to him for that. I would also like to thank Adalberto Farias, Dr Alexandre Mota and Dr Adolfo Duran for their friendship and support.

I would also like to thank: Professor Jim Woodcock for useful feedback in more than one occasion, and Dr Steve king for advice given in his role as assessor. Finally, I would also like to thank Filo Ottaway for the nice chats in portuguese over these years.

Last but not least. I would like to thank Chiara for her love, support, and to keep my mind away from the PhD over these last two years (If I am still sane I owe it to her); my parents for support, love and encouragement, my brother André and Evren, both for friendship and encouragement. And also to all my friends while here at York.

Nuno Amálio
August 2006

Valeu a pena? Tudo vale a pena
Se a alma não é pequena.
Quem quer passar além do Bojador
Tem que passar além da dor.
Deus ao mar o perigo e o abismo deu,
Mas nele é que espelhou o céu.

Fernando Pessoa, Mar Português, In Mensagem, 1934

Contents

1 Introduction **1**
- 1.1 Thesis hypothesis . . . 3
- 1.2 Scope . . . 3
  - 1.2.1 GeFoRME . . . 4
  - 1.2.2 UML+Z . . . 6
- 1.3 Outline . . . 7

2 Background: models, frameworks and patterns **9**
- 2.1 Software modelling and their languages . . . 10
  - 2.1.1 A model of software modelling languages . . . 11
  - 2.1.2 Model Refinement . . . 12
  - 2.1.3 Views and diagrams . . . 13
- 2.2 Semi-formal methods . . . 13
  - 2.2.1 Structured Methods . . . 15
  - 2.2.2 Object-Oriented Methods . . . 15
  - 2.2.3 UML . . . 15
  - 2.2.4 Discussion . . . 17
- 2.3 Formal methods . . . 20
  - 2.3.1 Historical Notes . . . 21
  - 2.3.2 Techniques of formal modelling . . . 22
  - 2.3.3 Formal Refinement . . . 23
  - 2.3.4 Model Analysis . . . 24
  - 2.3.5 Modelling Languages . . . 25
  - 2.3.6 Discussion . . . 28
- 2.4 Integrating semi-formal and formal methods . . . 30
  - 2.4.1 Approaches to the integration . . . 31
  - 2.4.2 Structured methods . . . 32
  - 2.4.3 Object-Oriented Methods . . . 33
  - 2.4.4 Discussion . . . 35
- 2.5 Reuse, domains and frameworks . . . 38
  - 2.5.1 Domain engineering and frameworks . . . 39
  - 2.5.2 Modelling frameworks . . . 40
  - 2.5.3 Discussion . . . 40
- 2.6 Patterns . . . 41
  - 2.6.1 Patterns in formal development . . . 42
  - 2.6.2 Describing patterns formally . . . 43
  - 2.6.3 Discussion . . . 43
- 2.7 Conclusions . . . 45

3 A formal template language enabling meta-proof 47
3.1 Motivation . . . 48
3.1.1 Meta-Proof . . . 48
3.2 A short introduction to FTL . . . 50
3.3 The Formal Definition of FTL . . . 51
3.3.1 Syntax . . . 52
3.3.2 Semantics . . . 52
3.3.3 Illustration: Instantiating templates using the semantics . . . 57
3.3.4 Testing . . . 58
3.4 The instantiation calculus . . . 58
3.4.1 Placeholders . . . 59
3.4.2 Lists . . . 60
3.4.3 Choice . . . 65
3.4.4 Testing . . . 66
3.5 Meta-proof . . . 66
3.5.1 Linking FTL with Z . . . 67
3.5.2 Characteristic instantiation and the rule of generalisation . . . 67
3.5.3 Illustration: meta-proof using the semantics . . . 68
3.5.4 Illustration: meta-proof using the instantiation calculus . . . 69
3.5.5 A logic for template-Z . . . 70
3.5.6 Proof with the template-Z logic . . . 71
3.6 Discussion . . . 73
3.7 Related Work . . . 74
3.8 Conclusions . . . 74

4 ZOO: an Object-Oriented style for Z 77
4.1 Core concepts of the OO paradigm . . . 78
4.2 Views and views structuring in Z . . . 79
4.3 The structure of the ZOO style . . . 79
4.3.1 Object and class . . . 80
4.3.2 Association . . . 80
4.3.3 System . . . 81
4.3.4 The views of the ZOO style . . . 82
4.4 Consistency Checking in ZOO . . . 82
4.5 The *UML+Z* template catalogue . . . 83
4.6 Generation of ZOO models . . . 84
4.7 The Bank case study . . . 84
4.7.1 Problem and UML diagrams . . . 85
4.8 ZOO model: state space . . . 86
4.8.1 Structural View . . . 87
4.8.2 Class View . . . 88
4.8.3 Relational View . . . 90
4.8.4 Global View . . . 91
4.9 ZOO model: operations . . . 92
4.9.1 Class view . . . 92
4.9.2 Relational View . . . 98
4.9.3 Global View . . . 98

4.10 Discussion . . . 102
4.10.1 Object-orientation in Z . . . 102
4.10.2 $UML + Z$ Templates catalogue . . . 103
4.11 Related Work . . . 104
4.12 Conclusions . . . 105

5 Inheritance in the ZOO style 107
5.1 Inheritance and related concepts . . . 107
5.1.1 Behavioural Inheritance and data refinement . . . 109
5.2 Inheritance in ZOO . . . 109
5.2.1 Inheritance Hierarchy . . . 110
5.2.2 Subclass specialisation . . . 111
5.2.3 Abstract Class . . . 113
5.2.4 Polymorphism . . . 113
5.2.5 Overview . . . 114
5.3 Behavioural inheritance and refinement . . . 115
5.4 Behavioural inheritance in the intensional view . . . 115
5.4.1 Specialising general Z data refinement for behavioural inheritance . . . 116
5.4.2 The restrictions of refinement . . . 117
5.4.3 Reaching a compromise: relaxing the refinement constraints . . . 118
5.5 Inheritance in the extensional view . . . 120
5.5.1 Promoted class operations . . . 120
5.5.2 Polymorphic operations . . . 121
5.6 Multiple inheritance in ZOO: the Queues case study . . . 122
5.6.1 Structural view . . . 122
5.6.2 Class, intensional view . . . 123
5.6.3 Class, extensional view . . . 125
5.6.4 Global View . . . 129
5.6.5 Discussion . . . 130
5.7 The Bank case study with inheritance . . . 131
5.7.1 ZOO model, state space . . . 132
5.7.2 ZOO model, operations . . . 139
5.7.3 Global View . . . 148
5.7.4 Discussion . . . 150
5.8 Discussion . . . 151
5.9 Related Work . . . 152
5.10 Conclusions . . . 154

6 Snapshot analysis 155
6.1 The analysis technique . . . 155
6.2 Snapshot analysis of the Bank case study . . . 157
6.2.1 Analysis of State space with single snapshots . . . 158
6.2.2 Analysing system constraints . . . 159
6.2.3 Analysing of operations with snapshot pairs . . . 162
6.2.4 Analysis of operations with snapshot sequences . . . 167
6.3 Discussion . . . 171
6.4 Related Work . . . 173

6.5 Conclusions . . . 174

7 Evaluation, Conclusions and Future Work 177
7.1 The Hypothesis . . . 177
7.1.1 "Possible" . . . 177
7.1.2 "Flexible and Practical" . . . 178
7.1.3 "Worth doing" . . . 180
7.2 Related Work: GeFoRME and $UML + Z$ . . . 181
7.3 Future Work . . . 181

References 183

A Formal definition of FTL and Instantiation Calculus 205
A.1 Syntax . . . 205
A.1.1 BNF Definition . . . 205
A.1.2 Z Definition . . . 206
A.2 Semantics . . . 206
A.2.1 Definitions in Equational Style . . . 206
A.2.2 Z Definition . . . 208
A.3 The Instantiation Calculus . . . 210
A.3.1 Definitions in Equational style . . . 210
A.3.2 Z Definition . . . 212

B Inference rules 217
B.1 Formal Proof, sequents, conjectures and inference rules . . . 217
B.2 Sequent Calculus . . . 218
B.3 Propositional Calculus . . . 218
B.4 Predicate Calculus . . . 218
B.5 Z Schema Calculus . . . 218
B.6 Template-Z inference rules . . . 219
B.6.1 Axiom rules . . . 219
B.6.2 Derived Rules . . . 219

C ZOO Generics toolkit 221
C.1 ZOO Domain Toolkit . . . 221
C.2 Other Generics . . . 225

D $UML + Z$ templates catalogue 227
D.1 Structural View . . . 228
D.2 Class View . . . 229
D.2.1 Intensional View . . . 229
D.2.2 Extensional View . . . 242
D.3 Relational View . . . 255
D.4 Global View . . . 258
D.5 Snapshots . . . 265

E Z Simulation Rules for Behavioural Inheritance **269**
E.1 A brief overview of Z's theory of data refinement . . . 269
E.1.1 Relational data refinement and simulation . . . 269
E.1.2 Z data refinement . . . 270
E.2 Deriving Z Simulation Rules for Behavioural Inheritance . . . 271
E.2.1 The simpler case, no communication . . . 272
E.2.2 Embedding Inputs and Outputs . . . 273
E.3 Adding operations to the subclass: Relaxing . . . 273

F ZOO Models **275**
F.1 Fully generated Bank . . . 275
F.1.1 Structural View . . . 275
F.1.2 Class Customer . . . 276
F.1.3 Class Account . . . 277
F.1.4 Association Holds . . . 280
F.1.5 Global View . . . 280
F.2 Bank with inheritance . . . 280
F.2.1 Structural View . . . 281
F.2.2 Class Customer . . . 281
F.2.3 Class Account . . . 282
F.2.4 Class Current . . . 284
F.2.5 Class Savings . . . 286
F.2.6 Class WBased . . . 289
F.2.7 Class BalBased . . . 292
F.2.8 Class Savings, Polymorphic definitions . . . 294
F.2.9 Class Account, Polymorphic definitions . . . 295
F.2.10 Association Holds . . . 295
F.2.11 Global View . . . 296
F.3 Inheritance Bank after snapshot analysis . . . 298
F.3.1 Global View . . . 298

G Proofs **301**
G.1 Proofs of behavioural Inheritance in ZOO . . . 301
G.1.1 ADTs with no communication . . . 301
G.1.2 ADTs with communication . . . 310
G.1.3 Relaxation for extra subclass operations . . . 317
G.2 Proof of template inference rules . . . 322
G.3 Meta-proofs . . . 325
G.3.1 Intensional View . . . 325
G.3.2 Extensional View . . . 329
G.3.3 Relational View . . . 331
G.3.4 Global View . . . 332

List of Figures

1.1 In a GeFoRME framework, diagrams are views of an underlying formal model. 4
1.2 The ingredients of a GeFoRME framework. 5
1.3 In a GeFoRME framework, formal models are instantiated from templates with information coming from diagrams and provided by the user. 5
1.4 Models in the $UML + Z$ framework. 6

2.1 Each software modelling description (a member of Syn, the syntactic domain) denotes one set of programs (subset of Sem, the semantic domain). 12
2.2 Model refinement: as the model is *refined*, its meaning-set becomes more restricted, until there is just one possible implementation. 13
2.3 A model described using two different notations. The models in $SynA$ denote solid-line blobs, and the ones in $SynB$ dashed-line ones. The relation between the two is set intersection. 14
2.4 A modelling language is defined by a meta-model, which, in turn, is defined using a meta-language. 16
2.5 The process of developing formal models. A model is written, type-checked, and then checked for the satisfaction of properties (through proof) and animated. 23
2.6 The process of developing an integrated model. The semi-formal model is written, and then formalised to produce a formal model. The formal model is then developed further and analysed. 31

3.1 Templates and the sets of strings they denote (all possible instances of a template). 53
3.2 A template and a legal substitution yields a string. 53
3.3 Templates transformed with the instantiation calculus and the sets of strings they denote. 59
3.4 Possible ($S(t)$) and well-formed instantiations ($WFS(t)$) of a Z template (t). 67
3.5 A template denotes a set of Z specifications, set $WFS(t)$ (an empty set of denotations is represented without an arrow). 68
3.6 A Z template denotes one meta-Z object and a set of Z specifications. A meta-Z object denotes a set of Z specifications. 71

4.1 Z views and their composition using Z schema conjunction. 79
4.2 The set of all object atoms of class A, OA, the set of all possible object states StA (class intension), and the mapping from existing objects to their current states (class extension). 80
4.3 The sets of all object atoms of classes A and B, and the tuples of the association. 81
4.4 A system composed of classes A and B, and one association between them. . 81

4.5 The views of ZOO and dependency relationships (arrow means dependency). 82
4.6 The UML class diagram of the trivial Bank system. 85
4.7 The UML statechart of the Account class in the trivial Bank system. 87

5.1 The *Shape* inheritance hierarchy. 108
5.2 An invalid inheritance hierarchy. 110
5.3 An inheritance hierarchy of Queues: Queue (unbounded queue), BQueue (bounded queue) and RQueue (resettable queue). 110
5.4 The set of all object atoms of the class A, and its subclasses B and C; A includes its exclusive objects (oAs) and the objects of its subclasses (oBs and oCs). Each class includes a set of all possible states of its objects ($STAs$, $STBs$, $STCs$), the class intension (the state of subclasses extends its superclasses). The extension defines a mapping function from existing object atom to its state; each class has its own mapping function. 114
5.5 The multiple-inheritance model of Queues. 122
5.6 The class diagram of the Bank system with inheritance. 131

6.1 The sets of all possible instances of a $UML+Z$ model (rectangle), and all valid instances (oval). 156
6.2 An operation is a relation between sets of valid model instances. 156
6.3 All possible model instances, valid instances, and valid instances that satisfy the precondition of some operation. 157
6.4 Invalid, spurious and reachable instances of a model. 157
6.5 Snapshot of state. A personal customer with a current account. 158
6.6 An invalid snapshot: the total balances are negative (constraint C2). 159
6.7 A savings account with a negative balance (constraint C1). 160
6.8 A current account with a negative balance above allowed overdraft (constraint C2). 160
6.9 Snapshot of an invalid state: a company with a savings account. 161
6.10 Snapshot of a customer with two current accounts. 161
6.11 Snapshot-pair: a customer is added to the system. 163
6.12 Snapshot-pair: a current account is created for an existing bank customer. . . 164
6.13 A savings account is created for an existing bank customer that does not hold a current account with bank. 165
6.14 A savings account is created for an existing bank customer that already holds a current account with bank. 165
6.15 Snapshot-pair: a savings account being created for a *company* customer. . . . 166
6.16 A current account being created for the wrong customer. 167
6.17 An attempt to withdraw money from an account that is *suspended*. 168
6.18 A personal customer with only a savings account. 170
6.19 A savings account is opened for a customer, followed by the closure of a current account. The customer remains with just one savings account. 171

List of Tables

4.1 *Full* and *partial* generation in the templates of each view of the ZOO style. . 85
4.2 The global operations of the trivial bank system. 86
4.3 The constraints of the trivial bank system. 86
4.4 Preconditions of system operations (calculated with meta-theorems and Z/Eves). 101

5.1 The extra operations of the extended bank system. 132
5.2 The extra constraints of the extended bank system. 132
5.3 Preconditions of system operations (calculated with meta-theorems and Z/Eves). 149

7.1 Effectiveness of meta-proof in proving the consistency of a ZOO model. . . . 179

Declaration

I hereby declare that the contents of this thesis are the result of my own original contribution, except where otherwise stated. The following material, presented in this thesis, has been previously published:

- *Modular UML Semantics: Interpretations in Z Based on Templates and Generics.* Nuno Amálio, Susan Stepney and Fiona Polack. Int. Workshop on Formal Aspects of Component Software. Hung Dang Van and Zhiming Liu (eds). UNU/IIST Technical Report 284. 2003

- *Formal Proof From UML Models.* Nuno Amálio, Susan Stepney and Fiona Polack. ICFEM 2004, Seattle, USA. Jim Davies et al (eds). vol 3308 of LNCS. Springer. 2004.

- *An Object-Oriented Structuring for Z based on Views.* Nuno Amálio, Fiona Polack and Susan Stepney. ZB 2005: Int. Conf. of B and Z users. vol 3455 of LNCS. Helen Treharne et al. Springer. 2005.

- *Frameworks based on templates for rigorous model-driven development.* Nuno Amálio. In IFM 2005: Integrated Formal Methods, doctoral symposium, Eindhoven, The Netherlands. Tech. Report, Dept. of Mathematics and Computer Science of the Technische Universiteit Eindhoven, 2005.

- *UML+Z: UML augmented with Z.* Nuno Amálio, Fiona Polack and Susan Stepney. In *Software Specification Methods: an overview using a case study.* Marc Frappier and Henri Habrias (eds). Hermes Science, 2006.

- Nuno Amálio, Susan Stepney and Fiona Polack. *A formal template language enabling metaproof.* In FM 2006: International Symposium of formal methods, Canada. vol 4085 LNCS. Springer. 2006.

- Nuno Amálio, Fiona Polack and Susan Stepney. *Frameworks based on templates for rigorous model-driven development.* In Special issue on the IFM2005 doctoral symposium. Electronic Notes in Theoretical Computer Science, 2006 (to appear).

But the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing costs of programming error.
[...] the cost of error in certain types of program may be almost incalculable — a lost spacecraft, a collapsed building, a crashed aeroplane or a world war.

Anthony Hoare [Hoa69]

1 Introduction

Software plays a crucial role in our modern world and is, more and more, part of our everyday life. Our reliance on software systems is due to increase in years to come as software spreads and grows ubiquitous. From personal computer to cars, planes and household appliances to become part of our environment, and from a secondary but crucial bookkeeping role to a primary role in the control of transportation, energy distribution, communications, banking and health care to become a critical component of our infrastructure. But, can the software that we use be trusted?

Today's software is more reliable than many had predicted [Hoa69, Hoa96, Mac96, Mac01]. Nevertheless, software-related failures are taking their toll [Cha05]. The number of software projects that fail before reaching completion is alarmingly high [RAE03]. A recent study estimated that the total wastage arising from IT project failures is around US$150 billion in the United States and US$140 billion in the European Union [RAE03]. One could even expect software manufacture to be like a process of natural selection, the projects that do complete delivering quality software, but evolution in software manufacture has still a long way to go: existing software is plagued with expensive defects. The press release of a recent study [NIS02a] on the effects of software defects says [NIS02b]:

> Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $ 59.5 billion annually, or about 0.6 percent of the gross domestic product, according to a newly released study commissioned by the National Institute of Standards and Technology (NIST). More than half of the costs are borne by software users, and the remainder by software developers/vendors.
>
> The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated $ 22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects. [...] Currently, over half of all errors are not found until "downstream" in the development process or during post-sale software use.

Governments are increasingly aware of our dependency on software, and the importance of its reliability. Recently, in response to increasing customer dissatisfaction, the US government legislated in favour of software consumer protection, minimum standards of software quality and suppliers' responsibilities [KP98, JK04, BSW]. The European Commission has also raised its concerns regarding the quality of software [Red05]:

> [...] Given that we now entrust vital aspects of our lives to software systems; quality and reliability is a social and economic prerequisite. In extreme cases, software faults can be catastrophic, costing human lives even. Such failures and vulnerability to cyber-criminals would undermine trust in the Information Society and the services that it delivers.
>
> We need urgently, therefore, to master complexity, so as to ensure that the systems that run industrial and societal applications remain dependable.

Despite the recent attention from the media and governments, the problems of software quality and software related disasters are not new. For years, the discipline of software engineering has been studying software disasters, and the complexity of software and its development. Since the 1970s, classic software engineering books have told (with some humour) tales of software development failures [Bro75]. The so-called *software crisis* that motivated the establishment of the software engineering discipline was 40 years ago. Despite some improvements, we seem to be making the same mistakes and increasing demand for software just makes the problem worse.

The NIST study (above) suggests that improved *program testing* may lead to better software. Although this is true, program testing is ineffective at a timely detection of software defects that cost much more if not discovered until later [Boe76]. In fact, mainstream software manufacture, with its recurring practice of *trial and error*, already suffers from its premature insistence on code and program testing. The problem is that code is expensive, it has too much detail, and is not at the right level of abstraction to help thinking about the problem and the design of its solution. Only through a careful consideration of the problem can the timely detection of those costly errors be possible; the level of abstraction needs to be raised.

It is this need for further abstraction that drives the increasing use of models in software engineering. This is the idea behind *model-driven development*, an approach that advocates models, rather than code, as the primary artifacts of software development. Models are used to describe the problem or the design of its solution. Models are analysed to uncover flaws and expose issues related with requirements and design. However, mainstream modelling practice is not rigorous. The modelling notations have a very vague semantics, they are not precise and they do not enable mechanical analysis of models. The result is that models can be written using these notations, but their meaning is not clear and there is no way to mechanically exercise them. It is like writing a program and not being able to execute it. The techniques based on these notations are called *semi-formal* methods and their major weakness (as it is will be explained in detail in the next chapter) is their lack of a sound mathematical basis.

For years, researchers have been developing mathematical techniques to model software. *Formal methods*, as these techniques are known, are inspired by the way mature engineering disciplines build their artifacts: based on *prediction and calculation* with sound mathematical theories. However, they are not widespread, being mainly used in the development of only a small fraction of the systems that are developed today; those where failures can have catastrophic consequences: *critical systems*.

In critical systems, reliability is the major concern. Other kinds of systems, however, are driven by other priorities, such as *time to market*, *feature count* and *cost of production*. The problem with formal methods, as they stand today, is that they clearly conflict with such priorities. Formal methods are notorious for being hard, requiring substantial effort in formal modelling and verification, they are only effectively usable by highly-skilled experts and they are not well integrated in traditional software development processes. Yet, it is difficult to see an improvement in software reliability without the use of formal methods.

This thesis tries to contribute to a wider application of formal modelling methods by addressing some of their shortcomings.

1.1 Thesis hypothesis

The hypothesis of this thesis is as follows:

> **It is possible to integrate formal methods into mainstream software modelling practice. This can be done in a flexible and practical way, so that the integration has an engineering value and is actually worth doing.**

The hypothesis actually comprises three sub-hypothesis: (a) the possible, (b) the flexible and practical, and (c) the worth doing. This thesis tries to gather evidence in their favour.

As the next chapter shows, there have been many approaches that perform such integrations with very positive results. So, before the work reported here even started, there was already strong evidence in favour of sub-hypothesis (a) (it is possible). This thesis tries to make (a) even more evident, and to make a stronger argument for both (b) and (c): it is possible to do it in a flexible and practical way so that is actually worth doing.

There are different ways to achieve the goal of practicality. In most cases, this is tackled through tool support: the aim is to get more automation by using computer tools. This effort is of utmost importance. This thesis, however, explores another approach: *reuse*. Given that formal methods are notorious for being hard and demanding too much effort in both modelling and analysis, the aim is to factor this effort so that it can be reused many times. Reuse does not conflict in any way with tool-support, both approaches are important and they complement each other.

An approach based on reuse assumes that different developments have something to share. Experience shows that this is the case. Commonalities can be identified even in systems with very diverse application domains and the more similar the application domains the more their systems have in common.

In this thesis, reuse is driven by the ideal of *correctness by construction*. If the factored reusable pieces are correct (they satisfy certain desired properties), then this correctness is preserved whenever those pieces are used, not requiring any further correctness checks.

Below, the overall approach followed in this thesis to support the hypothesis is explained.

1.2 Scope

The goals of this thesis have been set above: follow a rigorous approach to model-driven development that is flexible and practical. The rigour will be given by the use of formal

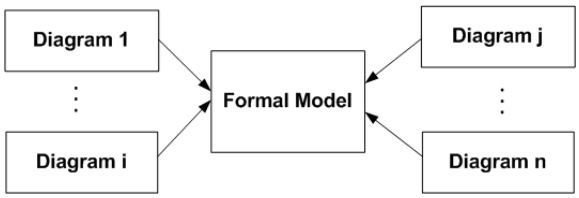


Figure 1.1: In a GeFoRME framework, diagrams are views of an underlying formal model.

methods and the key to practicality will be reuse. To realise these goals, this thesis proposes a general approach to build environments for rigorous model development, GeFoRME, and builds one such environment, $UML + Z$. These are discussed below.

1.2.1 GeFoRME

This thesis proposes an approach to build environments for *rigorous but practical* model-driven development. This approach is called GeFoRME: Generative Frameworks of Rigorous Model Engineering.

GeFoRME builds environments for engineers to construct, analyse and refine models of software systems. These environments are called frameworks and they provide an infrastructure to enable model development for some *problem domain*. The idea is to factor the commonalities of some *problem domain* so that they can be reused. GeFoRME frameworks are a large-scale unit of reuse, a whole modelling environment for some problem domain; they comprise a collection patterns, which are smaller reuse units.

GeFoRME is designed to facilitate an integration of formal methods within mainstream model development. So, each framework combines diagrams of mainstream modelling languages (such as the UML) with formal modelling languages. The diagrams act like a graphical interface for the formality that lies beneath. The formal method gives rigour and diagrams are accessible to a wide range of developers who do not need to be formal methods experts. In fact, GeFoRME frameworks enable the construction of formal models from diagrams. Diagrams constitute a partial description of some model. In GeFoRME, they are views of an underlying formal model (figure 1.1). In most cases, a collection of diagrams does not provide all the information of a model because not all properties can be expressed diagrammatically.

GeFoRME's core unit of reuse is the *template*, a very concrete form of pattern. To represent templates, this thesis proposes the Formal Template Language (FTL) and to achieve the goal of *correctness by construction* it proposes a *meta-proof* approach for FTL, which enables reasoning at the level of templates (e.g. calculating a pre-condition or proving an initialisation theorem) to establish *meta-theorems*. The templates and meta-theorems of some GeFoRME framework are assembled in a catalogue, so that every sentence of a formal model is generated by instantiating a template and, if applicable, proof related with the instantiated structure is simplified by instantiating a meta-theorem.

Figure 1.2 depicts the ingredients of a GeFoRME framework. The modelling notations of a GeFoRME framework comprise a set of diagram types and formal modelling languages. To support model-based development, GeFoRME comprises three key components, which define the framework's approach to modelling, analysis and refinement. To generate formal models and refinements, there is a catalogue of FTL templates and meta-theorems.

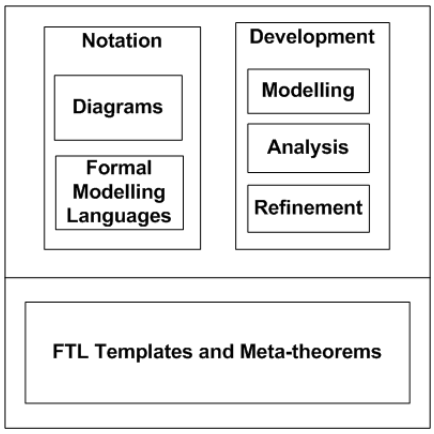


Figure 1.2: The ingredients of a GeFoRME framework.

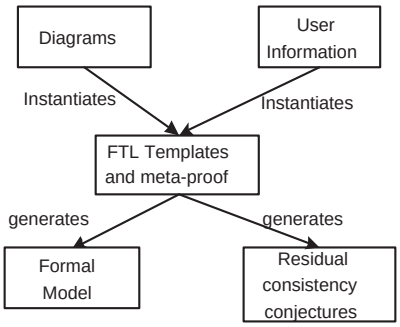


Figure 1.3: In a GeFoRME framework, formal models are instantiated from templates with information coming from diagrams and provided by the user.

The modelling component defines the diagrammatic vocabulary of the framework, the structure of the formal models, and how diagrams are represented in the formal model. The representation of diagrams in the formal model defines their semantics, which is done by defining a semantic mapping following the approach to denotational semantics [Ten76]. The set of diagram types of the framework constitutes the *syntactic domain* (every diagram of the framework's models is an instance of these diagram types), the catalogue of FTL templates captures the structure of the *semantic domain*, and the *semantic mapping* maps diagrams to template instantiations to generate formal models.

Figure 1.3 illustrates the process of model development in a GeFoRME framework. The user draws diagrams and needs to provide extra information for those properties that cannot be expressed diagrammatically. This is used to instantiate templates from the framework's catalogue to generate a formal model. The meta-theorems of the instantiated templates are also instantiated, resulting in residual model-consistency conjectures (the required conjectures are either fully discharged or simplified), which need to be proved to demonstrate the consistency of the model.

The analysis component defines the strategy to analyse models of the framework, preferably with diagrams. The refinement component includes a catalogue of model transforma-

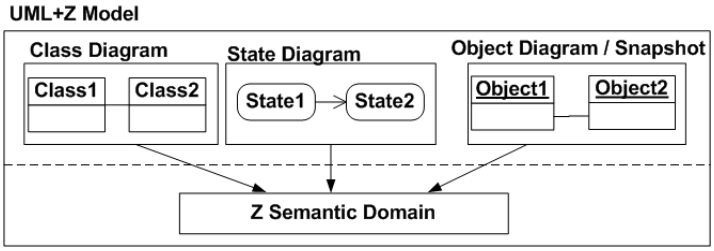


Figure 1.4: Models in the $UML + Z$ framework.

tions, represented as templates together with correctness meta-theorems, so that models can be transformed using the refactorings of the catalogue, and their correctness proofs simplified by appeal to meta-theorems.

1.2.2 UML+Z

To experiment with GeFoRME and evaluate the hypothesis, this thesis constructs a GeFoRME instance: the $UML+Z$ framework. As the name suggests, $UML+Z$ combines the mainstream modelling language UML with the formal modelling language Z. $UML + Z$ uses UML class and statecharts diagrams to build models, and object diagrams to analyse them, which are all represented in Z (figure 1.4).

The three main components of $UML + Z$ are discussed below.

Modelling

The $UML + Z$ framework comprises a Z semantic domain to express object-oriented (OO) models. This semantic domain (or Z style) is called ZOO. Every diagram of a $UML+Z$ model is mapped into ZOO.

$UML + Z$ also includes a catalogue of templates, which capture the structure of ZOO, together with their meta-theorems, which simplify consistency proofs. Every Z sentence of a model is generated by instantiating one of the templates of the catalogue.

Analysis

$UML + Z$'s analysis approach is based on Catalysis [DW98] snapshots and formal proof, and it is called *snapshot-based* analysis.

A snapshot, an object diagrams, is an instance of a class diagram and represents one state of the modelled system. Analysis draws snapshots to explore the model; proof then checks whether a given snapshot represents a valid system state or not.

Refinement

The refinement component of $UML + Z$ is left for future work and is not developed further in this thesis. The aim is to define a strategy to refine $UML + Z$ models, based on the theory of refinement for Z, and some example model transformations. The idea is to use FTL to capture refactorings and to explore meta-proof to reduce the proof overhead associated with these refactorings. The process is similar to the one followed in modelling: (a) refactorings and

associated correctness conjectures are captured with templates; (b) meta-proof is applied upon these representations to simplify (and in some cases fully prove) correctness conjectures. This will allow model transformations to be carried out by instantiating templates: the associated correctness proofs can then be simplified to smaller proofs after applying the associated meta-theorems.

1.3 Outline

This thesis motivates and develops the ideas outlined above. It has a total of 6 chapters and 7 appendices. This chapter introduced the thesis. The subsequent chapters are as follows:

Chapter 2 — *Background: models, frameworks and patterns.* This chapter surveys the literature in the area of software modelling, frameworks and patterns, and motivates the research pursued by the thesis. It examines the different approaches to modelling, and what has been done to achieve an adequate balance between rigour and practicality.

Chapter 3 — *A formal template language enabling meta-proof.* This chapter defines FTL (Formal Template Language) and its approach to *meta-proof*. It motivates the need for template representations, presents the rationale behind the design of FTL and formally defines the language. It also defines the language's approach to meta-proof, and defines a draft logic for proof with template representations of Z.

Chapter 4 — The core of the ZOO style. This chapter starts developing the ZOO style, the semantic domain of the $UML + Z$ framework. It defines the core of ZOO, which includes structures such as *class*, *association* and *system*, and illustrates how it can be used to build $UML + Z$ models by instantiating templates of the catalogue.

Chapter 5 — Inheritance in the ZOO style. This chapter extends the ZOO style with OO inheritance. It develops an approach to represent OO hierarchies in ZOO, and to check their behavioural conformance (according to the Liskov principle). In ZOO, behavioural conformance is grounded on the theory of data refinement for Z; new proof rules are derived by specialising the rules of general Z data refinement. The chapter then illustrates inheritance with $UML + Z$ models instantiated from templates of the catalogue.

Chapter 6 — Snapshot analysis. This chapter presents an approach to analyse $UML+Z$ models based on snapshots (object diagrams) and formal proof. Object diagrams are instances of some system, so the idea is to use these instances to explore the model, by checking whether those instances are valid (satisfied by the model) or not. The approach is illustrated with the $UML + Z$ case study developed in the previous two chapters, showing how snapshots can be used to analyse state, the effect of operations, and the effect of sequences of operations.

Chapter 7 — *Evaluation, Conclusions and Future Work.* This is the final chapter of the thesis. It evaluates the hypothesis, draws the conclusions and suggests future work.

Successful system development in the future will revolve around visual representations. We will first conceptualise, [...] as a series of increasingly more comprehensive models represented in an appropriate combination of visual languages. A combination it must be, since system models have several facets, each of which conjures up different kinds of mental images.

David Harel [Har92]

There is no effective way to test for ambiguity. Even if several people read the description and find that it is unambiguous, it is possible that they have different interpretations.

David Parnas [Par77]

2

Background: models, frameworks and patterns

Software engineering was founded in the late 1960s in response to the problems of large scale software development (the *software crisis*) [Som01, Mac01]. It was introduced with the aim of basing software manufacture on theoretical foundations and disciplined practice, as is done in mature engineering disciplines.

Since its foundation, software engineering has seen its most significant advances achieved through the use of *abstraction*. For instance, modern programming languages abstract away from the details of underlying machine language, providing high-level programming constructs that are intelligible to humans, facilitate human thought, and, consequently, improve the programming activity [Sha84, Wat04, Bar98]. Empirical studies show that high-level programming languages have contributed to increase both the programmer's productivity and the quality of the end-product [Bro75, Wei71].

Despite these and other advances, forty years on, software engineering practice is still far from mature. It is excessively code-oriented; and, unlike mature engineering disciplines, makes little use of calculation and prediction, and pays little attention to requirements (understand the problem) and design. Consequently, software manufacture is expensive and the end-product is often disappointing.

Essentially, most software systems are still built on a *trial and error* basis. This has rather unpleasant and expensive consequences. It is likely that pertinent system issues related to requirements or design will only come to the surface when the system is implemented and some interaction with the customer has occurred. Hence, the consequences that Boehm [Boe76] demonstrated still apply, namely: software manufacture is both expensive and inefficient because the later in the life-cycle some problem is addressed the more expensive it is to correct. This also has a negative impact on the quality of the end-product. The late addressing of issues and the continuous process of change, together with time to market constraints, contribute to a substantial *erosion*. It is likely that the end-product has lost one of its fundamental attributes, that of *conceptual integrity*, even before the system is put into production.

Clearly, this contrasts with a mature engineering practice where *models* play a crucial

role. Models are descriptions or representations of something[1]. Traditional engineering uses models to describe selected aspects of the real world, those models are then used to design and implement artifacts [Hal90a]. Ideally, engineering models are based on mathematics because they allow us to infer properties of the engineering artifact and the real world by using reasoning and calculation. The whole engineering modelling process is sketched by Hoare [Hoa99]: "Mature engineering starts with a mathematical model of the customer's requirements, decides the general strategy and structure of the solution, and then, with the aid of calculation derives the content and detail of the design, including optimisation and relevant parameters." A similar use of models is also made in other scientific disciplines, such as physics and economics. In economics for instance, mathematical models are used to infer properties of the current state of economies.

In software engineering, the use of models aims to address the problems identified by Boehm [Boe76]. So, models are used to describe both the problem (requirements) and the solution (design) in order to gain a better understanding of the issues involved. Once a model has been constructed it can be *analysed* to uncover flaws and expose fundamental issues. This role of models cannot possibly be assumed by programs (or code), because code is an expensive liability, and "as a repository of domain knowledge, environmental assumptions, design rationale or even required behaviour, code is a poor second to a carefully constructed model" [JR00].

The idea is not new, but there is a recent trend towards more use of models in mainstream circles of software engineering. This is the goal of model-driven development (MDD) [Sch06], which tries to alleviate the complexity of software development by using models.

This chapter surveys the state of the art in MDD. It starts by discussing modelling and modelling languages in general. Then, it discusses semi-formal and formal modelling languages in particular. Next, it shows the work that has been done to integrate these two classes of languages. Then, it discusses two promising approaches that can help MDD practice: frameworks and patterns. Finally, it draws conclusions, motivating the following chapters.

2.1 Software modelling and their languages

Software models are also called *specifications*, a term that is also common in engineering. In engineering, a specification is a precise statement of the requirements that a product must satisfy. It is used to describe the problem to be solved and for communication among engineers [Par77]. A specification forms a bridge between requirements and programs [Jac95b]. This is why specifications or models of software are seen as *contracts* between a client and an implementor: the contract says everything that the customer wants and everything that the implementor must do [Lam89, Mor94]. The roles of a contract may change during the development; in requirements it is a contract between the client of the system and the team that is going to implement it; at the level of design it is a contract between the designers (client) and the programmers (implementor). But how should we write models or specifications?

Models, like any other description, must be written using some language (or notation). A language consists of a *syntax* and a *semantics*. Syntax defines the structure and appearance of the language sentences, whose meaning (or interpretation) is defined by semantics. The difference between the two is better understood by considering numerals and numbers. Numerals are the symbols that are used to represent numbers and a collection of such symbols

[1] This corresponds to the *analytic* meaning of the word *model*, there are other meanings [Jac95b].

makes a notation; there are several numeric notations, such as roman, arabic, binary and decimal. This is syntax. The semantics is what these numeric symbols mean (or represent), which are the mathematical objects known as natural numbers. It so happens that the same number may be written in different ways, depending on the syntax being used, like 4 and *iv*. Over the years, the arabic notation was favoured over the roman one for common use, because it requires fewer symbols and it facilitates arithmetic calculation. But what should be, then, an appropriate software modelling notation?

Software models should be precise, abstract to avoid bias towards any specific implementation, and be written in terms of user-observable phenomena [Par77]. Clearly, natural languages cannot be used to write them, because they are prone to ambiguity [Par77, Mey85]. Programming languages cannot be used either, because programs describe one way to meet the requirements of a product, but not its actual requirements [Par77]. This means that we need specialised notations to model software systems.

Software modelling languages are divided into two big groups: *formal* and *semi-formal*. Formal languages are usually textual and have a formal (or mathematical) syntax and semantics. Semi-formal languages, on the other hand, have a formal syntax, which is usually diagrammatic, but no formal semantics. Semi-formal languages are not meaningless, it is just that their semantics is not mathematically defined.

2.1.1 A model of software modelling languages

As said above, numbers are the semantic universe of numerical notations. In software modelling languages, the semantic universe is made of programs, or, to be precise, mathematical representations of computer programs representing all possible solutions to some software problem. However, as said above, models or specifications often talk about the phenomena that are shared by the program and its environment [Jac95b] and that is why models often say something about the environment or the real world.

In formal terms (from [Win90]), a modelling language is a triple, $\langle\ Syn,\ Sem,\ Sat\ \rangle$. *Syn* and *Sem* are sets. They represent, respectively, the syntactic and semantic universe of the language, called syntactic and semantic domains. *Sat* is a relation between these two sets:

$$Sat : Syn \leftrightarrow Sem$$

Syn includes all possible descriptions of models expressible in the language, and *Sem* all possible programs denoted by the language. So, a modelling language provides a notation (syntactic domain), a universe of objects (semantic domain) and a precise rule defining which semantic objects satisfy each model description. That is:

$$\forall\, m : Syn;\ p : Sem \bullet Sat(m, p) \Leftrightarrow \text{p satisfies m}$$

Given a model m in *Syn* and some program p in *Sem*, then $Sat(m, p)$ if and only if p satisfies the model m (or p is a solution of m).

It is possible to construct a meaning function that is derived from the relation *Sat*, where each model denotes a set of programs (Figure 2.1):

$$fSat : Syn \rightarrow \mathbb{P}\ Sem$$

This is defined as:

$$\forall\, m : Syn \bullet fSat(m) = \{s \in Sem \mid Sat(m, s)\}$$

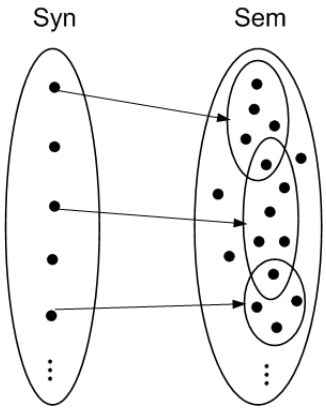


Figure 2.1: Each software modelling description (a member of *Syn*, the syntactic domain) denotes one set of programs (subset of *Sem*, the semantic domain).

A model m is consistent (or satisfiable) if its meaning-set is non empty ($fSat(m) \neq \varnothing$) and deterministic when its meaning-set is a singleton ($\#fSat(m) = 1$).

A modelling language may have many interpretations. These languages, sometimes termed ambiguous, have a key property: it is possible to define many different meaning functions (such as *fSat* above) or their equivalent satisfies relations (*Sat* above), where each function or relation corresponds to a possible interpretation of the language. This is often a characteristic of diagrammatic (or visual) languages (discussed in detail later).

A model has different levels of abstraction. A very abstract model is non-deterministic and leaves some things unspecified, hence it has many possible implementations. A more concrete model has a more restricted set of implementations, being closer to the final program. There is a delicate balance between saying just enough and saying too much in a model. If the model does not say enough then programmers may choose unacceptable programs. If the model says too much, then it is imposing implementation bias, leaving little design freedom. This leads to an important notion of software development: refinement.

2.1.2 Model Refinement

The fact that models may have different levels of abstraction gives the basis for a separation of concerns in the development. Models can be used at different phases of a software life-cycle, ranging from requirements (more abstract) to detailed design (more concrete). It also gives a basis for a stepwise approach to software development: abstract models are refined into more concrete ones in a stepwise manner, where each step carries some design decisions. This is known as *model refinement* [Wir71].

Figure 2.2 describes model refinement. A very abstract model denotes a very large (possibly infinite) set of programs. As the model is refined, the sets gets smaller and smaller, until the model cannot be refined any further (the final step has been reached) and the model denotes one single program. A refinement is *correct* if every behaviour (in a suitable sense) specified in the concrete model has an equivalent behaviour in the abstract model.

Formally, refinement is a (total) relation between model description [HHS86]:

$$\_ \sqsupseteq \_ : Syn \leftrightarrow Syn$$

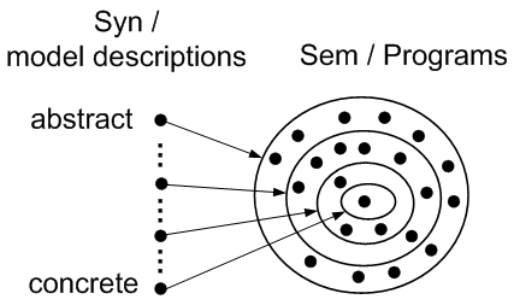


Figure 2.2: Model refinement: as the model is *refined*, its meaning-set becomes more restricted, until there is just one possible implementation.

Given models m_A (abstract) and m_C (concrete), then m_C refines m_A if the meaning-set of the concrete model is included in that of the abstract. That is:

$$\forall\, m_C, m_A : Syn \bullet m_C \sqsupseteq m_A \Leftrightarrow fSat(m_C) \subseteq fSat(m_A) \wedge fSat(m_C) \neq \varnothing$$

It is useful to see refinement in the context of contracts. A model sets a contract that says everything that its customer wants. A refinement of some contract must still satisfy the customer's demands, which is done by making some choices. For example, consider a contract that says: "You must take Mr. Smith from Gatwick Airport to the King's Cross railway station in London." Then if the implementor decides to take Mr. Smith to Kings Cross station by taxi, then that is surely an acceptable refinement of that contract, as it would be taking him by train, bus, or car. On the other hand, leaving Mr Smith at Victoria station, by whatever transport means, would not constitute a valid refinement.

2.1.3 Views and diagrams

In many situations, software engineers are required to use more than one modelling notation. Semi-formal techniques, for instance, provide several diagrammatic notations to describe different aspects of systems. In this case, such a diagram is a view: a *partial model*. A complete system model is defined by all the views.

Figure 2.3 depicts a model made of two views that are specified using different notations. There is just one semantic domain and the relation between the two notations is *set-intersection* (conjunction in [ZJ93]). Here, the meaning of a model is the intersection of the meaning-sets of the composing views, where a model is consistent if this intersection is not empty. There can be other kinds of relations between views, and the semantic domains may even be disjoint.

As said above, software modelling languages are divided in semi-formal and formal. The techniques that support software development based on them are called, respectively, semi-formal and formal methods. The next section discusses semi-formal methods in detail.

2.2 Semi-formal methods

In mature engineering disciplines, once a new mathematical technique proves to be effective at solving some problem, it is assimilated by engineers through education and training and

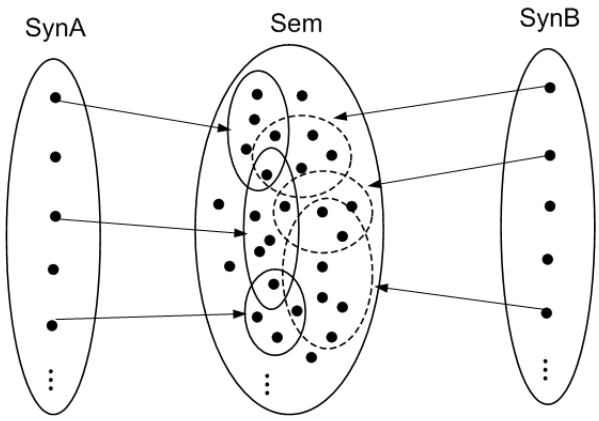


Figure 2.3: A model described using two different notations. The models in *SynA* denote solid-line blobs, and the ones in *SynB* dashed-line ones. The relation between the two is set intersection.

then used in practice [Par98]. Software engineering, however, is a relatively new discipline, only forty years have passed since its official foundation. Unfortunately, our societies demand software technologies, and practitioners cannot wait until mathematical methods become mature enough to be used to solve engineering tasks in practical ways. The urgent need to solve today's problems without mature mathematical solutions has driven the development of semi-formal methods and their associated notations. They emerged during the software crisis of the 1960s from a need to have some sort of models of software, in the same way that the whole software enterprise was crying out for engineering.

Semi-formal methods (SFMs) consist of a development method and a collection of notations for modelling software systems. Their notations are usually diagrammatic and do not have a formal semantics, which reflects their pragmatic nature. They were developed to support the programming activity, emerging from a need to abstract away from the details of code and to visualise the overall system structure and behaviour.

Their semantics is not only informal, it is also multiple. Semi-formal notations have many interpretations: they are ambiguous.[2] The same happens with words of our natural languages, and even with numerals. For example, 11 may mean the number *eleven* if we are in the context of a decimal system (the usual case), or *three* in the context of a binary one. Semi-formal notations also have these different contexts. They are applied in a wide variety of application-domains, and at different levels of abstraction. So, the choice of some semantics becomes a matter of convenience; engineers use one or another depending on the most immediate problem. The problem is that the ambiguity is not easily resolved.

Each semi-formal notation targets one or more *aspects* (or *views*) of a system. Aspects fall into two categories: *static* and *dynamic*. A set of descriptions produced using the different notations makes a system model.

SFMs have been driven by the underlying paradigm of programming. There are two main branches: *structured methods* and *object-oriented (OO) methods*. The following surveys these two branches. A more detailed survey can be found in [Wie98].

[2] In terms of figure 2.2, a semi-formal description may denote more than one meaning-set, each corresponding to one possible interpretation.

2.2.1 Structured Methods

Structured methods emerged in the late 1960s to support structured programming [DDH72]. Initially, they were driven by "top-down" functional decomposition, the approach advocated by structured programming, where the overall system processing is decomposed into smaller processing units. The *data-flow diagram* was designed to reflect this [Dem79]. It describes the flow of data into an information system from a global point of view, and then the global processing of data by the system and its decomposition into smaller units. Later, structured methods started to explore the idea of conceptual models of the reality with which the system is concerned, and models of data; the *entity-relationship diagram* [Che76], still used today, was designed for this purpose. The method SSADM [SSA90, GS95] embodies most of the ideas that evolved in this area since the late 1960s.

2.2.2 Object-Oriented Methods

OO methods emerged in the 1980s to support OO programming. Although based on a different paradigm, they borrowed many ideas from SMs. The first approaches focused on low-level design [Boo81, Boo86], only later they started to explore modelling of requirements [SM88] and higher-level design [Bal88]. Since then, many other methods have emerged. The whole evolution is marked by the Unified Modelling Language (UML), which entered the scene in the late 1990s as an attempt to unify the notations that were being used until then, and to separate notation from method. The pre-UML period is marked by methods such as, the object modelling technique (OMT) [RBP$^+$91], Coad-Yourdon [CY91], Shlaer-Mellor [SM92], object-oriented software engineering (OOSE) [JCJÖ93], Booch [Boo94], Fusion [CAB$^+$94], BON [WN95] and Syntropy [CD94]. The post-UML period is marked by methods that use UML, such as the Rational Unified Process [JBR99, Kru00], and Larman [Lar98]. An outsider in this group, the statecharts notation of David Harel [Har87], now also embodied in the UML, was designed to model reactive systems; it is based on mathematical ideas (hi-graphs and automata) and provides a notation that is rigorous but not entirely formal (it also has many semantics).

UML is discussed in detail below.

2.2.3 UML

UML [RJB99, BRJ99, OMG03, SP00, FS00] is a unification of semi-formal modelling notations. Booch and Rumbaugh started this effort by unifying the concepts of the methods they had designed, Booch and OMT. Later, the concepts of OOSE were incorporated, when Jacobson joined the effort. UML was adopted as an Object Management Group (OMG) standard in 1997. Currently, it is the de facto standard notation for modelling OO systems [Kob99, EFLR98]. There have been many versions of the UML, its current version is 2.0 [OMG03].

UML is a set of notations for modelling software systems. Most notations are essentially graphical, but there is also a textual notation (OCL, below). Each graphical notation targets one particular aspect of software systems. For example, UML class diagrams are good at describing the structure of data from a global viewpoint, and statecharts are good at describing the internal behaviour of objects of single classes. Because UML incorporates many languages, each with its many semantic interpretations, many authors define UML as a *family of modelling languages* [DSB99, Coo00, CEK01].

Diagrams explain the popularity of SFMs. They are intuitive, requiring minimal computer science knowledge in order to be understood or at least to have an idea of what they are trying to express. In general, they are good at giving overview of things, but when we need to go down to the details, we often find that not everything can be expressed diagrammatically, at least not with the diagrams provided with the UML. That is why the UML includes the Object Constraint Language (OCL) [WK99, KWC98, OMG06].

OCL, UML's constraint language, is strongly influenced by the Z formal modelling notation (next section) [KWC98]. It was designed to annotate diagrams with constraints, and so OCL expressions must always be written in the context of some diagram. OCL is used to express invariants, pre-conditions and post-conditions of operations, and state transition guards. Like Z, it is based on set theory. In pure UML, a model is made of diagrams and OCL constraints.

UML and OCL are defined through meta-modelling. First, the general notion of meta-modelling is explained, and then we take a closer look at the way UML and OCL are defined.

Meta-Modelling

Meta-modelling, a concept related to *ontology*[3] [Mos95, Wan96], tries to describe the relations between modelling concepts. A model is an abstraction of phenomena of some domain (real-world or other), a meta-model is yet another abstraction highlighting properties of the model itself [SAJ+02]. Usually, meta-models describe modelling concepts and their relations — in an object-oriented context, this involves concepts such as class and association.

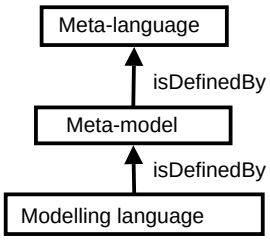


Figure 2.4: A modelling language is defined by a meta-model, which, in turn, is defined using a meta-language.

Several layers of meta-definitions may be involved in the definition of a model, as a meta-model may itself be defined by another meta-model. Ultimately, however, the last layer must be defined in some meta-language (figure 2.4).

Meta-modelling approaches aim at *flexibility* and *understandability*. The idea is to allow modelling languages to be tailored to specific purposes by defining and modifying the underlying meta-model. To achieve this, meta-models must be easy to understand and modify. Related to meta-modelling is the term *method engineering* [Bri96], which can be defined as "the engineering discipline to design, construct, and adapt methods, techniques and tools for the development of information systems" [Bri96].

[3]Philosophical ontology has roots in the metaphysics of Aristotle; it is the science of what is, of the kinds and structures of objects, properties, events, processes and relations in every area of reality [SW01]; it can be seen as a particular system of categories accounting for a certain vision of the world [Gua98]. The concept was brought by McCarthy to computer science in the late 1970s, where it is defined as a specific artifact designed with the purpose of expressing the intended meaning of a shared vocabulary [Gua98].

UML and OCL definitions

UML and OCL are both defined using meta-models in their OMG definitions. The OMG developed a language to define meta-models, the Meta-Object Facility (MOF), that is used in the current UML definition [OMG03]. Essentially, UML's metamodel is defined in terms of three views: abstract syntax, well-formedness rules, and semantics. Abstract syntax is expressed using MOF; well-formedness rules are expressed in OCL; and semantics is expressed using a combination of MOF and natural language.

2.2.4 Discussion

SFMs have evolved from about 30 years of software engineering experience. They have not been immune to the influence from formal methods: Syntropy, Fusion and OCL being clear examples of this influence.

The strengths and weaknesses SFMs in general and the UML in particular are discussed below.

Strengths

Popular wisdom says that "a picture is worth a 1000 words". Semi-formal notations have a visual (or diagrammatic) nature, and it is this, together with their pragmatic approach to development, that explains their popularity. In summary, the strengths of semi-formal techniques are as follows:

- *Intuitive and widely known notations.* Semi-formal notations are graphical, making them appealing, intuitive, and easily assimilable. They are good at describing particular aspects of systems, abstracting away from detail, and giving a good overall picture of what is being described. Sometimes they do not require a great deal of expertise in order to be understood. Many semi-formal graphical notations are defacto idioms among engineers being widely taught, which makes them a good vehicle of communication.

- *Connection with real world.* At the abstract level, SFMs emphasise a connection with the real world. This is actually emphasised by diagrams such as data-flow diagrams, which describe the flow of data from the environment into the system, and entity relationship and class diagrams, used in conceptual modelling to describe the main concepts of a software system and their relationships providing a common vocabulary coming from the application domain.

- *Methodological support, emphasising problem decomposition.* SFMs provide more than just a notation, providing step-by-step guidance on how to approach problems. They encourage problem decomposition, which helps to reduce complexity. In structured methods, decomposition is based on processing of data. In OO methods, it based on concepts and the structure of data.

Weaknesses of semi-formal methods

Semantics is the biggest drawback of semi-formal techniques. But there are other problems, such as SFM's approach to views, and tailoring (or customisation).

Semantics. Semi-formal notations have several problems related to semantics, namely:

- *Notations have unsound definitions.* Either they are defined informally and vaguely using natural language, or they are defined through meta-modelling using some meta-language (usually some diagrammatic notation) that is not precisely defined.
- *Certain core concepts have a vague semantics.* Some concepts, such as associations [Ste02], inheritance and aggregation [HSB99] in UML, do not have a clear semantics.
- *Semi-formal notations do not have a fixed interpretation.* Developers tailor the interpretation of diagrams to the problem at hand informally, tacitly and sometimes unconsciously. This constitutes a source of confusion and ambiguity; it may happen that different people take different interpretations of some modelling aspect.

These problems bring the following consequences:

- *Ambiguity and inconsistency in the resulting models.* It is likely that people disagree about the meaning of the models, that single diagrams are inconsistent or, even more likely, that different diagrams are not consistent with each other. Since there is not a clear definition of the semantics, it is not easy to resolve these issues.
- *Resulting models are imprecise.* Again a consequence of the lack of sound semantics, but it is also a problem with the limitations imposed by diagrams. Sometimes one wants to be a bit more detailed and precise, and there are no means for doing so. (UML tries to overcome this with OCL, but OCL has many problems, see below.)
- *Lack of means for mechnical analysis.* Since the notations do not have a formal semantics, resulting models cannot be mechanically analysed for consistency and satisfaction of desired properties.
- *Understanding more apparent than real.* The lack of precision, well-defined semantics, and means for mechanical analysis can make the understanding more apparent than real. All is too easy and superficial, and the specifier is never confronted with the relevant issues. In Parnas' words [Par77]: "obvious ambiguity is relatively harmless, it is subtle ambiguities that lead to expensive misunderstandings, because all parties think that they have understood."

Views. Theoretically speaking, modelling different aspects of software systems from different views is convenient because it gives a separation of concerns. But in practice it is problematic because the relation between the different views is not clear. This brings the following consequences:

- *Not clear how different views articulate with each other.* The relation between different diagrams is not clear and this can only be solved with proper semantic definitions designed to articulate different diagrams. Even in simple and apparently straightforward combinations such problems occur. For example, in a combination of class diagrams and statecharts: what is the exact dependency between the two?
- *Consistency across partial models cannot be checked.* Again, this is because the notations are not related at the level of the semantics. Consequently, it is possible that different partial descriptions that are mutually contradictory coexist and remain unnoticed until the system is implemented.

Tailoring. Semi-formal graphical notations may have many interpretations, which is not necessarily bad. After all, model descriptions may also have many meanings. The real problem is that there is no way to say what is the right interpretation to be followed in some development, or to say that two modelling concepts are almost the same, but they have a slight semantic variation. Essentially, it must be possible to make explicit what is often tacit. If this is not done, then different people take different interpretations of a model and there is the danger of misinterpretations that cost much more if only found later.

Moreover, step-by-step methods of SFMs are rigid. They are of little use in face of unexpected situations, which is often the case, and they are not designed to evolve with experience.

Weaknesses of UML

There are several problems with both UML and OCL. We start with the latter.

OCL. OCL converges with Z in its approach to model software systems, but it diverges on the complexity of its semantics. OCL's founders intended the language to be formal, but its official definition [OMG06] is semi-formal. There is a paper-and-pencil mathematical semantics of OCL [OMG03], but researchers have reported difficulties in a machine-checked formalisation of the language [BDW06, RG01, HHB01]. Hennicker et al say [HHB01]: "As soon as the technical details of UML/OCL are considered seriously, a number of semantic questions arise which cannot be answered properly by referring to the existing documentation."

Despite a more recent rewriting of OCL [OMG06, OMG03], many problems identified with OCL's initial definition still stand. OCL was designed to enable model execution. Executability, however, sacrifices abstraction [HJ89]. Indeed, in many ways OCL is more like an OO programming language then an abstract modelling language such as Z or Alloy. In OCL, predicates may be defined in terms of class operations, which may be recursive. And because an operation may not terminate, OCL is based on a three-valued logic (third value is *undefined*) [HHB01, KFdB$^+$04], as opposed to the simpler two-valued logics of similar languages (Z, Alloy and B). This complicates OCL's semantics: many rules of the predicate calculus need to be rewritten to account for the third logical value [HHB01]. An OCL constraint or query in a system configuration (a snapshot) should always yield a result, in the case of non-termination it should yield *undefined*, but this is a well-known undecidable problem [HHB01, KFdB$^+$04].

Vaziri and Jackson [VJ00] criticise OCL's unfitness for abstract modelling. Constraints based on operations gives an implementation flavour to the language, and brings precision and consistency concerns; OCL's type system is also too close to that of an OO programming language. OCL is also criticised for its verbose expressions, which are difficult to read, and for the unclear relation between an OCL expression and the accompanying UML diagram. Jackson [Jac06] notes that a UML association is interpreted in OCL not as a simple and more abstract relation, but as a function to sets for each association role, which gives a strong directionality to associations not allowing them to be traversed backwards.

There have been sucessful formalisations of OCL, but they are not fully conformant with its official definition. Moreover, they are applicable only to a much smaller subset of the UML. The approach of Richters and Gogolla [RG98, Ric01] is such an example. The authors have engineered a new language called USE [Ric01] that extends OCL with declarations of classes and associations. The semantics of USE formally defines a class model. The USE tool [USE]

allows mechanical validation of UML-based models (USE models) based on snapshots (this is discussed further in the snapshots chapter).

UML and OCL's multi-view architecture may suggest an independent definition of their semantics. But as USE and other efforts show (e.g. [KFdB+04]) a formal definition of OCL requires a a formal definition of a core of the UML (USE did it for a subset of UML class diagrams). Unless the formalised core is so general that applies to all UML variants, there is a commitment to some semantic interpretation of the UML. So that if another semantic interpration is required, it is likely that the OCL formalisation needs to be redefined. It seems difficult to balance UML's generality (as a family of modelling languages) and the need for a formal definition of a single OCL.

UML. The definition of UML has been criticised for its complexity. Solberg et al [SFR05] state:

> The current complexity of UML 2.0 can make understanding, using, extending, and evolving the metamodel difficult. [...] It will be extremely difficult to evolve the UML 2.0 metamodel to reflect changes in the UML standard using only manual techniques. How can one be sure that required changes are incorporated consistently across the metamodel? How can one determine the impact that a change will have on other metamodel elements? In particular, how can one ensure that the changes do not result in a metamodel that defines inconsistent or nonsensical language constructs?

In fact, the UML 2.0 definition is so complex that it is difficult to criticise it. In Hoare's words [Hoa81], "there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."

The UML definition is far from sound. In the previous version, UML was defined in terms of a subset of itself in the meta-model, but this subset was not formally defined elsewhere. This version defines the UML in terms of MOF (which is also based on a subset of UML class diagrams), but the same problem remains because MOF is not formally defined. Moreover, the language definition is ambiguous because of the use of natural language and OCL, which has many semantics problems (see [Ste02] for specific examples of ambiguities in the previous versions of UML).

2.3 Formal methods

The previous section mentions some of the problems that are involved in modelling without a mathematical basis: ambiguity, inconsistency, lack of precision, no means to perform mechanical analysis and understanding more apparent than real. It is to avoid these problems that mature engineering disciplines base their models on mathematics.

Software engineering is no exception and researchers have been working on mathematically-based techniques to support software development for about 40 years now. Initially, the use of these techniques in practice seemed hopeless, but recently there has been a more promising picture [CW96]. This is precisely what this thesis tries to show: practical model development with mathematically-based techniques is (a) possible, (b) practical and (c) worth doing.

In software engineering, mathematically-based approaches are called *formal*. Formal Methods (FMs) are based on sound mathematical theories, providing a framework to construct and analyse models in a systematic and sound manner [Rus95, Win90, CW96]. Formal notations have a mathematically-defined syntax and semantics, and usually a well-defined logical inference system. They allow the precise and unambiguous description of a system and its desired properties, and *formal analysis* of a specification. Certain notations also support the notion of formal *refinement*.

This section discusses and surveys FMs. It starts by discussing the research history that led to modelling languages in software engineering. Next, it discusses techniques for formal modelling, model refinement and model analysis. Then, it surveys formal modelling languages. Finally, it discusses strengths and weaknesses of formal methods.

2.3.1 Historical Notes

The quest for mathematical techniques to support software engineering started with research in the definition of programming languages [Luc]. The first technique to emerge is now so widespread that we forget that it is mathematically-based: syntactic definitions of formal languages based on generative grammars, realised in the language BNF (Backus Naur Form) [Bac60]. This achieved an effective separation of concerns, between syntax and semantics, which was not so clear at the time. Another important contribution came from another separation of concerns: between *concrete* and *abstract* syntax introduced by McCarthy [McC62]. This separates details of a notation (concrete) from its essential structure (abstract) and facilitates the definition of a mathematical semantics.

In [McC62], McCarthy also paves the way for research in formal semantics: "Ideally we would like a mathematical theory in which every true statement about procedures would have a proof, and preferably a proof that is easy to find, not too long, and easy to check. [...] Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program."

By the early 1960s, there were mathematical techniques to define syntax, and high-level programming languages, which were realised in the Algol-60 programming language. Mathematical (or formal) semantics would follow, motivated by a need to represent semantics independently of the language implementation, a need to understand the concepts of programming languages, and proof of properties of both programming languages and programs.

In the late 1960s, Scott and Strachey laid the theory of *denotational semantics*, which describes a programming language in terms of abstract mathematical objects [SS71]. In this approach, semantics is defined through mappings (meaning or semantic functions) from syntactic constructs to their mathematical meaning. In [SS71], the authors observe: "the interpretation of the language depends on the state of the system, [...] computer oriented languages differ from their mathematical counterparts by virtue of their *dynamic* character. An expression does not generally possess one uniquely determined value of the expected sort, but rather the value depends upon the state of the system [...]." So, they proposed a mathematical abstraction of a state transformation that sees system dynamics as transformation of states. This abstraction, a function from a state space S into itself ($S \rightarrow S$), describes the meaning of a command in a program as taking the system from one state into another.

At about the same time, systems of deductive logic were the inspiration to approaches to prove properties of programs. A logical system is made of a set of axioms and inference rules, giving a mechanism that allows certain formulas to be deduced from others. The first

approach to emerge was that of Naur [Nau66] who proposed an approach that is rigorous but not formal. More formal and also more influential, Floyd's work [Flo67] proposes a method to annotate flowcharts of programs with mathematical logic formulas with the aim of formal deductive reasoning. Floyd's novel idea was to see language semantics as a logic, where axioms and inferences rules are associated with the commands of the language; proof was a formal argument that the commands would establish logical formulas stated as annotations. In [Hoa69], Hoare defines *axiomatic semantics* by extending (to program text) and refining Floyd's approach. In that paper, Hoare says that "The most important property of a program is whether it accomplishes the intentions of its users." In his method, the intended function of a program is specified as logical assertions about the values of relevant variables at the end and intermediate points of the execution of a program. Then, proof would be based on these assertions and the axioms and inference rules of the language. Provided that the programming language implementation satisfies the axioms and rules of inference, then "all predictions based on these proofs will be fulfilled." The rules of the language are expressed using a special notation, $P\ \{Q\}\ R$, where P is the *precondition*, Q is the language command, and R is the expected result or *postcondition*; $P\ \{Q\}\ R$ can be interpreted as: If the assertion P is true before the execution of the command Q, then the assertion R will be true on its completion.

Hoare's axiomatic method checks the accuracy of programs within a mathematical framework. A program is, however, a solution to some problem, so analysis is done at the level of the solution, not the problem. So, mathematical approaches to describe and reason about software problems were still missing, but it was close. Modelling languages would emerge from the work on denotational and axiomatic semantics motivated by a need for further abstraction.

Several works are relevant in the leap to modelling languages. Data abstraction [Dij72, Hoa72a, DH72], stepwise refinement [Wir71, Hoa72b] and Dijkstra's [Dij75] calculus for the derivation of concrete programs from abstract ones. This made evident the value of abstraction from the level of concrete programs for conceptual and practical reasons, because both models and proofs of correctness are simplified in this way. The work on denotational semantics would also be influential; those techniques used to describe the semantics of programming languages would also be used to describe software.[4] By the mid 1970s, the first formal modelling languages were starting to emerge. The seminal paper on the formal language Z [ASM80] says:

> The concept of specification language is now widely spread; formalising a problem is well recognised as a necessary step preceding any programming. The formalisation technique, however, is still the purpose for intensive research [...]

2.3.2 Techniques of formal modelling

The most mature techniques describe software in terms of concepts from discrete mathematics, such as, sets, relations, graphs, partial orders and finite-state machines [Rus95]. *Calculation* is based on the methods of formal logic. The goal of *formal methods* is to use these mathematical concepts to supplement or replace the English prose, pseudocode, and various forms of diagrams that are traditionally used in documenting requirements, specifications and designs for software systems [Rus95]. The benefits of doing so are discussed later.

[4] The language VDM was designed with both aims in mind.

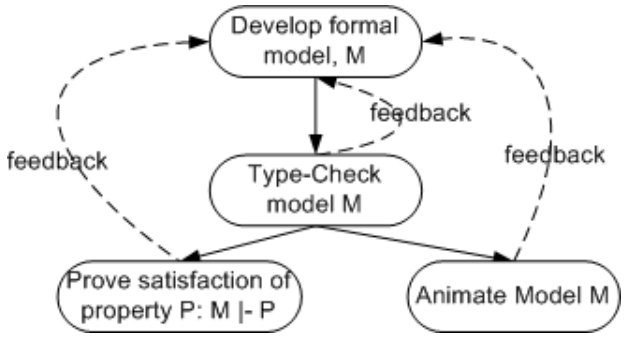


Figure 2.5: The process of developing formal models. A model is written, type-checked, and then checked for the satisfaction of properties (through proof) and animated.

Figure 2.5 depicts the iterative nature of formal model development. Models are written in some language in terms of discrete mathematical structures. Then, the model is type-checked (see below) in order to know if it makes sense. Then, from a model, we may calculate (or deduce) its properties by using the axioms and inference rules of the logic of the language. So, given a model M, we may check the satisfaction of the property P, by proving that, $M \vdash P$. This is done through theorem proving, model-checking or Boolean satisfiability (see below). Alternatively, the model can be animated. The feedback coming from proof and animation is likely to result in changes to the formal model.

Each modelling approach makes a choice of primitives or abstractions for building models of software systems. These should target two fundamental aspects of a software system: control and data [Rus95]. Control is concerned with the selection, timing, and sequencing of the operations performed in a software system. Data is concerned with how information is represented and manipulated by the software. Two common primitives in formal modelling are: the abstract data type (ADT), an abstraction of data, and the *process*, an abstraction to describe the interaction between a computing system and its environment.

Essentially, there are two techniques of description, and modelling languages favour one or another. The *model-oriented* approach models the object of interest directly, using mathematical stuctures such sets, tuples, sequences and relations. The *property-oriented* approach models the object indirectly by stating axioms that the object must satisfy.

The next section discusses formal refinement.

2.3.3 Formal Refinement

We have already seen that models need to be refined or transformed (p. 13). Abstract models need to be refined into design models and design models into programs. Formal refinement ensures that these transformations are correct; that is: the refined (or concrete) model conforms with the behaviour of the abstract. The correctness of a refinement is demonstrated through mathematical proof.

Refinement is sometimes too restrictive and it is because of this that formal refinement is difficult to apply in practice. The challenge is to come up with strategies that alleviate some of the refinement restrictions while remaining true to the principles of formal refinement. This thesis faces precisely this problem in chapter 5, where the problem is discussed in detail.

Despite its practical limitations, formal refinement is increasingly being used as a modelling technique to tackle the complexity of a problem. The abstract model expresses particular system properties, but omits others. The refinement then expresses the properties of the abstract model and others. This approach can be used to prove certain properties at the level of the simpler abstract model; provided that these properties are preserved by the refinement, the proof of satisfaction of those properties at the level of the refinement amounts to a refinement correctness proof. This was what was done in the Mondex development [SCW00] (see above) to prove the satisfaction of desired properties of an electronic Purse. This refinement approach can also be used for building programs and models in an incremental fashion, where the development starts with a very abstract model and incrementally adds more properties to refinements. This was used to model complex network standards [ACM02] and build complex pointer programs [Abr03]. This refinement technique has also been documented as a pattern for formal development (pattern *Do a refinement* [SPT03b]).

2.3.4 Model Analysis

Model analysis is the extraction of information from a model description. In software engineering, models are analysed for checking that the model is consistent, satisfaction of desired properties, and validation and verification of models.

The following discusses some common mechanical analysis techniques: *type-checking*, *theorem proving*, *model checking* and *boolean satisfiability*.

Type-checking

Most modelling languages support the notion of *type*, which denotes a set of values. Types allow the detection of nonsensical language expressions, such as multiplying a boolean by an integer. Modelling languages based on certain kinds of mathematical logic have to use types to keep the logic consistent and avoid paradoxes such as the one of Russell. The advantage of a language with types is that nonsensical expressions can be detected by using a computer program (*typechecker*). The typing rules of the language should enable type-checking algorithms that are decidable and not computationally complex. Type-checking is an effective mechanism at capturing errors: in programs it captures errors, without requiring the program to be run, and in models without requiring proof.

Animation

Animation (or simulation) is a technique to test models [JR00, HJ89, Rus95]. In programming, programmers generate sample executions in order to observe the behaviour of their programs. Model animation is similar, modellers generate sample states and transitions in order to observe the implications of their models. Some formal modelling languages lend themselves to executability; that is, they allow test cases to be run directly against the model. Executability may not be necessarily a good thing, being at odds with important properties of models such as abstractness and non-prescriptiveness [HJ89], but animation does not require executability.

Model Checking

This technique checks automatically that some desired property holds in a *finite* model of a system[CW96]. Model-checking applies to finite models of software systems only, but there

are techniques to allow its use for infinite (or very large) models (e.g. abstract interpretation). Model-checking explores all possible behaviours of the model and works by refutation; that is, to prove that a model satisfies some property P, what the model checker does is to try to find (exhaustively, analysing every single case) a counter-example (an instance of $\neg P$); if none is found then the model satisfies P.

The main advantage of model-checking is that it is *decidable*. That is, it is possible to automate it with a computer program (a *model-checker*). However it is a problem of high computational complexity, so the program may not be practical for larger models. In fact, the main challenge in model checking is in dealing with the *state explosion problem*, which occurs in complex systems with an enormous number of global states.

Theorem Proving

Many modelling languages have a well-defined logical inference system, which is used to prove properties of a model. *Theorem proving* is the process of finding a proof of a property from the specification of the system.

Theorem proving can be supported by tools. Automatic proving, however, is, in general, an undecidable problem for any reasonably expressive logic (e.g. first order logic is undecidable). This means that a fully automated theorem prover is, in general, impossible, but there are tools (*theorem provers*) that are able to automate certain theorem-proving tasks, and allow the intervention of the user to solve more complex cases.

Boolean Satisfiability (SAT)

Analysis based on boolean (or propositional) satisfiability (SAT) consists in finding a solution for a propositional formula. That is, an assignment of boolean values to propositional variables, such that the overall formula is satisfied (it is *true*). For example, a solution for the propositional formula,

$$(p \wedge q) \vee r$$

is: $p = \mathit{true}, q = \mathit{true}, r = \mathit{false}$. The SAT problem is decidable, but it is an NP complete problem [Coo71]. This means that an algorithm that solves the SAT problem fast enough for practical purposes (in polynomial time) has not yet been found, nor has a proof that such an algorithm does not exist. The problem is hard, and its complexity increases with the size of the problem. However, advances in SAT technology over the last decade enable the resolution of SAT instances with thousands of boolean variables and millions of clauses[5]. This is why the technique is increasingly being used to analyse models.

Usually, the modelling language is more expressive than propositional logic. The way it works is by translating formulas of the modelling language into propositional logic for analysis.

2.3.5 Modelling Languages

Formal modelling languages embrace a variety of approaches that differ considerably in techniques, underlying mathematical theories, goals, claims and philosophy [Rus95]. They are designed with different purposes.

[5]Part of the reason is that SAT in general is NP complete, but most instances are not so hard to solve.

One important characterisation of formal languages is whether they are *model-oriented* or *property-oriented* (see above). Each language favours one approach or the other, but usually this constitutes a style of modelling, and several languages allow both styles.

Modelling languages are also characterised by their focus on either description or analysis [JR00]. Some languages focus primarily on the form of the model, expressiveness and modelling abstractions, whereas others (typified by the developers of model checking) focus on automated analysis and neglect the way the model is expressed. Of course, a good language design should make a good combination of both features. A language whose semantics is so complex that it precludes model analysis, has its practical value compromised, and a language that neglects description is likely to be rejected, because users like to express ideas in simple, intelligible and elegant ways.

Refinement is another important characteristic. Some languages emphasise refinement of models into code, others emphasise refinement at the level of designs, and there are languages that ignore the refinement issue.

The following surveys various modelling languages based on their appropriateness to yield state-based descriptions, event-based descriptions and those that combine both features. More detailed comparisons of formal modelling languages can be found in [Win90, Jac06].

State-based

State-based languages capture rich-structures of software systems abstractly and succinctly. They are based on the idea of data abstractions and most of them favour a style of specification based on abstract data types (ADTs). The following surveys those languages that favour a model-oriented style of specification and those that favour a property-oriented one.

Model-Oriented. State-based languages that favour a model-oriented style of specification include the languages VDM, Z, B, Alloy and Object-Z. They all follow a similar approach to model software systems, based on the concepts of set theory and predicate logic.

VDM (Vienna Development Method) [Jon91] started in the early 1970s. Initially, the language was intended to be used to model the semantics of programming languages, but it was later adapted to model software systems in general. The latest version of the language, VDM-SL, is documented in the ISO standard [ISO96]. Jones [Jon99] gives a brief history of the VDM development. VDM favours a style of specification based on abstract data types (ADTs), providing special syntax to support this style of specification (record types).

Z was developed at the university of Oxford in the 1980s. It is based on typed set-theory, and first order predicate calculus, enriched with a structuring mechanism based on *schemas*. Z is flexible and adaptable to suit different styles of specification (this thesis proposes an OO style for Z), but the standard style is based on ADTs specified using the schema calculus. The seminal paper on Z is [ASM80]. The latest version of Z is the ISO standard [ISO02b]; the version documented in [Spi92] is still popular. Popular Z textbooks include [WD96, BSC94]. Z has a denotational semantics based on *untyped* first-order logic.

B is a language and a method design by Jean Raymond Abrial, one of the early contributors to Z. It includes a notation based on abstract machines (ADTs), and a method to refine abstract models into code in a stepwise-manner. The reference version of the language is documented in [Abr96]; a popular textbook is [Sch01]. Z and B converge on their approach to model software systems, which reflects Abrial's influence on both of them, but it diverges on the level of abstraction. Z is more abstract, focusing on modelling software systems, and B is

more like an abstract programming language with a very strong emphasis on refining models into code. Both languages also differ on their semantics, Z has a denotational semantics, and B has a semantics based on weakest-preconditions.

Alloy is a modelling language designed by Daniel Jackson at the Massachusetts Institute of Technology, which is strongly influenced by Z. The language is defined in [Jac04], and a recent textbook is [Jac06]. Alloy simplifies the semantics of Z, at the cost of being less expressive, with the aim of model analysis based on SAT solvers. Alloy expresses all structures as relations, including sets and scalars, and disallows high-order relations.

Object-Z is an extension to Z to facilitate the structuring of a model in an OO style. The reference language definition is [Smi00], other textbooks include [DR00]. It provides the class schema as a structuring mechanism (a collection of an internal state schema and operation schemas) and support for OO notions such as object, inheritance, and polymorphism. Object-Z is the most successful OO extension to Z, among the ones that emerged in early 1990s (see [SBC92] for a survey).

Property-Oriented. Property-oriented languages that are state-based include Larch [GHW85, GH93], Clear [BG80], OBJ [GT79], Anna [Lv85], and Act-One [EFH83]. They are not considered any further in this thesis.

Event-based

Behavioural languages emphasise interaction among processes and are suited for concurrent and distributed systems. Model oriented behavioural languages include the *process algebras* CSP [Hoa85, Ros98, Sch00], CCS [Mil80, Mil89], and the π-calculus [Mil99] (a language with its roots in CCS and focused on *mobility*). Also model-oriented, is the visual formalisms *Petri-Nets* [Pet77] to specify the behaviour of concurrent systems.

Property-oriented behavioural languages include those based on temporal logic [Pnu81, Lam89, MP91].

Combinations of state-based and event-based

Approaches based on combinations aim at gaining further expressibility by linking different languages. The following approaches link languages through a unified semantics:

- LOTOS (Language of Temporal Orderings of Specifications) [BB87, EM85, Tur93, ISO02a], a protocol specification language based on a combination of ActOne and CCS.

- RAISE (Rigorous Approach to Software Engineering) [Gro92], a language based on VDM, but enriched with features of CSP and CCS, for the expression of concurrent properties, and features of property-oriented languages, such as description through the use of axioms and modularity; the integration of different features was done with a single semantics for the whole language [Mil93].

- Circus [WC02, SWC02, CSW02], a concurrent specification language, with a refinement calculus, resulting from the combination of Z and CSP; the combination is done through the unified theory of programming [HH98]; Circus adds to CSP the expressiveness of Z for describing complex data structures.

Other approaches make a combined use of modelling languages, but without changes to the semantics. A notable example is *CSP* ‖ *B* [ST02, TSB03, STE05].

2.3.6 Discussion

FMs allow models to be expressed precisely and concisely, and with a level of abstraction that is appropriate to the task. They also enable mechanical analysis of models and some provide formal support to stepwise-refinement

There have been many success stories of formal methods. [CGR93, CGR95] surveys the application of formal methods in industry. Praxis High Integrity Systems, a company specialised in formal based development, has many success stories of its own [Hal96, KHCP00]. The Mondex development [SC00, SCW00] is a landmark in specification and refinement with Z, a case where industrial practice drove the theory, which is still used as case study for further research [BPJS05].

Despite these successes, formal methods have not been embraced by industry. Their use still remains mainly confined to critical systems, where mathematics is essential. Moreover, their use requires expertise.

The following compares formal methods, and discusses their strengths and weaknesses.

Comparison

Model-oriented methods (including the visual ones) are considered to be easier to understand than property-oriented ones. However, they are criticised for being *over-prescriptive* [Rus95], as they suggest how something is to be implemented rather than just the properties it is required to have.

Property-oriented languages on the other hand are less prescriptive than model-oriented ones, as it is possible to constrain certain relationships or values without having to define them exactly [Rus95]. On the other hand, it is very easy to write conflicting constraints that cause the specification to become inconsistent; inconsistent specifications are unimplementable and are dangerous as they can be used to prove anything [Rus95]. The use of definitions via axioms, in contrast to explicit definitions, may lead to under-specification. The definition of axioms is also error-prone, as it is easy to miss axioms when defining a complex data-type.

Unifying formalisms, such as RAISE, may result in languages which are large, and lose some of the conceptual integrity and elegance of the original formalisms. It also makes harder the task of defining a semantics, proof theory and refinement calculus for the language. Milne reports some of those problems in the definition of RAISE [Mil93].

Visual languages, due to their graphical nature, are intuitive. However, they have a very limited scope in terms of what they can express, when compared to the textual languages.

Benefits of formal modelling techniques

The following are recognised benefits that accrue from the use of formal modelling techniques.

- *Formal modelling aids understanding.* Formal modelling helps to gain a deep understanding of the system and its domain. It encourages the specifier to be abstract, yet rigorous and precise, forcing the modeller to ask all sorts of questions.

- *Formal modelling helps to clarify requirements.* Formal modelling clarifies the customer's vague ideas, revealing ambiguities, inconsistencies and incompleteness in the requirements.

- *Formal models constitute precise and unambiguous descriptions, which aids communication.* Formal languages have a well-defined syntax and semantics, and are based on discrete mathematics. They yield precise and unambiguous descriptions of software systems, which may lead to a better communication among those working on the system.

- *Formal models enable rigorous analysis, aiding verification and validation.* The analysis of formal models can be used to support verification and validation. In verification, a formal model can be checked for the satisfaction of desired properties, and that a refined design or implementation satisfies its specification. In validation, a requirements model can be checked against its requirements either through animation or proof, and for black-box system testing by generating test cases from the model.

- *Formal modelling aids system evolution.* Formal specifications help to assess the impact of changes upon a system implementation. If the specification is updated as a result of a system change, the actual change and its impact upon the system become clearer. This helps to maintain the conceptual integrity of the system.

Problems of formal methods

Some recognised shortcomings of formal software engineering methods are given below.

- *Formal approaches require expertise.* The use of formal approaches requires a great deal of expertise. This can be observed by the success cases of formal techniques, most of them involve experts from academia that have been involved in the development of the approach. Besides the underlying mathematics, users need to understand the idioms of modelling, proof, and how to use the toolset that is available for the technique.

- *Lack of full description coverage.* Most formal methods are suited to describe particular aspects of systems, but usually not all aspects. Statecharts describe the reactive behaviour of components, languages like Z are good at describing system structure, and languages like CSP are good at describing interaction between components. The problem is when all these aspects need to be modelled: what is the best way to do it? How to ensure that the different descriptions written in different languages are consistent with each other?

- *Lack of methodological support.* Formal methods provide a notation to write models, and approaches to analyse them. Languages like B give some methodological support by allowing stepwise refinement from abstract models into code. However, software engineering practice requires further support: guidelines, approaches to modelling, and patterns.

- *Not well integrated with existing system development practice.* The use of formal methods requires an extensive period at the start of the project during which attention is focused exclusively upon formal system specification and no code gets written. This is at odds with development practice, which commences code development sooner rather

than later, and management practice, which typically uses metrics like lines of code to track project progress [CM95].

- *Large variety of formal approaches.* The large variety of formal methods makes the choice of a particular one difficult [CM95]. In certain safety and security related circles, there is a consensus that formal methods should be used to attain high-levels of assurance, but there is much less consensus on which formal methods should be used, under which circumstances and why.

- *Lack of tool support.* Despite one of the claimed advantages of formal methods being amenability to formal analysis, most formal methods have little automated support beyond type-checking; developers are usually left the onus of performing proofs, which demand too much time and expertise for practical application [JD96].

- *High up-front costs and reduced market.* Formal methods require a high up-front cost. Practitioners need to be trained, and, since there is not much experience in using formal methods, the costs associated with their use are high. They also require an investment of time and money in specification, before any code is written. Ultimately, they result in higher product quality, fewer errors and a reduced need to modify and maintain systems once they are introduced. In the software industry, however, cost and investment expectations reflect traditional software practice rather than the right way of doing things [CM95]. For example, much revenue is generated in software maintenance. The fact that formal methods have a very restricted market (only in security and safety areas) does not help [CM95]. A wider market would justify investments in better tool support and would result in more experience in using formal methods.

2.4 Integrating semi-formal and formal methods

Although the increased rigour, precision and means of calculation that formal techniques offer seems indisputable, formal methods have not been taken up by industry. To explain this many reasons have been hypothesised, education being one of them, but Michael Jackson has a different view. In [Jac98], Michael Jackson criticises existing formal methods for being *constructive*, in the sense that their "goal is the creation of new structures for problems and their solutions", and *universal*, in the sense that they are too general and have a presumption of universal applicability. "The methods of value", claims Jackson, "are micro-methods, closely tailored to the tasks of developing particular well-understood parts of particular well-understood products." Micro-methods constitute "the most useful context for the precision and reliability that formality can offer." Jackson defends his view in the light of the practice in traditional engineering, the favourite analogy of formal methods' advocates [Jac98]:

> Formalism in traditional engineering is applied to well-understood components of well-understood characteristics. The context for applying the formalism is almost fixed, and the calculations to be made are almost standardised.

As discussed above, formal methods have their success stories, but they are not widespread. So far, they have been embraced only in domains where reliability is absolutely crucial, such as safety-critical and security-critical systems. Moreover, usually the people that are involved in these projects are not just experts, they are the designers and the most influential users of some formal method. This sort of expertise is not available just around the corner.

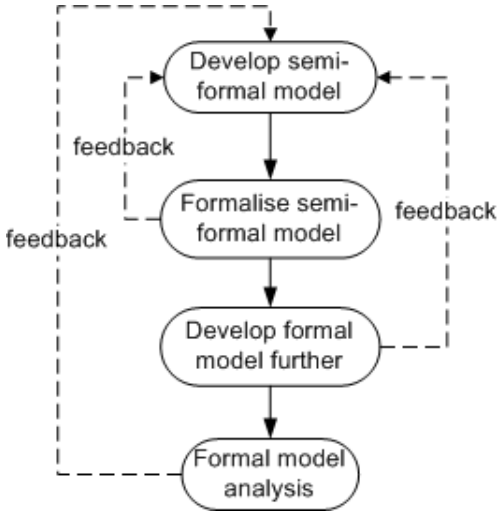


Figure 2.6: The process of developing an integrated model. The semi-formal model is written, and then formalised to produce a formal model. The formal model is then developed further and analysed.

Indeed, experience shows that formal methods are not easy to use, formal modelling and analysis requires a lot of effort and expertise, and languages like Z, for example, are just too general. It is not easy to build formal models from a blank sheet of paper. It is because of problems like this that formal methods have been criticised for not being practical.

Mainstream software engineering relies on semi-formal methods, but these are notorious for their lack of rigour. So, it is to overcome the problems of both formal and semi-formal methods that people have been trying, since the early 1990s, to use them together. The proposers of so called *integrated methods*, have been trying to integrate formal and semi-formal methods into a single framework of model-development. This is done with the goal of supplementing the semi-formal method with formal techniques in order to introduce rigour in the development, and to sweeten formal methods usage with methods and diagrams. The integration would benefit both sides:

- The semantics of the semi-formal notations would be defined precisely and unambiguously by using a formal modelling language.
- The use of a formal modelling language would bring mechanical means of analysis.
- And the formal approach would be integrated into a development method, where many developers, who are not necessarily experts in the formal approach, can participate.

The following starts by discussing approaches to the integration of semi-formal methods. Then, it surveys existing work that has been done to integrate structured and OO methods. Finally, it discusses some of the issues that are involved in this area.

2.4.1 Approaches to the integration

Formal and semi-formal methods are integrated with different goals. Some approaches aim at using formal and semi-formal methods side-by-side throughout the development. Other

approaches aim at remaining essentially semi-formal, using the formal method for semantics, consistency-checking and analysis of diagrams. Others still try to remain formal throughout the life cycle and just use diagrams for visualisation and communication.

Figure 2.6 describes a model development process with a *tightly* integrated method. The process starts with the development of the semi-formal model, which is then formalised. The results of the formalisation are likely to impact on the semi-formal model. The formal model is then developed further to include those properties that cannot be expressed diagrammatically. Finally, the model is thoroughly analysed. Different approaches may propose variations to this process, but most of them follow something similar.

But how to integrate diagrams and formal (textual) models? What exactly is the relation between the two? Most approaches go from diagrams into formal models, but there are also approaches that extract diagrams from the formal model. Going from diagrams into a formal model is described by some authors as *translation* because one language description is transformed into another. However, the word translation implies preservation of meaning, and what is being done is to formalise the vague meaning of diagrams. That is why the term *formalisation* is used here instead, but emphasising the fact that there may be many formalisations for some type of diagram, one is being defined. The term *mapping* is also used to refer to the process of going from diagrams to formal text, which emphasises the theory underlying this approach: *denotational semantics* [Ten76, Sto77]. Essentially, syntactic descriptions of diagrams are mapped into a semantic domain represented in a formal language.

There has been an evolution in the way diagrams are formalised. Initially, the formalisation was *cognitive-based*, and so, the formal model reflected the specifier's overall understanding of what the diagrammatic model expresses; it could involve some cognitive heuristics, but, essentially, the overall understanding was the main driving force. This evolved into *rule-based*, that is, based on precise rules on how to formalise the different syntactic elements of the diagrammatic language. This, in turn, evolved into a more *conceptual-based* approach, where there is an increased concern with the formal representation of the meta-concepts of the diagrams (e.g. class, object) in the formal model [FL95]. Most approaches in this area propose a formalisation of diagrams and underlying concepts (their representation), but the mapping itself is, usually, informally defined. Lately, however, there have been some approaches that also formalise the mapping.

Integrated methods work accompanied the evolution of semi-formal methods and formal modelling languages. Initially, they focused on structured methods, then object-oriented methods and most recently on the UML.

The following surveys integration approaches involving structured and OO methods with various formal methods. Although the survey covers several formal methods, there is a special emphasis on state-based model-oriented formal methods (see above), the class of FM to which Z (the formal method used in this thesis) belongs.

2.4.2 Structured methods

There are several approaches to integrate structured methods (Yourdon, SSADM, etc.) with formal modelling languages, such as VDM, Z, B, CCS and algebraic-languages. The following makes a survey of some those approaches; a more detailed study is given in [SFD92].

Integrations with VDM

- The approach of Larsen et al supplements SA/SD with VDM [LvKP+91, PvKP91]. It includes a VDM formalisation of data-flow diagrams (DFDs) and a method (SAVDM).
- The approach of Fraser et al intends to supplement SA with VDM [FKV91]. It includes two approaches to formalise DFDs: one is cognitive, the other is *rule-based*.
- Dick and Loubersac propose to enrich VDM with graphical notations based on SA, which are extended to capture more details of the VDM models [DL91].
- [Ham91] reports the development of a method based on Yourdon and VDM at Rolls-Royce, which is mainly Yourdon-based, but uses VDM to describe processes and data structures of DFDs.

Integrations with Z

- The approach of Semmens and Allen supplements Yourdon's structured analysis with Z [SA91]. It includes Z formalisations of entity-relationship diagrams (ERDs) and DFDs, and was embodied as a British Telecom proprietary method.
- The *SAZ* method of Polack et al is based on SSADM and Z [PWM93, MP95, Pol01]. SAZ includes a Z formalisation of ERDs, and may be used for quality assurance of the SSADM specification or integrated in the SSADM method.

Other Integrations

- Nagui-Raïss formalised extended ERDs in B with the aim of developing tool-support for formal modelling based on ERDs and a rule language stating system dynamics [NR94].
- Galloway proposed a variant of the Ward-Mellor method of structured analysis for real-time systems supplemented with CCS [Gal96].
- France proposed semantic interpretation of DFDs, for sequential and concurrent systems, in algebraic specification techniques [Fra92].

2.4.3 Object-Oriented Methods

Integrated approaches of structured methods try to do too much at once. They propose both formalisation of diagrammatic concepts and integration with the structured methods. This affects the quality of the formalisation which is, in many cases, superficial. This changes with OO methods, where there is a separation of concerns: on one side the formalisation of the underlying concepts of object-orientation, on the other, the integration with the OO method.

There are approaches to integrate OO methods with Z, Object-Z, B and CSP. [AP03] presents a detailed comparison involving Z and Object-Z.

Integrations with Z

The most influential work in this area is the one of Hall [Hal90b, Hal94], which provides a Z formalisation of OO concepts, such as class and object (this is discussed further in the ZOO chapters). This approach was used to supplement the Shlaer-Mellor method with Z [Ham94], to

combine OO analysis and Z [RA96], and, with some variations to Hall's original formalisation, in the work of France et al to integrate Z in the method Fusion [FLP97, FBLPG97, FBR96] and in the requirements specification technique based on Z and the UML [FGB00, BF98].

Other works involving Z include:

- Weber's approach to specify safety-critical control systems based on three views: the architectural view, based on object and class diagrams, the reactive view based on statecharts [Har87], and the functional view based on Z [Web96].
- Dupuy proposed a formalisation of OO class models and statecharts into Z [Dup00].

Integrations with Object-Z.

- Metamorphosis is a method that integrates Object-Z with common features of OO methods [Ara96]. It includes translation rules for static and dynamic models.
- Achatz and Schulte's method combines Fusion with Object-Z [AS97].
- Dupuy formalises OO class models and statecharts in Object-Z [DLCP97, Dup00].
- Kim and Carrington formalised UML class [KC00, KC99] and state-diagrams [KC02b, KC02a] in Object-Z; the mapping is also also formally defined.

B Combinations

- Lano et al formalised OO class models and statecharts in B [LH94, LHW96].
- Malioukov's approach integrates B with the OMT method [Mal98].
- Nguyen et al [FLN96, Ngu98, FLN01] formalised concepts of OO class and behavioural (state-transition and scenario diagrams of OMT) models in B.
- Laleau and Polack build on Nguyen's formalisation to develop a UML profile for information systems to support the description of structural and behavioural aspects [LP01a, LP01b]. There is ongoing work on tool-support for this approach [LP02].
- Meyers and Souquières supplemented the OMT method with B with the goal of formal verification, providing formalisations of OO class and state-transition diagrams [MS99].
- Snook and Butler proposed a UML profile for integrated development with B, including tool support (U2B) and a formalisation of UML class and state diagrams [SB00, SB06].
- Treharne's approach to integrate UML and B is based on the tool U2B, and involves use-case and class diagrams; only *stereotyped* classes require a complementary B model [Tre02].
- Ahmed et al. proposed an approach to extract statecharts from B specifications [HTVW02], with the aim of B-based development, but extracting UML diagrams for visualisation.

Integrations with CSP

- Engels et al. developed an approach for verifying the consistency of UML behavioural models using CSP and its analysis tools [EKH01].
- Davies and Crichton proposed a behavioural CSP semantics for UML models, where classes are mapped to processes [DC02]. The aim is to analyse the behaviour of UML models and refine UML models through the refinement of the CSP model.
- Fischer et al. proposed a semantics of UML-RT [6] structure diagrams in CSP for *architecture descriptions* [FOW01], which does not cover all features of structural diagrams of UML-RT and it is (the authors emphasise) one possible interpretation.

Other integrations

- Kwon formalised statecharts in the input language of the SMV model-checker [McM92], based on the operational semantics of conditional term rewriting systems [Kwo00].
- Paige and Ostroff developed a meta-modelling technique based on BON and PVS[7] to consistency-check and analyse meta-models [PO01].
- Reggio et al. propose an approach to formalise active classes with their statecharts in CASL, with the goal of formal-modelling and analysis based on the UML [RACH00].
- Liu et al. proposed semantic models of UML using the transition systems[8] formalism for requirements analysis [LLH02], involving class diagrams, object diagrams and use case diagrams, with the aim of verifying and analysing requirement analysis models [LLH02].
- Ramos et al proposed an approach to formalise UML-RT in Circus, by improving the formalisation of Fischer et al for CSP (see above) and with the aim of exploiting Circus to describe rich state structures and event-based behaviour [RSM05].

2.4.4 Discussion

The following analyses the research pursued on integrated approaches. It starts by discussing some issues involving the formalisation of diagrammatic concepts. Then, it compares integrated approaches. Next, it analyses wider issues related with development. Finally, it summarises the conclusions of the analysis.

Formalisation of diagrammatic concepts

It is not easy to formalise OO concepts in B. The natural way to do it is to use the structuring mechanisms of the formal language. So, in B this done by mapping each class to a B machine (like in [Ngu98]). B restricts the way machines can be composed in order to ensure that that proof is compositional, but this restricts the way classes may be related in a B model; as a result: associations must be unidirectional, and there are also problems in the formalisation of OO inheritance. Such restrictions do not occur in Z and Object-Z formalisations (see [AP03]

[6] UML-RT [SR98] is a UML profile for embedded and real-time software systems.
[7] PVS is a specification and verification system based on theorem proving [OSRSC99].
[8] Transition systems [HMP94] is a notation for general reactive systems.

and ZOO chapters of this thesis). To overcome these restrictions, recent formalisations do not use the structuring mechanisms of the B language, and formalise all OO concepts into a single B machine [BDG05, SB06]. This, however, is not a solution because it brings *scalability* problems; it might work for small models, but for bigger models it is likey to become cumbersome and complicate both proof and refinement.

The advantage of OO languages, like Object-Z, over non OO ones, such as Z, is that the concepts of OO are directly available as primitives of the language. This eases expression of OO concepts, but at cost of flexibility, and, in the case of Object-Z, abstraction. Flexibility because the representations provided by Object-Z may not be the ones that the specifier wants [AP03]. Abstraction because the structure of the Object-Z model resembles too much that of an OO program, with its rigid partitioning into classes, rather than an abstract model. (This is discussed further in the ZOO chapter).

Each integration approach has a purpose; the choice of formal method is driven by that purpose and the formalisation is driven by that choice. If the goal is to study interaction of a class (or component) with its environment then a language like CSP is the most appropriate choice and classes are formalised as CSP processes. But if the goal is to express class structures and relations, then a formalisation in a state-based language (such as Z) is more suitable. Approaches that try to address both concerns involve new integrated formal methods, such as the one aimed at Circus [RSM05].

Comparison

The intended use of formal methods in the integration has evolved over time. Initially, the integration was done with the aims of precision and well-defined semantics, but as formal methods tool support improved approaches started to focus more and more on the use of the formal method for analysis and consistency-checking of diagram-based models.

None of the approaches has achieved or attempted a full UML formalisation. Instead, they focus on a restricted subset of the language, mostly class diagrams and statecharts. Most approaches do not even attempt to formalise all features of a diagram, focusing on those features that are appropriate for the purpose of the approach. The aim is to enable model specification and analysis within a precise and rigorous framework.

Lately, there is an increased awareness that a formalisation constitutes only one possible interpretation of diagrams (e.g. [FOW01]). There is also an increased concern with variants of UML: either by creating new domain-specific UML profiles (e.g. [LP01a, SB06]) or by focusing on existing UML profiles (like UML-RT [FOW01, RSM05]).

Most approaches formalise the different diagrams in a single semantic model, expressed in a single formal language. Many types of diagrams may be supported, but they are all mapped into a single semantic model. The same approach was followed in the OCL formalisation of Richters (see above), where UML class diagrams and OCL constraints are expressed in terms of the same unified model. Essentially, there is a front-end made of diagrams (partial representations) and a back-end made of formal text (a total representation). This contrasts with the multi-view architecture of UML, where a model is expressed using different notations, which somehow articulate with each other; so a more natural semantics to support this architecture would be to give separate semantic models to each diagram and then compose them into a whole, but this is easier said than done.

Wider Issues

Most integrated approaches focus on formalisation of diagrams, but this is not the end of the story. Experience shows that not all properties of software systems are expressible with diagrams of semi-formal methods. Usually, constraints and operation specifications cannot be expressed diagrammatically. (In UML-only development this role is played by OCL.) This, however, is a problem for approaches whose goal is semi-formal only development, intending to use the formal method for automated consistency-checking; because consistency-checks must take constraints into account, there must be a way to put them into the formal model (by translation from OCL for example). So, either model constraints are being forgotten or there is an untold story: in practice, users are required to add these constraints to the formal model. Sometimes, it is just not clear how both descriptions, formal and semi-formal, are supposed to coexist.

As said above, the underlying theory is denotational semantics. Integrated approaches map diagrams into a formal model; in formal terms there is a function, which is uni-directional. The problem is that users are required to use the formal model and this uni-directionality means that changes in the formal model that affect the diagrammatic representations will not be reflected in the diagrams. (Of course, clever users also change the diagram when they know that it is affected by some change in the formal model; but the point is: this is not captured by the mapping.) In languages using denotational semantics, the semantic model is hidden, but, as argued above, it is not so in integrated approaches. One may try to minimise the exposure to the formal model, but it is not possible to hide everything. Ideally, integrated approaches should aim at considering diagrams and formal text as alternative representations, like in Alloy, where the user may choose to work with diagrams (where not all is expressible) or formal text (everything may be expressible).

As we have seen, it is difficult to hide the formal model, and the goal of model analysis makes it even harder. Let us take the simpler problem of model consistency-checking. Very few formal methods give consistency-checking at the push of a button. Only those based on model-checking do this, but this requires the model to be finite. Alloy also does this, but it does not give 100% certainty. In languages like Z, B, Object-Z and CSP, consistency needs to proved, and this requires expertise even if a theorem prover is being used. This becomes more complicated when we consider model analysis in general, because users are required to express the property that they want to check, and then prove it (or push a button in the case of model-checking). If the analysis reports some problem, the user is required to interpret the problem and then correct it (not a very easy thing to do if you are not an expert).

Definitely, in an integrated approach the formal model is a first class citizen that cannot be ignored. The formal model is always the full and complete model, its quality impacts on the overall development. If the formalisation is modular and conceptually clear, then it is easier for an expert to move between diagrams and formal model, easier to prove properties of the formal model, easier to refine models, and easier to design diagrams that cover the aspects not covered by existing diagrams. A formalisation that is cumbersome will make all this much harder, possibly rendering the whole approach impractical.

Summary

Integrated methods research has taught us many things: (a) diagrams and formal methods can coexist within the same development, (b) this coexistence is useful, providing many benefits

and (c) a formalisation of a diagrammatic language, like UML, is far from trivial.

Despite all this, like formal methods, integrated methods never really caught-on. The work that has been done is a step in the right direction, but much more is yet to be done. Some approaches just use the formal method for description, and neglect model analysis. Other approaches neglect the quality of the formalisation, which compromises the quality of the whole approach. All approaches mentioned here just focus on going from diagrams into formal model, providing no support for model analysis; to explore analysis, users are required to be experts in the formal language.

2.5 Reuse, domains and frameworks

So far, we have seen that there are methods of software development that are not very rigorous: semi-formal methods. On the other hand, there are these methods, which are mathematically rigorous, but not practical: formal methods. We have seen that we could integrate a semi-formal method with a formal one, making the semi-formal method rigorous, and easing the task of working with the formal method. But, somehow, this still seems far from the *micro-methods* advocated by Jackson.

Indeed, traditional semi-formal methods are as *universal* and *constructive* as their formal counterparts. A language like UML is simply huge. The semi-formal methods, themselves, are too general and rigid. They provide step-by-step guidance, but users can easily get lost in circumstances unforeseen by the method. Semi-formal methods are just too rigid, they are not designed to grow, evolve or learn with experience. This evolution component is essential to the idea of micro-methods advocated by Jackson [Jac98]:

> Traditional engineers do not use constructive methods for large decisions, nor do they use universal methods. They work on traditional problems for which they have developed traditional solutions. A new bridge or car or aeroplane or TV set is very closely similar to its predecessors, and shares the same structuring of problem and solution. The traditional engineer does not start with a clean sheet of paper. Successful designs evolve over many generations in communities in which many engineers are working on almost identical products.
>
> Specialisation is the inevitable precondition and accompaniment of this evolution of successful designs.

Jackson mentions specialisation and evolution, where there is actually an underlying concern of software engineering: *reuse*. Reuse, which was already one of the claimed advantages of the OO paradigm, is nicely defined in [BP89, p. xv] as:

> Software reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system. This reused knowledge includes artifacts such as domain knowledge, development experience, design decisions, architectural structures, requirements, design, code, documentation, and so forth.

Integrated methods were motivated by practical reasons. The goal was to define an environment to model software systems practically and rigorously. Integrated methods already give some reuse: rather than starting a formal model from a blank sheet of paper, the user

gets a formal model structure by drawing diagrams. That model structure is reused many times. This is nice, but still rigid. The missing bit seems this reuse of experience, evolution of knowledge and specialisation.

In the 1970s, Dijkstra [Dij72] and Parnas introduced an idea that would be very influential in the area of software engineering in general, and reuse in particular. The concept of *program families* is defined by Parnas as [Par76]:

> We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family member.

This idea went on to influence the fields of software architecture and component-based development. But it also influenced the work on *domain-engineering*.

2.5.1 Domain engineering and frameworks

In software engineering, domain is a word with two related, but slightly different meanings. One meaning emphasises the *real world* and refers to the different subject matters of the real world (such as banking, airline reservations, etc); it is a particular part of the world that can be distinguished because it is conveniently considered as a whole [Jac95b] — the *application or problem domain*. The other meaning of the word domain relates to the computing world, a computing domain is a family (or class) of systems for some application area. Of course, the two are related because an application domain has a family of related systems (a computing domain). *Domain* engineering tries to provide reusable solutions for a family of systems.

A seminal work in the area of domain engineering is Draco [Nei84, Nei80, Nei89]. Draco organises software construction knowledge into a number of related domains; each encapsulates the needs and requirements for different implementations of a family of systems. Domains are themselves structured into levels of abstraction, and so there are: application domains, modelling domains, and execution domains.

Also related is the notion of *framework*, which emerged in the early 1980s to support program development. The first frameworks originated from the Smalltalk-80 environment, which supported "reuse of design through frameworks of partially completed code" [Deu89]. They consisted of a collection of classes that users could reuse and extend. The Model-View-Controller (MVC) user interface of Smalltalk-80 was perhaps the first widely used framework [Bos00]. Similar frameworks are still popular, such as Microsoft's COM, OMG's CORBA, and Java's AWT and Beans.

The tailoring of OO frameworks is categorised as *white-box* and *black-box* [JF88]. The former uses inheritance as the mechanism of extension: framework users customise the framework through subclassing of framework classes. The latter customises frameworks based on parameterisation. In practice, however, most frameworks use both mechanisms.

The idea of frameworks as large-scale units of reuse has been adapted to higher-levels of abstraction. Consequently, the word framework has now many different meanings. Framework is a word that is part of the title of this thesis, so it is better if we define precisely its meaning in this scope (adapted from [JF88, Gab96]):

> A *framework* is an environment that embodies a set of artifacts that are used to develop software systems for a family of related problems. They can be customised,

> specialised, or extended to provide more specific, more appropriate, or slightly different capabilities.

Another related concept is that of *pattern*, which is discussed in detail in the next section. For now, patterns are another unit of reuse, more flexible than frameworks. Frameworks are increasingly being designed in terms of patterns [Joh92, BMA97, Joh97, RJ97].

The remainder of this section surveys approaches to develop modelling frameworks.

2.5.2 Modelling frameworks

Catalysis [DW98] proposes a method to build OO model frameworks based on UML. It uses templates (a form of patterns) to define frameworks and is designed with the aim of making models reusable assets. Catalysis defines the semantics of UML constructs adapted to the context in which they are used and is an approach that is inspired by the rigour of formal methods, but all its artifacts are produced in a semi-formal way.

Recently, there has been an interest in UML meta-modelling frameworks, which enable the creation of UML variants for some domain. Övergaard [Ö00] proposes BOOM, a meta-modelling framework for the definition of OO modelling languages. BOOM comprises a formal OO language called ODAL (defined using the π-calculus), which is used to specify the classes of the meta-model. It includes a set of class definitions that can be customised by sub-classing, and follows a denotational approach to language definition. Each definition consists of syntactic constructs and how they are interconnected, a mapping from constructs to BOOM classes (using ODAL expressions), and detailed semantics of the BOOM classes in ODAL.

Clark et al proposed the meta-modelling framework (MMF), an environment for the definition of UML variants [CEK02, CEK01, ACES01]. MMF comprises a language (MML) for defining modelling notations, a tool that checks and executes MML definitions, a method consisting of a model based approach to language definition, and a set of patterns, embodying good practice in language definition. MML is an OO-modelling textual notation, with an expression language based on OCL. It is used to define classes, associations, packages, and templates. The semantics of MML is defined like the FSL Alloy: through a translation to a small language called MML calculus, which is based on the ς-calculus[9] of Abadi and Cardelli [AC96]. MMF also follows a denotational approach to language definition. Each definition comprises three distinct components, concrete syntax, abstract syntax, and semantics, each with its own definition, which comprises a class diagram and OCL constraints. The concepts used in the models of the components are defined in MML. The mappings between language components is defined in terms of OCL constraints on associations between model elements.

2.5.3 Discussion

Designing a framework is a complex activity, which has to take into account the fact that frameworks may deteriorate. Any framework that is being used in the real world continually evolves, because its underlying domain keeps changing [Rie00]. Pattern-based approaches try to tackle this problem by making more flexible frameworks that are adaptable to a context.

[9] The ς-calculus is a meta-language used to define object-oriented languages; it is inspired by Church's λ-calculus used in the definition of procedural languages.

In all the approaches, there is the concern with defining the semantics of the modelling language precisely. Catalysis does it semi-formally by resorting to *package semantics*, defined with templates, diagrams and OCL. BOOM and MMF, however, do this formally, by using new formal languages.

BOOM and MMF differ in the way they use meta-modelling. MMF uses meta-modelling to define all language components: concrete syntax, abstract syntax and semantics. BOOM just uses meta-modelling to define the abstract syntax.

The approaches to defining language variants are attractive. They propose to define language variants in a modular way based on meta-modelling, but avoiding the shortcomings of current UML definitions by defining precisely the meta-modelling concepts using a formal meta-language. They introduce other improvements over the current UML definition, such as a clear boundary between abstract syntax and semantics, and better semantic definitions based on formal notations rather than natural language.

However, these approaches are based on new formal languages. Formal languages are not easy to get right, taking substantial time until they reach maturity. So far these formalisms have been used to define the semantics of static constructs. It is unlikely that they will cope with more complex aspects of behaviour, especially concurrency. Moreover, these new formalisms lack many useful mechanisms available in mature formal languages, such as theorem-proving, refinement calculus, and existing formal analysis tool support. In general, ODAL and MML are too implementation-oriented when compared with languages like Z and Alloy. This may be a limitation, as they may be inappropriate to state abstract descriptions (overspecification). ODAL includes syntactic constructs for method definitions, message passing, and typical programming language constructs such as iterative loops and 'if-then-else' clauses — making the underlying semantics more complex. The MML-calculus, the underlying formalism of MML, has a semantics based on the ς-calculus, which is already implementation-oriented, with its semantics defined operationally.

BOOM and MMF lack an important feature of Catalysis: templates. This is the key to variation and flexibility in framework definitions. The variation features provided by templates cannot be expressed in formal languages such as MML and ODAL.

2.6 Patterns

Over the last decade, the work of Christopher Alexander on architectural patterns [Ale79, AIS77] has been the source of inspiration for software engineering theorists. Patterns are described by Alexander as [AIS77, p.x]:

> Each pattern describes a problem that occurs over and over again in our environment. And then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Patterns were introduced in software engineering with the goal of *reuse*. They provide solutions to recurring problems in a context, and they document, promote and share expertise and best practice [SPT03b]. Alexander's patterns consist of a *context*, describing when a pattern is applicable, the *problem* that the pattern resolves in that context, and a *configuration*, that describes physical relationships that solve the problem [GHJV93]. Software patterns

share the same relation between context, problem and solution, providing abstract solutions that are adaptable to a context [GHJV93, Fow03].

Software patterns have had a tremendous impact. They are popular among software engineers, widely used and have their place in the software engineering terminology. The seminal patterns book [GHJV95] is a reference for OO designers and programmers; another popular reference in design and programming is [Lar98]. Patterns have also been applied to OO semi-formal requirements modelling [Coa92, Fow97]. Although originally introduced to support OO development, patterns, an engineering concept, have also been applied to other software engineering paradigms, such as formal development.

Patterns are not supposed to exist in isolation. Alexander explains [AIS77, p.xiii]:

> In short, no pattern is an isolated entity. Each pattern can exist in the world, only to the extent that is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it.

So, Alexander's pattern language is a networked assembly of patterns that can be used to generate architectural artifacts, such as, buildings, rooms and gardens.

Software patterns are described in various ways. They use informal templates and natural language text, software diagrams, program generic structures (such as C++ templates), pieces of program code, and, recently, there has been an attempt to describe patterns formally. Usually, software patterns are assembled in a catalogue.

The following surveys the use of patterns in formal development and approaches to the formal representation of patterns.

2.6.1 Patterns in formal development

The KAOS method, a goal-driven approach to requirements elaboration, uses refinement patterns [DvL96]. Goals in KAOS capture high-level requirements, they are represented as temporal-logic formulas and they need to be refined (or decomposed) into subgoals. To aid this refinement, KAOS includes a library (or catalogue) of goal refinement patterns and associated proofs. The instantiation of a refinement pattern gives the required subgoals, and enables the reuse of the proofs that are associated with the pattern; provided the conditions set by the pattern are met, then the goal decomposition is sound (no further proof is required). The approach is generic: goal refinements can be instantiated in many different situations, and instantiation is based on matching.

Dwyer et al [DAC99] propose a hierarchical catalogue of patterns to assist the construction of temporal-logic based specification. The catalogue aims to capture common idioms and experience with temporal-logic specifications. Each pattern includes a description and mappings (formalisations) of the pattern in several temporal logic formalisms (such as LTL and CTL[10]). The patterns are essentially temporal logic formulas, where variation is given by variables. Users are required to adapt patterns to their problems by performing a substitution of variables from the pattern description; mechanisation of this substitution is not addressed.

Stepney et al [SPT03a, SPT03b] develop a catalogue of patterns for the language Z. These patterns capture the experience and expertise of the authors and Z community in modelling with Z. Patterns are either described directly in Z (an instance of the pattern), or by resorting

[10]Linear Temporal Logic (LTL) and Computation Tree Logic (CTL).

to informal templates (intuitive names stand for parameters). The catalogue includes patterns that capture common idioms of Z specifications (such as Z promotion), together with several structural, architectural, development and domain patterns for Z.

Abrial [Abr05] demonstrates how the pattern concept can be applied to formal development, and how useful this is. In Abrial's approach, however, users are required to adapt the pattern manually to their contexts; [Abr05] mentions that further work on "generic instantiation" is required.

2.6.2 Describing patterns formally

Lano et al [LBG96] proposed a formal treatment of design patterns in VDM++ (an OO extension to VDM) based on the notion of system transformations (a refinement). The correctness of those refinements is proved in the object-calculus formalism.

Mikkonen [Mik98] formalised design patterns in DISCO, a method based on the temporal logic of actions. This allows the formalisation, refinement and instantiation of design patterns.

Eden et al [Ede00, Ede01] developed the language LePUS (LanguagE for Patterns Uniform specification) to formally represent OO design patterns. LePus is based on high-order monadic logic, and has been used to specify the design patterns that appear in the literature. A variant of LePUS, eLePUS, has also been proposed [RC01].

Reynoso et al [FMR01, RM00] formalise patterns of [GHJV95] in RAISE, based on a RAISE formalisation of OO concepts. The authors suggest the use of "renaming mappings" as a mechanism for pattern reuse.

Blazy et al [BGL03] formalised several OO design patterns (such as *composite* [GHJV95]) in B. Each design pattern is formalised in a single B machine, and reuse of the pattern is achieved through the B mechanism of machine inclusion and renaming.

Kim and Carrington [KC05] proposed a role-based approach to specify design patterns with Object-Z that is based on meta-modelling. The approach is illustrated with the patterns *factory* and *abstract factory* of [GHJV95]. Pattern reuse is achieved through the inheritance and renaming mechanisms of Object-Z.

2.6.3 Discussion

So far, patterns have been successful and have had an impact on practice. This is because they allow programming and design ideas to be represented, shared and reused. There are, however, some problems with current pattern technology and we are still far from Alexander's general pattern vision.

Adaptability to a context is the goal of research on patterns. Patterns are an artifact that can be reused, but in a flexible way. This flexibility contrasts with the stiff reuse technology based on components and library of routines that we have today. Current technology gives reuse only if it is small and general. Whenever something larger and domain-specific is required, then many different components and routines for every possible situation are needed [Big94], every possible combination needs to be predicted, there is no way to allow for some variation.

The problem with current approaches is that this flexibility is not mechanical. In fact, it is to enable flexibility that patterns are described in informal ways, so that they can be easily adaptable to many situations, but at the cost of the user. The limitations on current pattern

technology are precisely this. The following discusses compares and discusses the pattern reuse mechanisms proposed in the literature.

Pattern reuse mechanisms

In the literature, software patterns are usually described in terms of instances. To instantiate them, users are required to understand the pattern instance and its context, and then adapt it to their own context. The exception includes generic mechanisms of programming languages (such as C++ templates), logical formulas that can be reused by instantiation and informal templates. But the problem with informal templates is that their meaning is not precisely defined.

The approaches that describe patterns formally are limited in their expressibility and reuse mechanisms. The LePUS, B and Object-Z approaches are designed to deal with OO design patterns, but, as shown above, the notion of pattern is more general. LePUS is too tied to OO programming, not being abstract enough to deal with patterns in general. The same holds for Mikkonen's approach, which is too tied to OO design. The B and Object-Z approaches could be generalised, however. Basically, they represent patterns in a structure (a B machine or an Object-Z class schema), and then reuse the pattern by using inclusion and variable renaming. Although more general, this is still limited in its expressibility, and inclusion of some structure does not give us all the reuse that is required. For example, the promotion patterns of [SPT03a, SPT03b] cannot be described generically using a similar approach based on Z schema inclusion, which is more flexible that B machine inclusion and more general than Object-Z class inclusion (or inheritance). These same limitations are likely to apply to the approach of Lano that uses the VDM++ formalism (the author is not familiar with this formalism), because VDM is not so different from Z or B, and VDM++ is just an OO extension to VDM.

As shown above, the notion of pattern is an engineering concept that may be applied beyond the realm of OO development. As the works of Stepney et al and Abrial show, a pattern description may be formal, but that does not make it necessarily mechanically *reusable*. As it is suggested by Abrial, the mechanisms of *generic instantiation* need to be addressed.

The temporal-logic based approaches also propose mechanisms based on simple variable substitution and renaming, but again they are limited. KAOS defines refinement patterns as logical rewrite rules, which are instantiated based on variable substitution. This works with simple logical formulas (maybe goals are just simple formulas), which do not have much structure. But as suggested by Alexander and demonstrated in [SPT03a, SPT03b], realistic patterns require more structure and more means of expressibility. The patterns in [DAC99] suggests variable substitution and renaming for the patterns of the catalogue, but this mechanism is never formally defined.

The current generic technology that is available (C++ templates, Z generics) is still limited. The Z schema generics, for instance, which give more structure than logic formulas, would not be enough to represent the patterns that this thesis needs (See [ASP03] for further details and [PS99] for what happens when you do use generics). That is also why the patterns in [SPT03a, SPT03b] are not expressible in terms of Z generics.

All pattern descriptions in the literature lack a fundamental feature of patterns: *generativity*. It is not possible to mechanically generate instances from a pattern description. This is also emphasised by Alexander [Ale99]:

> To what extent does [a pattern] language *generate* (hence produce) entities (buildings, rooms, groups of buildings, neighbourhoods, etc.) that are whole and coherent? [...] We were always looking for the capacity of a pattern language to generate coherence, and that was the most vital test used, again and again, during the process of creating a language. The language was always seen as a whole. We were looking for the extent to which, as a whole, a pattern language would produce a coherent entity.
>
> Have you done that in software pattern theory? Have you asked whether a particular system of patterns, taken as a system, will generate a coherent computer program? If so, I have not heard about that. [...] So far, as a lay person trying to read some of the works that have been published by you in this field, it looks to me more as though mainly the pattern concept, for you, is an inspiring format that is a good way of exchanging fragmentary, atomic, ideas about programming. Indeed, as I understand it, that part is working very well. But these other two dimensions, (1) the moral capacity to produce a living structure and (2) the generativity of the thing, its capability of producing coherent wholes – I haven't seen very much evidence of those two things in software pattern theory. Are these your shortcomings? Or is it just because I don't know how to read the literature?

2.7 Conclusions

This chapter has surveyed current approaches to model-driven development (MDD), and approaches to reuse that see an increasing application in MDD: frameworks and patterns.

We have seen that semi-formal methods are far from rigorous, and formal-methods far from practical. We have also seen that it is possible to integrate the two, but, despite some improvements, the integrations that have been proposed are still not practical, which limits their engineering value. Moreover, the integration still falls behind the vision of very specialised software engineering methods. Despite the benefits of the integration, both semi-formal and formal methods are still too general, and further specialisation is required.

The main benefits of specialisation are reuse and optimisation. We have seen how frameworks, a large unit of reuse, are being used in software engineering in general and MDD in particular. A framework targets some application domain and is used to build software engineering artifacts (model or code) by reuse.

To enhance their flexibility, we have seen how frameworks are being increasingly defined in terms of patterns. We also discussed the flexibility of patterns as units of reuse; they represent a solution to a problem that is adaptable to a context. However, current pattern technology still limits their full potential.

This thesis proposes GeFoRME, an approach to build frameworks for rigorous MDD development based on patterns. GeFoRME tries to address many limitations with current approaches that have been found in the literature:

- It is designed with the aim of integrating both formal and semi-formal methods, but tries to provide full support for the whole development. A GeFoRME framework consists of approaches to modelling, analysis and refinement, where formal models are generated from templates. The $UML + Z$ framework developed in the next chapters realises some of these goals.

- GeFoRME also tries to diminish the dependency on the expert that characterises integrated approaches. This is done by trying to provide full support for the whole model development, but also by trying to hide the formal model as much as possible, and have formal models generated from templates. As it will be shown for $UML + Z$, the expert is still necessary, but lay developers can also participate in the development.

- This chapter has mentioned that diagrammatic notations have multiple interpretations, so GeFoRME is designed to target some application domain, where variability in the semantics of diagrammatic concepts is substantially reduced. The $UML + Z$ framework developed in the next chapters has a flexible semantic domain, which allows alternative representations of some concepts within the same development that can be represented visually as UML stereotypes.

- This thesis tries to improve on existing MDD frameworks based on the UML, by doing it formally and with mature formalisms. GeFoRME is designed for the integration of languages of formal development and diagrammatic languages. The $UML+Z$ framework developed in this thesis uses a mature formal modelling language, Z.

- The shortcomings of current pattern technology have been identified. Most approaches do not provide a formal representation of patterns that enable an important feature of patterns: generativity. The next chapter develops the template language of GeFoRME: FTL. This allows formal representations of templates, and enables formal reasoning with template representations of formal models. FTL is then used to build the catalogue of templates of the $UML + Z$ framework.

The next chapter develops the key tool of GeFoRME: the language FTL and its meta-proof approach. The subsequent chapters use FTL to develop the catalogue of $UML + Z$ templates, which allow the generation of formal models by instantiating templates.

To make a pattern explicit, we merely have to make the inner structure of the pattern clear. [...] We must first define some physical feature of the place, which seems worth abstracting.

Christopher Alexander [Ale79]

3

A formal template language enabling meta-proof

We have seen that GeFoRME's core unit of reuse is the template, a very concrete form of pattern. This chapter develops a language of templates and an approach to proof with templates of formal models. The Formal Template Language (FTL) and its meta-proof approach enable the construction of catalogues of templates and meta-theorems for GeFoRME frameworks.

It is time to give a more precise meaning to the term *template*. We said that templates are a concrete form of patterns, which gives an idea, but it does not say much. There is a definition in the Oxford English Dictionary that helps: "An instrument used as a gauge or guide in bringing any piece of work to the desired shape." This is the meaning that is used here: the "pieces of work" of the definition are, here, sentences of some formal language. So, here, templates capture the shape (or form) of sentences and are used to generate sentences whose form is as specified by the template.

The use of template-like representations is not new. In computer science and mathematics, the so-called *schematic* representations capture the form of sentences of some language, which cannot be represented in the language itself. For example, the law of the associativity of conjunction in the propositional calculus is usually expressed as:

$$(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$$

But P, Q and R do not belong to the language of predicates; instead, these *meta-linguistic variables* stand for any predicate of the language. This is the sort of thing we are looking for, but we need more flexible and powerful mechanisms of abstraction.

This chapter starts by motivating FTL and meta-proof. Then, it gives some intuition about FTL by providing a brief overview of its main constructs. Next, the language is formally defined with a syntax and a semantics. Then, the chapter moves on to define a calculus for the instantiation of FTL templates, which is important for meta-proof. Next, meta-proof is illustrated with the formal language Z, and a template logic for Z is proposed. Finally, the chapter reflects on the reported work and compares it with related work.

3.1 Motivation

Templates of some form appear often in the computer science literature. For example, a popular Z book [WD96, p. 150] introduces the Z schema with a template:

Name

declaration

---

predicates

This is an *informal template*, it says that a schema has a name, a set of declarations and a set of predicates. Intuitively, we guess that the names of the template are to be substituted by values, which is typically confirmed by a template instance:

Bank

$accs : \mathbb{P}\ ACCID$

$accSt : ACCID \pfun Account$

$\mathrm{dom}\ accSt = accs$

The problem with informal templates, however, is that it is difficult to distinguish the template from the instance, and is difficult to know what instantiations are valid, making it impossible to reason rigorously with them. FTL can represent templates of any formal language precisely; it is used here with Z. The informal template given above in FTL is:

$\ll Name \gg$

$[\![\ \ll declaration \gg\ ]\!]$

$[\![\ \ll predicate \gg\ ]\!]$

A variable within $\ll\gg$ denotes a *placeholder*, which is substituted by a value when the template is instantiated; a term within $[\![\ \ ]\!]$ denotes a list, which is replaced by many occurrences of the term in the instantiation. To make the *Bank* schema, $\ll Name \gg$ is substituted by the schema name *Bank*, $[\![\ \ll declaration \gg\ ]\!]$ by two Z declarations, and $[\![\ \ll predicate \gg\ ]\!]$ by one Z predicate.

3.1.1 Meta-Proof

The form of Z sentences can be represented as FTL templates. It is also possible to explore templates of Z for reasoning (or proof). This has practical value: template developers can establish *meta-theorems* for templates that are applicable to all instantiations of the templates involved. This is motivated with an example.

In Z, the introduction of a state space definition of an abstract data type (ADT), such as *Bank* above, into a specification, entails a demonstration that the description is consistent: at least one state satisfying the description should exist. This normally involves defining the initial state of the ADT (the initialisation) and proving that the initial state does exist (the initialisation theorem). The *Bank* is initialised assuming that in the initial state there are no accounts:[1]

[1] A Z schema may have an horizontal or a vertical form. *BankInit* is here given in the horizontal form.

$$BankInit == [\ Bank\ ' \mid accs' = \varnothing \land accSt' = \varnothing\ ]$$

The consistency of *Bank* is demonstrated by proving the Z conjecture,

$$\vdash?\ \ \exists\ BankInit \bullet true$$

A proof-sketch of this conjecture is (see appendix B for a brief explanation of formal proof, and for a definition of the inference rules used):

$$\begin{array}{l}
\vdash\ \ \exists\ BankInit \bullet true \\
\equiv [\text{By } \exists\ Sc \text{ (twice)}] \\
\vdash\ \ \exists\ accs' : \mathbb{P}\ ACCID;\ accSt' : ACCID \pfun Account \bullet accs' = \varnothing \land accSt' = \varnothing \\
\equiv [\text{By one-point}] \\
\vdash\ \ \mathrm{dom}\ \varnothing = \varnothing \land \varnothing \in \mathbb{P}\ ACCID \land \varnothing \in ACCID \pfun Account \\
\equiv [\text{By set theory and propositional calculus}] \\
\mathrm{true}
\end{array}$$

This is proved automatically in the Z/Eves [Saa97] theorem prover.

The *Bank* Z schema above is an instance of a very common structure in Z specifications: the state of a *promoted* ADT [WD96, SPT03a, SPT03b]. The theorem proved above applies just to the *Bank* ADT, but does it apply also to all promoted ADTs that are similar in form to *Bank*? If it does, can this result be proved once and for all?

Bank was generated from a template, but that template is too general and not useful for the kind of investigation that we want to do. Instead, we need a more restricted template representing a promoted Z ADT, of which *Bank* is an instance. A promoted ADT comprises a set of identifiers, a function mapping identifiers to state, a predicate restricting the mapping function, and a predicate representing an optional state invariant:

$$\begin{array}{l}
\lessdot P \gtrdot == [\ \lessdot ids \gtrdot : \mathbb{P}\ \lessdot ID \gtrdot;\ \lessdot st \gtrdot : \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \mid \\
\qquad \mathrm{dom}\ \lessdot st \gtrdot = \lessdot ids \gtrdot \land \lessdot I \gtrdot\ ]
\end{array}$$

(Promoted ADT's without the optional invariant, such as *Bank* above, are instantiated from this template by instantiating $\lessdot I \gtrdot$ with the value *true*.)

Likewise, we represent as templates the empty initialisation of a promoted ADT, and the initialisation conjecture:

$$\begin{array}{l}
\lessdot P \gtrdot Init == [\ \lessdot P \gtrdot\ ' \mid \lessdot ids \gtrdot' = \varnothing \land \lessdot st \gtrdot' = \varnothing\ ] \\
\vdash?\ \ \exists\ \lessdot P \gtrdot Init \bullet \mathrm{true}
\end{array}$$

We can reason with these templates by analysing their *well-formed* instantiations. In those cases, P, id and st hold names, ID and S are sets, and I is a predicate. By expanding the template schemas using the laws of the schema calculus, and applying the one-point rule (see proof above), we get the formula,

$$\begin{array}{l}
\vdash \mathrm{dom}\ \varnothing = \varnothing \land \varnothing \in \mathbb{P} \lessdot ID \gtrdot \land \varnothing \in \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \\
\qquad \land \lessdot I \gtrdot' [\lessdot ids \gtrdot' := \varnothing, \lessdot st \gtrdot' := \varnothing]
\end{array}$$

which reduces to, $\lessdot I \gtrdot' [\lessdot ids \gtrdot' := \varnothing, \lessdot st \gtrdot' := \varnothing]$. If $\lessdot I \gtrdot$ is instantiated with *true*, then the formula reduces to *true*. This establishes two meta-theorems, where the second is a

specialisation (or a corollary) of the former, that are applicable to all promoted ADTs instantiated from these templates. The specialised meta-theorem gives the nice property of *true by construction*: whenever these templates are instantiated, such that $\ll I \gg$ is instantiated with *true*, then the initialisation conjecture is simply *true*. Even when $\ll I \gg$ is not instantiated with *true*, the formula to prove is simpler than the initial one.

The argument outlined above is rigorous and valid, but it is not formal. To work towards formal meta-proof, so that tool support is possible, a formal semantics for the template language is required.

In the following, FTL is given a brief introduction followed by an overview of its formal definition. Then, the instantiation calculus of FTL, a calculus for the partial instantiation of templates, is presented. Finally, a meta-proof approach for Z based on FTL is developed and illustrated for the rigorous proof above.

3.2 A short introduction to FTL

FTL expresses templates that can be instantiated to yield sentences of some language (the target language). FTL is general in the sense that it can capture the form (or shape) of sentences of any formal language, and although designed with Z in mind, it is not tied to Z or to any other language.

FTL's abstraction mechanism is based on *variables*, which allow the representation of variation points in structures. Variables have, in FTL, their usual mathematical meaning: they denote some value in a scope. Template instantiation is, essentially, substitution of variables by values.

A consequence of the generality of FTL is that it is possible to generate meaningless sentences from templates. FTL templates assist the author; they do not remove the obligation to check that what the author writes is sensible. For meta-proof, however, generated sentences should have a meaning, so, only those instantiations of a template that are *well-formed* are considered (discussed in detail below).

The following illustrates the main constructs of FTL.

Text. A template may contain text of the target language, which is present in every instance. For example, the trivial template, *true*, always yields, *true*, when instantiated. Usually, templates comprise text combined with other constructs.

Placeholder. A placeholder is represented by enclosing one variable within $\ll\gg$. Placeholders, when not within lists, denote one variable occurrence, and they are substituted by the value assigned to the variable when the template is instantiated. The template, $\ll x \gg : \ll t \gg;\ \ll y \gg : \ll t \gg$, includes four placeholders and three variables, x, t and y; this can be instantiated with the substitution set, $\{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}, y \mapsto \text{“}b\text{”}\}$, to yield: $a : \mathbb{N};\ b : \mathbb{N}$.

List. A list comprises one list term, a list separator (the separator of the instantiated list terms) and a string representing the empty instantiation of the list, and it is represented by enclosing the list term within $[\![\ ]\!]$. The list term is a combination of text, parameters and possibly other lists. Often, the abbreviated form of lists, without separator and empty instantiation, is used.

A placeholder within a list denotes an *indexed set* of variable occurrences. This means that $\ll x \gg$ and $[\![\ll x \gg]\!]$ actually denote different variable occurrences; $\ll x \gg$ denotes an occurrence of the variable x, but $[\![\ll x \gg]\!]$ denotes the occurrence of the indexed set of variables, $\{x_1, \ldots, x_n\}$.

The template, $[\![\ll x \gg : \ll t \gg]\!]_{(\text{“;”}, \text{“}\{\}\text{”})}$, can be instantiated with the sequence of substitution sets, $\langle \{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}\}, \{x \mapsto \text{“}b\text{”}, t \mapsto \text{“}\mathbb{P}\ \mathbb{N}\text{”}\} \rangle$. This yields: $a : \mathbb{N};\ b : \mathbb{P}\,\mathbb{N}$. Lists can be instantiated with an empty instantiation, $\langle\ \rangle$; here this gives $\{\}$, the list empty instantiation.

As every variable occurrence must denote the same value within a scope, the same sequence of substitution sets are shared across lists. For example, the template,

$$[\![\ll x \gg : \ll t \gg]\!]_{(;,\{\})}$$
$$[\![\ll y \gg : \ll t \gg]\!]_{(;,\{\})}$$

has two lists, which share the substitutions of the variable t; the substitution $\langle \{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}, y \mapsto \text{“}p\text{”}\}, \{x \mapsto \text{“}b\text{”}, t \mapsto \text{“}\mathbb{P}\ \mathbb{N}\text{”}, y \mapsto \text{“}q\text{”}\} \rangle$, yields:

$$a : \mathbb{N};\ b : \mathbb{P}\,\mathbb{N}$$
$$p : \mathbb{N};\ q : \mathbb{P}\,\mathbb{N}$$

A consequence of this sharing of the variable t is that those two template lists always generate the same number of list terms (two in the example above) and valid substitution sequences for those lists must have the same length.

Choice. The FTL choice construct expresses *choice* of template expressions. That is, only one of the available choices is present in the instantiation. There are two kinds of choice: optional and multiple. Optional choice is represented by enclosing a template expression within $(\!|\ |\!)^?$ and it means that the enclosed expression may be present in the instantiation or not. Multiple choice is represented by enclosing the choice expressions within $(\!|\ |\!)$ and separated by $[]$ (see below); it means that one of the choice expressions must be present in the instantiation. Choices are instantiated with a choice-selection, a natural number, indicating the selected choice; non-selection takes the value zero.

The template $(\!|\ \ll x \gg : \ll t \gg\ |\!)^?$ can be instantiated with $(1, \{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}\})$, to yield: $a : \mathbb{N}$. To avoid the presence of the expression in the instantiation, the template can be instantiated with $(0, \{\})$, which simply yields the empty string. In the multiple-choice template,

$$(\!|\ \ll x \gg : \ll t \gg\ [] \ll x \gg : \ll t_1 \gg \nrightarrow \ll t_2 \gg\ |\!)$$

the first choice is instantiated with $(1, \{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}\})$ to yield: $a : \mathbb{N}$; the second with $(2, \{x \mapsto \text{“}f\text{”}, t_1 \mapsto \text{“}\mathbb{N}\text{”}, t_2 \mapsto \text{“}\mathbb{N}\text{”}\})$ to yield: $f : \mathbb{N} \nrightarrow \mathbb{N}$.

3.3 The Formal Definition of FTL

FTL has a denotational semantics [Ten76] based on its abstract syntax. FTL should generate sentences of some target language upon instantiation, and this is naturally represented in the domain of strings (the semantic domain).

FTL has been fully specified in Z (using Z's ISO standard definition [ISO02b]). The full definitions (including the Z one) can be found in appendix A. The following defines FTL using a representational style that is easier to read; it combines Z sets and operators, the Backus-Naur form (BNF) to define the syntax, and the equational style of denotational semantics.

3.3.1 Syntax

This section defines the syntactic sets of the language. The set of all identifiers (I) gives variables to construct placeholders. The set of all text symbols ($SYMB$) is used to construct the set of all strings, which are sequences of text symbols. There is also a special symbol to denote the empty string (defined as an empty sequence):

$$[I, SYMB] \qquad Str == \operatorname{seq} SYMB \qquad \Lambda : Str$$

The remaining syntactic sets are defined by structural induction, using the BNF (Backus-Naur Form) notation. The syntactic set of template expressions is made up of those objects that are either an atom (A), a choice (C), or either of these followed by another expression:

$$E ::= A \mid C \mid A\ E \mid C\ E$$

The syntactic set of choice comprises optional and multiple choice; optional choice is formed by one expression and multiple-choice by a sequence of expressions (set CL):

$$C ::= (\!|\ E\ |\!)^{?} \mid (\!|\ CL\ |\!) \qquad CL ::= E_1\ [\!]\ E_2 \mid E\ [\!]\ CL$$

The set of atoms, A, comprises placeholders, text (T), and lists (L); a placeholder is formed by an identifier, the name of a variable:

$$A ::= \ll I \gg \mid T \mid L$$

The set of lists, L, comprises two forms of list: normal and abbreviated. A normal list comprises a list term (set LT, a sequence of atoms), a list separator (SEP) and the empty instantiation of the list (EI); the abbreviated form just includes the list term:

$$L ::= [\![\ LT\ ]\!]_{(SEP,EI)} \mid [\![\ LT\ ]\!] \qquad LT ::= A \mid A\ LT$$

The abbreviated list is just syntactic sugar for a normal list with the empty string as separator and empty instantiation:

$$[\![\ LT\ ]\!] = [\![\ LT\ ]\!]_{(\Lambda,\Lambda)}$$

List separators, list empty instantiations and text are just strings:

$$SEP ::= Str \qquad EI ::= Str \qquad T ::= Str$$

3.3.2 Semantics

A template denotes a set of strings, corresponding to all its possible instances (Fig. 3.1).[2] The semantics of FTL could be specified in this direct way by calculating all possible instances of

[2] But only a subset of these strings has a meaning in the target language.

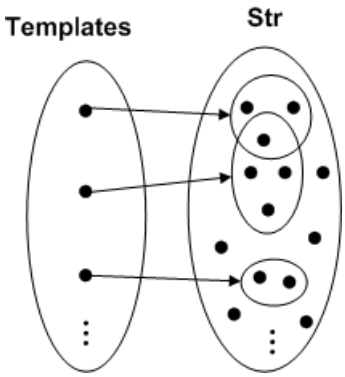


Figure 3.1: Templates and the sets of strings they denote (all possible instances of a template).

a template. That is, given the set of template expressions (E) defined above, the meaning of a template would be given by the function:

$$\mathcal{M} : E \rightarrow \mathbb{P}\ Str$$

But this does not explain *instantiation*. That is, how users generate sentences from templates. A more useful semantics should take this into account.

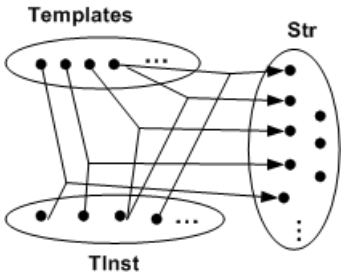


Figure 3.2: A template and a legal substitution yields a string.

The same meaning of templates as denoting a set of strings, can be achieved by taking an indirect path, which considers the action of instantiation. A template instantiation consists of substitutions for the template's variables and selections for the template's choices. A template has a set of all legal instantiations, and it is by instantiating a template with these instantiations that we get the set of all strings denoted by the template.

So, instead, the semantics is defined by an instantiation function, which calculates the string (sentence) generated by instantiating a template with an instantiation (figure 3.2). That is, a semantic function such as:

$$\mathcal{M} : E \rightarrow TInst \nrightarrow Str$$

The semantics of FTL is defined following this approach. The semantic functions are defined by structural induction on the syntax of FTL. To make definitions easier to read, the presentation follows an equational style, which omits quantifier declarations. The full Z definition of the semantics is given in appendix A.

The semantic functions for the different syntactic constructs of FTL are defined below. They use some auxiliary definitions that are defined in appendix A.2.

Semantics of atoms

Atoms are instantiated with substitutions for the variables that occur within placeholders, which may stand-alone or be within lists.

The environment structure (Env), defined as a partial function from identifiers (variables) to strings (values), represents a set of variable substitutions:

$$Env == I \nrightarrow Str$$

This allows the instantiation of placeholders that are not within lists. For example, the template $\ll x \gg : \ll t \gg$ is instantiated with, $\{x \mapsto a, t \mapsto \mathbb{N}\}$, an instance of Env, to yield $x : \mathbb{N}$. As a placeholder within a list denotes an indexed set of variable occurrences, it seems natural to instantiate these variables with a sequence of substitution sets (seq Env), but this does not work with nested lists. So, a recursive structure is required, the environment tree:

$$TreeEnv ::= \mathsf{tree}\ \langle\!\langle Env \times \mathrm{seq}\ TreeEnv \rangle\!\rangle$$

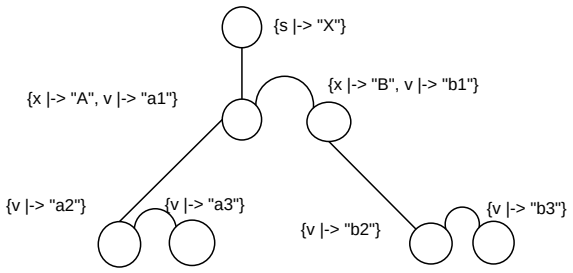
TreeEnv comprises one environment, to instantiate the placeholders that stand alone in the current scope, and a sequence of *TreeEnv* to instantiate the placeholders that are within the lists of the current scope. This structure is better understood with an example.

Illustration. Consider the template with a nested list, and one instance of that template:

$[\ll s \gg]$
$[\![\ \ll x \gg ::= \ll v \gg\ [\![\ \mid\ \ll v \gg\ ]\!]\ ]\!]$

$[\ X\ ]$
$A ::= a1 \mid a2 \mid a3$
$B ::= b1 \mid b2 \mid b3$

A pictorial representation of the instantiation that yields the instance above is:



The actual instantiation is:

$$\begin{array}{l}\mathsf{tree}(\{s \mapsto \text{“}X\text{”}\},\\ \quad \langle \mathsf{tree}(\{x \mapsto \text{“}A\text{”}, v \mapsto \text{“}a1\text{”}\},\\ \quad\quad \langle\ \mathsf{tree}(\{v \mapsto \text{“}a2\text{”}\}, \langle\ \rangle), \mathsf{tree}(\{v \mapsto \text{“}a3\text{”}\}, \langle\ \rangle)\ \rangle),\\ \quad \mathsf{tree}(\{x \mapsto \text{“}B\text{”}, v \mapsto \text{“}b1\text{”}\},\\ \quad\quad \langle\ \mathsf{tree}(\{v \mapsto \text{“}b2\text{”}\}, \langle\ \rangle), \mathsf{tree}(\{v \mapsto \text{“}b3\text{”}\}, \langle\ \rangle)\ \rangle\ \rangle)\end{array}$$

The semantic function extracts the required substitutions from the *TreeEnv* structure.

Semantic Function. The semantic function takes an atom (A) and a *TreeEnv*, and returns a string (Str):

$$\mathcal{M}_{\mathcal{A}} : A \rightarrow TreeEnv \nrightarrow Str$$

This is defined by the equations (where $e \in Env$ and $ste \in \text{seq } TreeEnv$):

$$\mathcal{M}_{\mathcal{A}}(\ T\ )(\mathsf{tree}(e, ste)) = T$$
$$\mathcal{M}_{\mathcal{A}}(\ \ll I \gg\ )(\mathsf{tree}(e, ste)) = \begin{cases} e(I) & \text{if } I \in \text{dom } e \\ undefined & \text{otherwise} \end{cases}$$
$$\mathcal{M}_{\mathcal{A}}(\ L\ )(\mathsf{tree}(e, ste)) = \mathcal{M}_{\mathcal{L}}(\ L\ )\ ste$$

If the atom is a piece of text (a string), then the text is returned. If the atom is a placeholder, then either there is a substitution for the placeholder's variable in the current environment (e) and the substitution is returned, or otherwise and the function is undefined. If the atom is a list, then the list instantiation function is called in the current sequence of environment trees (ste).

Semantics of lists

First, the meaning of list terms (LT) is defined. Its semantic function takes a list term and a *TreeEnv* structure and returns a string:[3]

$$\mathcal{M}_{\mathcal{LT}} : LT \rightarrow TreeEnv \nrightarrow Str$$

This is defined by the equations (where $te \in TreeEnv$):

$$\mathcal{M}_{\mathcal{LT}}(A)\ te = \mathcal{M}_{\mathcal{A}}\ (A)\ te$$
$$\mathcal{M}_{\mathcal{LT}}(A\ LT)\ te = \mathcal{M}_{\mathcal{A}}\ (A)\ te \mathbin{+\!\!+} \mathcal{M}_{\mathcal{LT}}\ (LT)\ te$$

If the list term is just one atom, then the atom is instantiated in the the current tree. If it is one atom followed by the remaining list term, then instantiation of the atom in the current tree is concatenated with the instantiation of the remaining list term ($\mathcal{M}_{\mathcal{LT}}$).

The semantic function for lists takes a list (L) and a sequence of *TreeEnv* and returns a string:

$$\mathcal{M}_{\mathcal{L}} : L \rightarrow \text{seq } TreeEnv \nrightarrow Str$$

This is defined by the following equations (where $ste, stet \in \text{seq } TreeEnv$). If the *TreeEnv* sequence is empty (there are no more substitutions for the list variables) the list's empty instantiation is returned:

$$\mathcal{M}_{\mathcal{L}}(\ [\![\ LT\ ]\!]_{(SEP,EI)}\ )\ \langle\rangle = EI$$

If the *TreeEnv* sequence is not empty, the list term (LT) needs to be written for each substitution of its variables in the *TreeEnv* sequence, considering that the substitutions may finish before the end of the sequence.[4] So, two cases need to be considered: (a) there is a substitution for the list term in the current environment and (b) otherwise. In the first case, the instantiation of the list term in the current environment is concatenated with the

[3] The operator $+\!\!+$ denotes string concatenation (defined as sequence concatenation).

[4] This because the same *TreeEnv* sequence may instantiate many lists, which may not have the same number of instantiations. Consider the lists $[\![\ll x \gg]\!]$ and $[\![\ll y \gg]\!]$; the first list may be instantiated five times and the second six times, so the last environment in the *TreeEnv* sequence will not have a substitution for x.

instantiation for the rest of the *TreeEnv* sequence (recursive call to $\mathcal{M}_{\mathcal{L}}$). In the second case, the list's empty instantiation is written:

$$\mathcal{M}_{\mathcal{L}}(\ [\![\ LT\ ]\!]_{(SEP,EI)}\ )(\langle \mathsf{tree}(e, ste)\rangle \frown stet)$$
$$= \begin{cases} \mathcal{M}_{\mathcal{LT}}(LT)(\mathsf{tree}(e, ste)) & \\ \qquad +\!\!+\ \mathcal{M}_{\mathcal{L}}(\ [\![\ SEP\ LT\ ]\!]_{(\Lambda,\Lambda)}\ )(stet) & \text{if } \mathcal{V}_{\mathcal{LT}}\ LT \cap \operatorname{dom} e \neq \varnothing \\ EI & \text{otherwise} \end{cases}$$

The predicate $\mathcal{V}_{\mathcal{LT}}\ LT \cap \operatorname{dom} e \neq \varnothing$ is true if there is a substitution for the list term in the current environment (e).[5] The recursive call to $\mathcal{M}_{\mathcal{L}}$ modifies the list's term, separator and empty instantiation: the list term is modified to be the list separator followed by the term (so that in lists that are instantiated more than once the separator is written every time), and the separator and empty instantiation become the empty string (so that the separator is not written twice and the empty instantiation is not written at all).[6]

Semantics of choice

A choice expression is instantiated by selecting one of its choices. The selection is represented as a natural number. In optional choice, 0 means non-selection and a positive natural number the selection of the only choice provided. In multiple choice, a positive natural number indicates the selected choice.

The semantic function for choice lists takes a choice list (CL) and a positive natural number and, unlike the other semantic functions, returns an expression rather than a string:

$$\mathcal{M}_{\mathcal{CL}} : CL \to \mathbb{N}_1 \nrightarrow E$$

This is because the instantiation of choices is done in two steps: (a) the selection is made and (b) the selected choice (an expression) is instantiated in the usual way. The semantic function for choices does the first step, the second step uses the semantic function for normal expressions (below).

The function $\mathcal{M}_{\mathcal{CL}}$ is defined by the equations (where $n \in \mathbb{N}_1$):

$$\mathcal{M}_{\mathcal{CL}}(E_1\ [\!]\ E_2)\ 1 = E_1$$
$$\mathcal{M}_{\mathcal{CL}}(E_1\ [\!]\ E_2)\ 2 = E_2$$
$$\mathcal{M}_{\mathcal{CL}}(E\ [\!]\ CL)\ n = \begin{cases} E & \text{if } n = 1 \\ \mathcal{M}_{\mathcal{CL}}\ CL\ (n-1) & \text{otherwise} \end{cases}$$

The first two specify selection when there are just two choices; the left choice is selected when the selection value is 1 and the right when it is 2. The third equation specifies selection when there are more than two choices; two cases need to be considered: (a) if the selection (n) has the value 1 then the leftmost choice is selected, and (b) otherwise the function $\mathcal{M}_{\mathcal{CL}}$ is called recursively eliminating the leftmost choice and decrementing the selection value:

The semantic function for choice takes a choice (C) and a natural number, and returns an expression:

[5] The function $\mathcal{V}_{\mathcal{LT}}$ gives the set of variables of a list term (see appendix A.2.2 for its Z definition).

[6] Suppose that we want to instantiate the list $[\![\ \lessdot P \gtrdot\ ]\!]_{(\wedge,\mathrm{true})}$ to yield $2 < 3 \wedge 4 > 5$; according to the semantics, after writing the first term, $2 < 3$, the list is modified to $[\![\ \wedge\ \lessdot P \gtrdot\ ]\!]_{(\Lambda,\Lambda)}$, so that the second term is written as $\wedge\ 4 > 5$, and the end of the instantiation is written as the empty string.

$$\mathcal{M}_{\mathcal{C}} : C \to \mathbb{N} \nrightarrow E$$

This is defined by the equations (where $n \in \mathbb{N}_1$):

$$\mathcal{M}_{\mathcal{C}}(\llbracket\ E\ \rrbracket^?)\ 0 = \Lambda$$
$$\mathcal{M}_{\mathcal{C}}(\llbracket\ E\ \rrbracket^?)\ n = E$$
$$\mathcal{M}_{\mathcal{C}}(\llbracket\ CL\ \rrbracket)\ n = \mathcal{M}_{\mathcal{CL}}\ (CL)\ n$$

There are two equations for optional choice: in the first, if the selection value is 0 (non-selection) then an expression made up of the empty string is returned; in the second, the selection value is a positive natural number (n), and the choice expression is returned. If it is a multiple choice, then the function for choice lists is called.

Semantics of expressions

The global environment (*GEnv*) structure, which represents a total template instantiation, builds on the *TreeEnv* structure; it comprises a sequence of natural numbers, the selections of the template's choices, and a *TreeEnv* structure, the substitutions of the template's variables:

$$GEnv == \operatorname{seq}\mathbb{N} \times TreeEnv$$

The semantic function takes an expression and a *GEnv* and returns a string:[7]

$$\mathcal{M}_{\mathcal{E}} : E \to GEnv \nrightarrow Str$$

This is defined by the equations (where $chs \in \operatorname{seq}\mathbb{N}$, $te \in TreeEnv$, $n \in \mathbb{N}$):

$$\mathcal{M}_{\mathcal{E}}(\ A\ )(chs, te) = \mathcal{M}_{\mathcal{A}}(\ A\ )\ te$$
$$\mathcal{M}_{\mathcal{E}}(\ C\ )(\langle n \rangle, te) = \mathcal{M}_{\mathcal{E}}(\ \mathcal{M}_{\mathcal{C}}(\ C\ )\ n\ )(\langle\ \rangle, te)$$
$$\mathcal{M}_{\mathcal{E}}(\ A\ E\ )(chs, te) = \mathcal{M}_{\mathcal{A}}(\ A\ )\ te\ +\!\!+\ \mathcal{M}_{\mathcal{E}}(\ E\ )(chs, te)$$
$$\mathcal{M}_{\mathcal{E}}(\ C\ E\ )(\langle n \rangle \frown chs, te) = \mathcal{M}_{\mathcal{E}}(\ \mathcal{M}_{\mathcal{C}}(\ C\ )\ n +\!\!+_E\ E\ )(chs, te)$$

If the expression is an atom, then the atom is instantiated in the environment tree (call to $\mathcal{M}_{\mathcal{A}}$). If the expression is a choice, then the choice is resolved by consuming the selection value (call to $\mathcal{M}_{\mathcal{C}}$) and the selected choice is instantiated (call to $\mathcal{M}_{\mathcal{E}}$). If the expression is an atom followed by another expression, then the instantiation of the atom (call to $\mathcal{M}_{\mathcal{A}}$) is concatenated with the instantiation of the rest of the expression (recursive call to $\mathcal{M}_{\mathcal{E}}$). Finally, if the expression is a choice followed by another expression, then the choice is resolved (call to $\mathcal{M}_{\mathcal{C}}$) and concatenated with the rest of the expression, and the concatenated expression is instantiated (recursive call to $\mathcal{M}_{\mathcal{E}}$).

3.3.3 Illustration: Instantiating templates using the semantics

The language definition can be used to instantiate templates. This is illustrated here for the template of the promoted Z ADT:

$$TPADT == \text{“}\langle\!\langle P \rangle\!\rangle == [\ \langle\!\langle ids \rangle\!\rangle : \mathbb{P}\langle\!\langle ID \rangle\!\rangle;\ \langle\!\langle st \rangle\!\rangle : \langle\!\langle ID \rangle\!\rangle \nrightarrow \langle\!\langle S \rangle\!\rangle\ |$$
$$\operatorname{dom}\ \langle\!\langle st \rangle\!\rangle = \langle\!\langle ids \rangle\!\rangle \land \langle\!\langle I \rangle\!\rangle\ ]\text{”}$$

[7] $+\!\!+_E$ is expression concatenation, see appendix A.2 for definition

First, the substitution set for the template is specified in the environment e:

$$e == \{P \mapsto \text{“}Bank\text{”}, ids \mapsto \text{“}accs\text{”}, st \mapsto \text{“}accst\text{”}, ID \mapsto \text{“}ACCID\text{”},\\ S \mapsto \text{“}Account\text{”}, I \mapsto \text{“}true\text{”}\}$$

The environment tree and the global environment build upon e. As there are no lists or choices, the sequences of trees and choice selections are empty:

$$t == \mathsf{tree}(e, \langle\rangle) \qquad\qquad g == (\langle\rangle, t)$$

The template is now instantiated by applying the semantic functions:

$$\begin{array}{l}
\mathcal{M}_{\mathcal{E}}(TPADT)(g)\\
\equiv [\text{By defs of } TPADT, \mathcal{M}_{\mathcal{E}} \text{ and } g]\\
\mathcal{M}_{\mathcal{A}}(\lessdot P \gtrdot)(t) \mathbin{+\!\!+} \mathcal{M}_{\mathcal{E}}(== [\ \lessdot ids \gtrdot : \mathbb{P}\ \lessdot ID \gtrdot;\ \lessdot st \gtrdot : \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \mid\\
\quad \mathrm{dom}\ \lessdot st \gtrdot = \lessdot ids \gtrdot \land \lessdot I \gtrdot\ ])(g)\\
\equiv [\text{by defs of } \mathcal{M}_{\mathcal{A}} \text{ and } \mathcal{M}_{\mathcal{E}}]\\
e(P)\ \mathbin{+\!\!+} \mathcal{M}_{\mathcal{A}}(\text{“} == [\ \text{”})(t) \mathbin{+\!\!+} \mathcal{M}_{\mathcal{E}}(\lessdot ids \gtrdot : \mathbb{P}\ \lessdot ID \gtrdot;\\
\quad \lessdot st \gtrdot : \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \mid \mathrm{dom}\ \lessdot st \gtrdot = \lessdot ids \gtrdot \land \lessdot I \gtrdot\ ])(g)\\
\equiv [\text{by defs of } e, \mathbin{+\!\!+}, \mathcal{M}_{\mathcal{A}}, \mathcal{M}_{\mathcal{E}}]\\
\text{“}Bank == [\ \text{”} \mathbin{+\!\!+} \mathcal{M}_{\mathcal{A}}(\lessdot ids \gtrdot)(t) \mathbin{+\!\!+} \mathcal{M}_{\mathcal{E}}(: \mathbb{P} \lessdot ID \gtrdot;\\
\quad \lessdot st \gtrdot : \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \mid \mathrm{dom}\ \lessdot st \gtrdot = \lessdot ids \gtrdot \land \lessdot I \gtrdot\ ])(g)
\end{array}$$

By applying the semantic functions in this way, we obtain:

$$\begin{array}{l}
\text{“}Bank == [\ accs : \mathbb{P}\ ACCID;\ accSt : ACCID \pfun Account \mid\\
\qquad \mathrm{dom}\ accSt = accs \land \mathrm{true}]\text{”}
\end{array}$$

3.3.4 Testing

FTL's semantics has been tested using the Z/Eves theorem prover, based on its Z definition. The proved theorems demonstrate that the semantic functions when applied to sample templates and instantiations yield the expected Z sentence. Test conjectures were chosen to give a good coverage of all FTL constructs and all possible instances of templates containing those constructs.

The derivation presented above is demonstrated by proving the conjecture:

$$\begin{array}{l}
\vdash ?\ \mathcal{M}_{\mathcal{E}}(TPADT)(g)\\
\quad = \text{“}Bank == [\ accs : \mathbb{P}\ \ ACCID;\ accSt : ACCID \pfun Account \mid\\
\qquad \mathrm{dom}\ \ accSt = accs \land \mathrm{true}]\text{”}
\end{array}$$

And this conjecture is proved automatically in the Z/Eves prover.

3.4 The instantiation calculus

The semantics of FTL is defined by calculating the string (sentence) that is generated from a template given an instantiation. The instantiations of the semantics are *total*, that is, there must be substitutions for all the template's variables and selections for all the template's

choices. Sometimes, however, we may be interested in *partially* instantiating a template, for example, substituting just one variable in a template and leaving the rest of the template unchanged.

The *instantiation calculus* (IC) of FTL is an approach to transform templates by taking instantiation decisions on a step-by-step basis. In this setting, a template has been fully instantiated when there no placeholders, choices or lists left; all instantiation decisions have been resolved.

To have a better idea of the calculus, consider a transformation of the promoted ADT template where the variable P is substituted with "*Bank*":

$$Bank == [\ \ll ids \gg : \mathbb{P}\ \ll ID \gg;\ \ll st \gg : \ll ID \gg \pfun \ll S \gg \mid \\ \mathrm{dom}\ \ll st \gg = \ll ids \gg \land \ll I \gg\ ]$$

This is a more refined template, one in which the decision of substituting the variable P has been taken. By applying a similar sequence of transformations, the *Bank* schema would be reached.

The calculus is defined by instantiation functions, which take a template expression (E) and a partial instantiation, and return the partial instantiation of the given template:

$$I : E \to PInst \to E$$

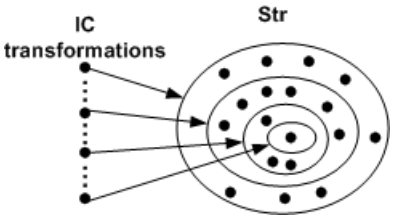


Figure 3.3: Templates transformed with the instantiation calculus and the sets of strings they denote.

Templates refined with the IC are just like any other FTL template: they denote a set of strings. As templates are refined with the calculus, the sets of strings they denote become smaller and smaller, until they cannot be refined any further and just denote a singleton set of strings (Fig. 3.3).

The IC is divided into placeholders, lists and choice. The IC functions, like in the semantics, are defined by structural induction on the syntax of FTL. The IC was tested, in a similar fashion to the semantics (see above), by using its Z definition and the Z/Eves prover. The full Z definition of the IC can be found in A.3.

3.4.1 Placeholders

The IC function simply replaces placeholders by a substitution of their variables, provided that a substitution has been provided. To represent a set of variable substitutions, the *Env* structure (see above) is reused.

The instantiation function for atoms takes an atom (A) and an environment, and returns another atom:

$$\mathcal{IP}_{\mathcal{A}} : A \to Env \to A$$

This is defined by the equations ($e \in Env$):

$$\mathcal{IP}_{\mathcal{A}}(T)\ e = T$$

$$\mathcal{IP}_{\mathcal{A}}(\ll I \gg)\ e = \begin{cases} e(I) & \text{If } I \in \mathrm{dom}\ e \\ \ll I \gg & \text{otherwise} \end{cases}$$

$$\mathcal{IP}_{\mathcal{A}}(L)\ e = L$$

If the atom is text then the text is returned. If the atom is a placeholder, then either there is a substitution for the variable in the environment (e) and instantiation takes place, or otherwise and the placeholder is returned uninstantiated. If the atom is a list then the list is returned; lists have their own function.

The placeholder choice functions do not resolve the choice, they just replace the placeholders, for which a substitution has been provided, in the choice expressions. The multiple choice function takes a choice list (CL) and an environment, and returns a choice list:

$$\mathcal{IP}_{\mathcal{CL}} : CL \rightarrow Env \rightarrow CL$$

This is defined by the equations ($e \in Env$):

$$\mathcal{IP}_{\mathcal{CL}}(E_1 \; [\!] \; E_2) \; e = \mathcal{IP}_{\mathcal{E}} \; (E_1) \; e \; [\!] \; \mathcal{IP}_{\mathcal{E}} \; (E_2) \; e$$
$$\mathcal{IP}_{\mathcal{CL}}(E_1 \; [\!] \; CL) \; e = \mathcal{IP}_{\mathcal{E}} \; (E_1) \; e \; [\!] \; \mathcal{IP}_{\mathcal{CL}} \; (CL) \; e$$

This says to replace all placeholders within the choice expressions based on the substitutions given in e (calls to $\mathcal{IP}_{\mathcal{E}}$), keeping the choice unresolved.

Optional choice is similar. The function takes a choice (C) and a set of substitutions (Env), and returns another choice:

$$\mathcal{IP}_{\mathcal{C}} : C \rightarrow Env \rightarrow C$$

The equations request the choice expressions to be instantiated (calls to $\mathcal{IP}_{\mathcal{E}}$ and $\mathcal{IP}_{\mathcal{CL}}$), but the choice remains unresolved:

$$\mathcal{IP}_{\mathcal{C}}((\![\; E \;]\!)^?) \; e = (\![\; \mathcal{IP}_{\mathcal{E}} \; (E) \; e \;]\!)^?$$
$$\mathcal{IP}_{\mathcal{C}}((\![\; CL \;]\!)) \; e = (\![\; \mathcal{IP}_{\mathcal{CL}} \; (CL) \; e \;]\!)$$

The instantiation function for expressions receives an expression and an environment and returns an expression:

$$\mathcal{IP}_{\mathcal{E}} : E \rightarrow Env \rightarrow E$$

This is defined by the equations:

$$\mathcal{IP}_{\mathcal{E}}(A) \; e = \mathcal{IP}_{\mathcal{A}} \; (A) \; e$$
$$\mathcal{IP}_{\mathcal{E}}(C) \; e = \mathcal{IP}_{\mathcal{C}} \; (C) \; e$$
$$\mathcal{IP}_{\mathcal{E}}(A \; E)e = (\mathcal{IP}_{\mathcal{A}} \; (A) \; e) \;\; (\mathcal{IP}_{\mathcal{E}} \; (E) \; e)$$
$$\mathcal{IP}_{\mathcal{E}}(C \; E)e = (\mathcal{IP}_{\mathcal{C}} \; (C) \; e) \;\; (\mathcal{IP}_{\mathcal{E}} \; (E) \; e)$$

Essentially, the atoms and choices that make the template expression are instantiated recursively for the given set of substitutions (e).

3.4.2 Lists

IC's notion of partial instantiation complicates the semantics of lists. The following discusses some of the issues.

Some issues

The straightforward way to instantiate a list is as a sequence of substitution sets. For example, the template, $[\![\lessdot x \gtrdot : \lessdot t \gtrdot]\!]_{(\text{“;”},\text{“\{\}”})}$, is instantiated with, $\langle\{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}\}, \{x \mapsto \text{“}b\text{”}, t \mapsto \text{“}\mathbb{P}\ \mathbb{N}\text{”}\}\rangle$, to yield: $a : \mathbb{N};\ b : \mathbb{P}\,\mathbb{N}$; the empty instantiation, $\langle\rangle$, yields $\{\}$. But there is a problem with the empty instantiation when there is more than one list. For example, if the template,

$$[\![\lessdot x \gtrdot : \lessdot t \gtrdot]\!]_{(;,\ \{\})}$$
$$[\![\lessdot y \gtrdot]\!]$$

is instantiated with $\langle\rangle$, then, both lists are instantiated empty, there is no information to tell which list should be instantiated. In the IC, however, we want to be able to select which lists to be instantiated empty.

Therefore, a better approach is to consider a list instantiation as comprising a set of variables and a sequence of substitutions for those variables. The strategy is: instantiate all lists containing these variables with the following substitution sequence. In the example above, $(\{y\}, \langle\rangle)$ instantiates the second list to the empty string, and leaves the first list uninstantiated. The instantiation, $(\{x, t\}, \langle\{x \mapsto \text{“}a\text{”}, t \mapsto \text{“}\mathbb{N}\text{”}\}, \{x \mapsto \text{“}b\text{”}, t \mapsto \text{“}\mathbb{P}\ \mathbb{N}\text{”}\}\rangle)$, instantiates only the first list, leaving the second uninstantiated.

However, multiple lists cannot always be instantiated independently. A placeholder within a list denotes an indexed set of variables occurrences, and variables should be substituted with the same value in every occurrence. So, if two lists have a variable in common then their instantiations are no longer independent. For example, consider templates T_1 (left) and (T_2),

$$[\![\lessdot x \gtrdot : \lessdot t \gtrdot]\!]_{(\text{“;”},\text{“\{\}”})} \qquad [\![\lessdot x \gtrdot]\!]_{(\text{“,”},\ \text{“”})}$$
$$[\![\lessdot x \gtrdot]\!]_{(\text{“,”},\ \text{“”})} \qquad [\![\lessdot x \gtrdot = \lessdot v \gtrdot]\!]_{(\text{“;”},\ \{\})}$$
$$[\![\lessdot v \gtrdot \in \lessdot t \gtrdot]\!]_{(\text{“}\wedge\text{”},\ \text{“true”})}$$

T_1 has two lists dependent on the variable x. T_2 has a chain of list dependencies (list-dependency is a transitive relation)

The problem is: what should be result of the partial instantiation $(\{x\}, \langle\{x \mapsto \text{“}a\text{”}\}, \{x \mapsto \text{“}b\text{”}\}\rangle)$? What should happen to the uninstantiated t and v? The solution is to rename those uninstantiated variables to an indexed variable occurrence. So those templates are refined to:

$$a : \lessdot t_1 \gtrdot;\ b : \lessdot t_2 \gtrdot \qquad a, b$$
$$a, b \qquad a = \lessdot v_1 \gtrdot;\ b = \lessdot v_2 \gtrdot$$
$$\lessdot v_1 \gtrdot \in \lessdot t_1 \gtrdot \wedge \lessdot v_2 \gtrdot \in \lessdot t_2 \gtrdot$$

And the instantiation can now be completed with a call to the function for placeholders.

So far, we have been working in the simpler world of simple lists. Things get more complex with nested lists, because, as discussed in the semantics, it is not possible to instantiate such lists with a sequence of environments. In the semantics, this was solved by introducing a recursive structure.

In the IC, the strategy is to instantiate them with a sequence of environments, but on a level-by-level basis. That is, one call to the instantiation function for lists instantiates all lists at the top level, leaving the inner lists uninstantiated; further calls to the instantiation function instantiate the inner lists, until there are no more lists to instantiate.

Suppose the template:

$$[\![\ll x \gg ::= \ll v \gg [\![\mid \ll v \gg]\!]]\!]$$

The outer-list can be instantiated with the environment sequence,

$$\langle \{x \mapsto \text{“}A\text{”}, v \mapsto \text{“}a1\text{”}\}, \{x \mapsto \text{“}B\text{”}, v \mapsto \text{“}b1\text{”}\} \rangle$$

but now some care is required because the straightforward partial instantiation,

$$A ::= a1 \ [\![\mid \ll v \gg]\!]$$
$$B ::= b1 \ [\![\mid \ll v \gg]\!]$$

with two lists and the variable v, is not the correct one because v denotes the same variable occurrence in both lists. Hence, it is not possible to obtain the following instantiation of the original template,

$$A ::= \ a1 \ \mid \ a2 \ \mid \ a3$$
$$B ::= \ b1 \ \mid \ b2 \ \mid \ b3$$

which would be obtainable with the FTL semantics.

A placeholder within a list denotes an indexed set of variable occurrences. This indexing can have many levels depending on the nesting of the list (e.g. an indexed set of indexed variable occurrences). So, in the partial template above, the problem is that the placeholders with the variable v should actually refer to distinct variables. To solve this problem, after the first instantiation, those variables need to be renamed to an indexed occurrence:

$$A ::= a1 \ [\![\mid \ll v_1 \gg]\!]$$
$$B ::= b1 \ [\![\mid \ll v_2 \gg]\!]$$

This way, the desired instantiation can be obtained by another call to the instantiation function with:

$$\langle \{v_1 \mapsto \text{“}a_2\text{”}, v_2 \mapsto \text{“}b_2\text{”}\}, \{v_1 \mapsto \text{“}a_3\text{”}, v_2 \mapsto \text{“}b_3\text{”}\} \rangle$$

The following defines the variable renaming mechanism, a function to obtain variable dependencies, and then the instantiation functions.

Variable Renaming

To handle the renaming of variables, the set I (of identifiers) is partitioned in two, MI (main identifiers) and SI (sub-identifiers):

$$\frac{MI, SI : \mathbb{P}\, I}{\langle MI, SI \rangle \text{ partitions } I}$$

It is assumed that all identifiers that are used in a user-defined template (before any instantiation) are from the set MI; SI includes all identifiers used for renaming during instantiation steps.

The renaming mechanism is based on generating indexed sub-identifiers from an original identifier. For example, p, has sub-identifiers, $p_1, p_2, \ldots, p_n$. So, there is a total injective

function that, given an identifier and a positive natural number, say n, yields the sub-identifier with index n:

$$subI : I \times \mathbb{N}_1 \rightarrowtail SI$$

This function associates n indexed sub-identifiers with an identifier. The function is injective so that two identifiers do not have sub-identifiers in common. For example, given an identifier p and the number 1, the function yields: p_1.

To perform the actual renaming, there are functions defined on the syntax of FTL. The renaming function for atoms takes an atom and a positive natural number and returns another atom:

$$\mathcal{R}n_{\mathcal{A}} : A \rightarrow \mathbb{N}_1 \rightarrow A$$

This function renames all parameters that may be included in the atom. This is defined by the equations:

$$\mathcal{R}n_{\mathcal{A}} \lessdot I \gtrdot \ n = \lessdot subI(I, n) \gtrdot$$
$$\mathcal{R}n_{\mathcal{A}} \ (T) \ n = T$$
$$\mathcal{R}n_{\mathcal{A}} \ (L) \ n = \mathcal{R}n_{\mathcal{L}} \ (L) \ n$$

If the atom is a placeholder, then the its variable is renamed by its n-th sub-identifier. If it is text, no renaming takes place. If it is a list, then the list renaming function is called.

The remaining functions for the other syntactic constructs are similarly defined. See appendix A.3 for their full definitions.

List Dependency

As said above, a variable is list-dependent on another if they are within the same list, or, as list dependency is a transitive relation, if there is some other variable to which they are both list-related. Since this information is important to define the IC, there are functions that give the variable dependencies. The variable-dependency set grows as the template is examined.

The list-dependency function for lists, takes a list and a set of variables and returns another set of variables:

$$\mathcal{DV}_{\mathcal{L}} : L \rightarrow \mathbb{P}\, I \rightarrow \mathbb{P}\, I$$

This is defined by the equations (where $is \in \mathbb{P}\, I$):

$$\mathcal{DV}_{\mathcal{L}}([\![\ LT \ ]\!]_{(SEP,EI)})is = \begin{cases} \mathcal{V}_{\mathcal{LT}} \ LT \cup is & \mathcal{V}_{\mathcal{LT}} \ LT \cap is \neq \varnothing \\ is & \text{otherwise} \end{cases}$$

If the LT has variables in common with the set is, then a dependency has been found and the union is returned. Otherwise, is is returned: no dependency has been found.

The remaining functions are similarly defined. See appendix A.3 for complete definitions.

Instantiation Functions

A list instantiation is a set of indexing variables and a sequence of substitution sets (environments); this is defined in the list environment:

$$LEnv == \mathbb{P}\, I \times \operatorname{seq} Env$$

On the list instantiations coming from the user there are some restrictions. So the constrained list environment is defined as:

$$LEnvC : \mathbb{P}\, LEnv$$

this says that domain of each environment is the set of indexing variables:

$$(is, \langle\rangle) \in LEnvC$$
$$(is, \langle e \rangle) \in LEnv \Leftrightarrow \operatorname{dom} e = is$$
$$(is, \langle e \rangle \frown se) \in LEnvC \Leftrightarrow \operatorname{dom} e = is \land (is, se) \in LEnvC$$

The list instantiation function takes a list (L) and a list environment, and returns the instantiated list as a template expression (E):

$$\mathcal{IL}_{\mathcal{L}} : L \rightarrow LEnv \rightarrow E$$

During the instantiation of lists, renaming may take place. As renaming is based on indexing, there is an auxiliary instantiation function, which keeps a counter of indexes (a positive natural number):

$$\mathcal{IL}_{\mathcal{LC}} \;:\; L \rightarrow LEnv \times \mathbb{N}_1 \rightarrow E$$

The first function is defined by one sole equation in terms of the second one:

$$\mathcal{IL}_{\mathcal{L}}\ (L)\ se = \mathcal{IL}_{\mathcal{LC}}\ (L)\ (se, 1)$$

A list is instantiated with the counter starting at 1.

The second function is the one that actually does the instantiation. Its first equation is as follows (where $is \in \mathbb{P}\, I, n \in \mathbb{N}$):

$$\mathcal{IL}_{\mathcal{LC}}\ (\llbracket\ LT\ \rrbracket_{(SEP,EI)})\ ((is, \langle\rangle), n) = \begin{cases} EI & \text{if } \mathcal{V}_{\mathcal{LT}}\ LT \cap is \neq \varnothing \\ \llbracket\ LT\ \rrbracket_{(SEP,EI)} & \text{otherwise} \end{cases}$$

The instantiation sequence is empty. Then, (a) either the variables to instantiate are part of the list term, and the result is the list's empty instantiation, or (b) the list has no variables to instantiate, and the list is returned uninstantiated.

The second equation is (where $e \in Env, se \in \operatorname{seq} Env$):

$$\mathcal{IL}_{\mathcal{LC}}\ (\llbracket\ LT\ \rrbracket_{(SEP,EI)})\ ((is, \langle e \rangle \frown se), n)$$
$$= \begin{cases} \mathcal{R}n_{\mathcal{E}}(\mathcal{IP}_{\mathcal{E}}(\mathcal{LT}_2\mathcal{E}\ LT)\ e)\ n \\ \qquad \dagger\!\!\dagger_E\ \mathcal{IL}_{\mathcal{LC}}(\llbracket\ SEP\ LT\ \rrbracket_{(\Lambda,\Lambda)})((is, se), n+1) & \text{If } \mathcal{V}_{\mathcal{LT}}\ LT\ \cap is \neq \varnothing \\ \llbracket\ LT\ \rrbracket_{(SEP,EI)} & \text{otherwise} \end{cases}$$

The instantiation sequence is an environment followed by a sequence of environments. Then, two cases need to be considered: (a) there is a substitution for the variables of the list term in the current environment (e) and (b) otherwise.[8] In the first case, the list term is instantiated in the current environment (e) and concatenated with the instantiation of the list for the rest of the environment sequence (se); the instantiation of the list term in the current environment involves: converting the list term to an expression (call to $\mathcal{LT}_2\mathcal{E}$)[9], instantiating the expression in the current environment (call to $\mathcal{IP}_\mathcal{E}$) and renaming the variables that remain from the instantiation based on the current index (call to $\mathcal{R}n_\mathcal{E}$ with index n). In the second case, the list's is returned uninstantiated.

The remaining instantiation functions are defined in the usual way. The instantiation function for atoms, $\mathcal{IL}_\mathcal{A} : A \rightarrow LEnv \rightarrow E$, calls $\mathcal{IL}_\mathcal{L}$ whenever the atom is a list:

$$\mathcal{IL}_\mathcal{A}\ (\ll I \gg)\ le = (\ll I \gg)$$
$$\mathcal{IL}_\mathcal{A}\ (T)\ le = T$$
$$\mathcal{IL}_\mathcal{A}\ (L)\ le = \mathcal{IL}_\mathcal{L}\ (L)\ le$$

The instantiation function for expressions includes an auxiliary definition, $\mathcal{IL}_{\mathcal{E}0} : E \rightarrow LEnv \rightarrow E$, which propagates the instantiation throughout the template expression:

$$\mathcal{IL}_{\mathcal{E}0}\ (A)\ le = \mathcal{IL}_\mathcal{A}\ (A)\ le$$
$$\mathcal{IL}_{\mathcal{E}0}\ (C)\ le = \mathcal{IL}_\mathcal{C}\ (C)\ le$$
$$\mathcal{IL}_{\mathcal{E}0}\ (A\ E)\ le = (\mathcal{IL}_\mathcal{A}\ A\ le) +\!\!+_E\ (\mathcal{IL}_{\mathcal{E}0}\ E\ le)$$
$$\mathcal{IL}_{\mathcal{E}0}\ (C\ E)\ le = (\mathcal{IL}_\mathcal{C}\ C\ le)\ (\mathcal{IL}_{\mathcal{E}0}\ E\ le)$$

The actual expression function takes an expression and a constrained list environment and returns another expression:

$$\mathcal{IL}_\mathcal{E} : E \rightarrow LEnvC \rightarrow E$$

This is defined as a call to the auxiliary expression function, but first all variable dependencies of the variable set is are collected (call to $\mathcal{DV}_\mathcal{E}$):

$$\mathcal{IL}_\mathcal{E}(E)(is, se) = \mathcal{IL}_{\mathcal{E}0}(E)(\mathcal{DV}_\mathcal{E}\ E\ is, se)$$

See appendix A.3 for the definition of the list instantiation function for choice (functions $\mathcal{IL}_\mathcal{C}$ and $\mathcal{IL}_{\mathcal{CL}}$).

3.4.3 Choice

As in the semantics, each choice is instantiated with a natural number, and a template expression is choice-instantiated with a sequence of choice selections.

To define the instantiation for choice, the semantic functions of choice and choice list can be reused, because they transform templates rather than returning a string (see semantics of choice, above, for more details).

[8] Two cases need to be considered because the end of a list instantiation is not necessarily the end of the instantiation sequence (see discussion in semantics of lists).

[9] See appendix A.3 for $\mathcal{LT}_2\mathcal{E}$'s definition.

The choice instantiation function for expressions, takes an expression (E) and a sequence of natural numbers and returns an expression:

$$\mathcal{IC}_{\mathcal{E}} : E \rightarrow \operatorname{seq} \mathbb{N} \nrightarrow E$$

This is defined by the equations:

$$\mathcal{IC}_{\mathcal{E}}\ (A)\ \langle\rangle = A$$
$$\mathcal{IC}_{\mathcal{E}}\ (C)\ \langle n \rangle = \mathcal{M}_{\mathcal{C}}\ C\ n$$
$$\mathcal{IC}_{\mathcal{E}}\ (A\ E)\ sn = A\ (\mathcal{IC}_{\mathcal{E}}\ E\ sn)$$
$$\mathcal{IC}_{\mathcal{E}}\ (C\ E)\ (\langle n \rangle \frown sn) = (\mathcal{M}_{\mathcal{C}}\ C\ n) +\!\!+_E\ (\mathcal{IC}_{\mathcal{E}}\ E\ sn)$$

If the expression is just one atom, then the instantiation is the atom itself because an atom does not include choices. If the expression is a choice, then the choice is instantiated (call to $\mathcal{M}_{\mathcal{C}}$). If the expression is an atom followed by a remaining expression, then the atom followed by the choice-instantiation of the remaining expression is returned. If the expression is a choice followed by a remaining expression, then the choice is instantiated (call to $\mathcal{M}_{\mathcal{C}}$) and concatenated with the choice-instantiation of the remaining expression.

3.4.4 Testing

The IC was throughly tested using proof with the Z/Eves theorem prover. Besides checking the correctness of the instantiation functions, it was also checked whether the IC is consistent with the semantics of FTL. The application of a sequence of partial instantiations with the calculus on a template should yield the same result as an equivalent total instantiation of the template with the semantic functions.

We need a way to compare the IC with the semantics of FTL. The IC functions return template expressions, whereas semantic ones return strings. To compare them we consider empty expression-level instantiation,

$$gei == (\langle\ \rangle, \mathsf{tree}(\varnothing, \langle\ \rangle))$$

which, when applied with the semantic function to a template that has no placeholders, lists or choices (a fully instantiated template), yields the corresponding string. This can be used to obtain the string representation of a template that has been totally instantiated with the IC.

Now, it can be proved that the instantiation calculus yields the same as the semantic functions, in the instantiation of the *TPADT* template for the *Bank* schema. Considering the definitions of section 3.3.3, this is demonstrated by proving the conjecture:

$$\vdash ?\ \ \mathcal{M}_{\mathcal{E}}\ (\mathcal{IP}_{\mathcal{E}}\ TPADT\ e)\ gei = \mathcal{M}_{\mathcal{E}}(TPADT)\ g$$

And this is proved automatically in Z/Eves.

3.5 Meta-proof

The formal definition of FTL and IC can be used to support meta-proof. The approach presented here is developed for Z, but it is more general; the same ideas can be applied to any

language with a proof logic. Appendix B.1 briefly introduces formal proof and its relation with Z.

Meta-proof with Z is a proof on a generalisation of a commonly occurring Z conjecture. First, the setting for meta-proof with Z is presented, by discussing the link between FTL and Z for meta-proof, and by considering generalisation and the concept of characteristic instantiation. Then, the approach is illustrated for the initialisation conjecture of promoted ADTs.

3.5.1 Linking FTL with Z

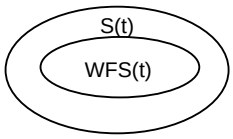


Figure 3.4: Possible $(S(t))$ and well-formed instantiations $(WFS(t))$ of a Z template (t).

FTL is a general language: it captures the form of sentences and makes no assumptions in terms of meaning from the target language. Meta-Proof, however, requires FTL to be linked with the target language, so that reasoning with template representations makes sense. So, meta-proof with Z considers only those templates that yield Z sentences and instances that are *well-formed*. In Z, *well-formed* means that the sentence is type-correct.

The set of all possible well-formed Z sentences resulting from the instantiation of an FTL template (see figure 3.4) can be defined formally. Given the syntactic and semantic definitions above, the set of all possible instances of a template is given by the function:

$$S == (\lambda\, t : E \bullet \{gi : GEnv \bullet \mathcal{M}_{\mathcal{E}}(t)\ gi\})$$

The function $\mathcal{ZTC}$ tells whether the given string is a type-correct Z specification:

$$\mathcal{ZTC}\_ : \mathbb{P}\ Str$$
$$\mathcal{ZTC}\ s \Leftrightarrow\ s\ is\ type\ correct$$

Then, the set of all well-formed Z instances of a template, a subset of all possible instances, is given by:

$$WFS == (\lambda\, t : E \bullet \{s : S(t) \mid \mathcal{ZTC}\ s\})$$

Essentially, this is how FTL is linked with Z for meta-proof: through the well-formed instances of Z templates. In this sense, a Z template denotes a set of Z specifications (figure 3.5), the set of all possible type-correct instances. This is the basis for meta-proof.

3.5.2 Characteristic instantiation and the rule of generalisation

In the process of reasoning with Z templates, there is a point where a switch from the world of templates into the world of Z occurs: a general template formula becomes an instance. This involves a *characteristic* instantiation. For example, in the template formula, $\varnothing \in \mathbb{P} \ll ID \gg$, it is clear that ID must hold a name referring to a set; the sentence would not be type-correct otherwise. So, here, a characteristic instantiation is required: let X be an arbitrary set, instantiate ID with X to give, $\varnothing \in \mathbb{P}\, X$, which is trivially true. The set X is a characteristic

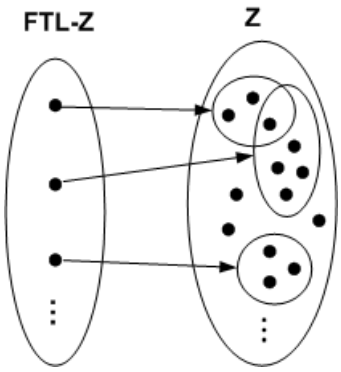


Figure 3.5: A template denotes a set of Z specifications, set $WFS(t)$ (an empty set of denotations is represented without an arrow).

instantiation of that formula. In this simple case, only one characteristic instantiation needs to be considered. Other cases require induction (for lists) or case analysis (for choice).

But, when is it safe to conclude the truth of the template statement from the proof of its instance? In formal logic there is a similar problem. Suppose a set A and a predicate P, the truth of the predicate logic statement, $\forall\ x : A \bullet P$, implies that it is true for every value in A. But how can such a statement be proved? We could prove that it is true for each value in A, but this is not practical because it may involve a large or even infinite number of proofs. This is solved by proving that P holds for an *arbitrary* member of A: if no assumptions about which member of A is chosen in order to prove P, then the proof generalises to all members. This is, in fact, a known inference rule of predicate logic called *generalisation* (or *universal introduction*) [WD96, Abr96]:

$$\frac{\Gamma \vdash \forall\, x : A \bullet P \quad [\forall\text{-I}]}{\Gamma;\ x \in A\ \vdash P \quad [x \notin FV(\Gamma)]}$$

This is the principle behind *characteristic* instantiation. An arbitrary instantiation of a template formula is introduced so that the proof of the instance generalises to the proof of the template.

We now apply the principle of characteristic instantiation to make some meta-proofs.

3.5.3 Illustration: meta-proof using the semantics

We can do meta-proof by using the semantics of FTL: (a) first, we define characteristic instantiation(s) of the templates involved; (b) then, templates and meta-conjectures are instantiated with characteristic instantiations using the semantic functions; and (c) the proof is performed in the Z domain using the generated instances.

The Z templates and meta-conjecture of the promoted ADT are:

$$\begin{aligned}
&\llparenthesis P \rrparenthesis == [\ \llparenthesis ids \rrparenthesis : \mathbb{P}\ \llparenthesis ID \rrparenthesis;\ \llparenthesis st \rrparenthesis : \llparenthesis ID \rrparenthesis \pfun \llparenthesis S \rrparenthesis \\
&\qquad\qquad \mid \mathrm{dom}\ \llparenthesis st \rrparenthesis = \llparenthesis ids \rrparenthesis \land \llparenthesis I \rrparenthesis\ ] \\
&\llparenthesis P \rrparenthesis Init == [\ \llparenthesis P \rrparenthesis\ ' \mid \llparenthesis ids \rrparenthesis' = \varnothing \land \llparenthesis st \rrparenthesis' = \varnothing\ ] \\
&\vdash ?\quad \exists\ \llparenthesis P \rrparenthesis Init \bullet \mathrm{true}
\end{aligned}$$

In well-formed instantiations of these templates, P holds a name not declared before in the specification; $\ll ids \gg$ and $\ll st \gg$ also hold names and these two names must be different; ID and S hold names referring to sets; $\ll I \gg$ holds a predicate. So, a characteristic instantiation is, $e == \{P \mapsto \text{“}P\text{”}, ids \mapsto \text{“}ids\text{”}, st \mapsto \text{“}st\text{”}, ID \mapsto \text{“}ID\text{”}, S \mapsto \text{“}S\text{”}, I \mapsto \text{“}I\text{”}\}$, where ID and S are arbitrary sets, and I is an arbitrary predicate.

This is used to define the global environment, $g == \langle\,\rangle \times \mathsf{tree}(e, \langle\,\rangle)$. The instantiation of the templates, using the defined semantics of the language, yields:

$$P == [\ ids : \mathbb{P}\ ID;\ st : ID \pfun S \mid \mathrm{dom}\ st = ids \land I\ ]$$
$$PInit == [\ P\ ' \mid ids' = \varnothing \land st' = \varnothing\ ]$$
$$\vdash ?\quad \exists\ PInit \bullet \mathrm{true}$$

The initialisation conjecture reduces to $I'[ids' := \varnothing, st' := \varnothing]$. And if the variable I in the template is substituted with *true*, then the conjecture is trivially true — given arbitrary sets ID and S, it is automatically proved in Z/Eves.

3.5.4 Illustration: meta-proof using the instantiation calculus

FTL's IC is designed to support meta-proof so that proofs at the template level could be done by transforming template formulas, where each transformation performs some instantiation: the application of the generalisation rule.

To make the proof with the IC, we build a sequent from the definitions and the Z meta-conjecture. This sequent is then transformed by using the IC, then cast into Z, and, finally, proved within the world of Z:

$$\ll P \gg == [\ \ll ids \gg : \mathbb{P}\ \ll ID \gg;\ \ll st \gg : \ll ID \gg \pfun \ll S \gg \mid \mathrm{dom}\ \ll st \gg = \ll ids \gg \land \ll I \gg\ ];$$
$$\ll P \gg Init == [\ \ll P \gg\ ' \mid \ll ids \gg' = \varnothing \land \ll st \gg' = \varnothing\ ]$$
$$\vdash\ \exists\ \ll P \gg Init \bullet \mathrm{true}$$

$$\equiv [\mathsf{apply}\ \mathcal{IP}_{\mathcal{E}}\ \mathsf{with}\ \{P \mapsto \text{“}P\text{”}, ids \mapsto \text{“}ids\text{”}, st \mapsto \text{“}st\text{”}, I \mapsto \text{“}I\text{”}\};\ I\ \mathsf{is\ arbitrary\ predicate}]$$
$$P == [\ ids : \mathbb{P}\ \ll ID \gg;\ st : \ll ID \gg \pfun \ll S \gg \mid \mathrm{dom}\ st = ids \land I\ ];$$
$$PInit == [\ P\ ' \mid ids' = \varnothing \land st' = \varnothing\ ]$$
$$\vdash\ \exists\ PInit \bullet \mathrm{true}$$

$$\equiv [\mathsf{apply}\ \mathcal{IP}_{\mathcal{E}}\ \mathsf{with}\ \{ID \mapsto \text{“}ID\text{”}, S \mapsto \text{“}S\text{”}\};\ ID\ \mathsf{and}\ S\ \mathsf{are\ arbitrary\ sets}]$$
$$P == [\ ids : \mathbb{P}\ ID;\ st : ID \pfun S \mid \mathrm{dom}\ st = ids \land I\ ];$$
$$PInit == [\ P\ ' \mid ids' = \varnothing \land st' = \varnothing\ ];\ ID \in \mathbb{U};\ S \in \mathbb{U}$$
$$\vdash\ \exists\ PInit \bullet \mathrm{true}$$

$$\equiv [\mathsf{By}\ \exists\ Sc\ \mathsf{(twice)\ and\ thin}]$$
$$ID \in \mathbb{U};\ S \in \mathbb{U}$$
$$\vdash\ \exists\ ids : \mathbb{P}\ ID;\ st : ID \pfun S \bullet ids' = \varnothing \land st' = \varnothing \land I'$$

$$\equiv [\mathsf{By\ one\text{-}point}]$$
$$ID \in \mathbb{U};\ S \in \mathbb{U}\ \vdash\ \mathrm{dom}\,\varnothing = \varnothing \land \varnothing \in \mathbb{P}\ ID \land \varnothing \in ID \pfun S \land I'[ids' := \varnothing, st' := \varnothing]$$

$$\equiv [\mathsf{By\ set\ theory\ and\ propositional\ calculus}]$$
$$I'[ids' := \varnothing, st' := \varnothing]$$

This in the template form is: $\ll I \gg'[ids' := \varnothing, st' := \varnothing]$. If the variable I is replaced by true then the conjecture reduces to true:

$$\mathrm{true}'[ids' := \varnothing, st' := \varnothing] \equiv \mathrm{true}$$

3.5.5 A logic for template-Z

The proof methods illustrated above manipulate template formulas by instantiating them. Only when everything has been instantiated can other types of inference be applied in proofs. The proofs above first instantiate the template formulas and then prove what results from the instantiation using the inference rules of the target language (in this case Z). This is done based on the notion of characteristic instantiation introduced above.

As can be observed from the proofs above, this may involve going from the world of templates to the world of the target language and then back to templates; the proofs above had to abstract the result of the inference back into the template form after the instantiation had been performed. Moreover, proofs can get substantially more complex in the presence of lists and choice because more than one instantiation case needs to be considered. A better approach would be to perform the proof in the world of templates and go to the target language only when strictly necessary. This, however, requires rules of inference for template formulas of some target language: a logic.

A logic comprises a set of axioms or basic rules and another set of rules which are derived from the axioms (called derived rules or theorems) [GS93]. This section proposes an approach to extend the logic of the target language so that it is possible to perform proof inferences (other than instantiation) with template formulas. This extension can be seen as a layer (a logic for template formulas) that builds on top of the existing logic of the target language. The challenge is to come up with a set of basic rules that is small and a template-logic that is sound (that is, only true theorems may be proved in the logic).[10]

This section shows how such a template logic could be built and proposes a draft template-Z logic, which extends the logic of Z. Note that a logic for Z is still a contentious issue among Z users because there is no agreed system of logic. The Z standard [ISO02b] defines the syntax and semantics for Z, but not a logical system. There is, however, a logic for Z because people have been doing proof in Z for years, logics for Z have been proposed in the literature (e.g. [WD96] proposes a logic for Z based on natural deduction) and there are theorem provers for Z (Z/Eves [Saa97] and Proofpower [Art] for instance), there is just not one sole formal system of logic that Z users agree on. To devise a template logic for Z, this thesis assumes a system of logic for Z that is based on the sequent calculus and well-known Z inference rules that are available in the literature; this is documented in appendix B.[11]

Template formulas over some language add one more layer of abstraction: the world of templates of some language. Above, we have seen that it is characteristic instantiation that allows us to switch from the more abstract world of templates to the target language in proofs. In fact, the basic rules of the template-Z logic proposed here perform *characteristic instantiations*. All other rules are derived from these basic rules and the existing inference rules for Z.

These basic rules (or the rules of characteristic instantiation) are, for now, left *unproved*. However, they would need to be proved for the logic to be claimed sound; this is left for future work. They could be proved by considering a meta-world of Z (Fig. 3.6), where its objects

[10] It is more difficult to achieve completeness (that is that every true theorem may be proved using the logic) because most modelling languages are incomplete (Z, for instance, is incomplete); this is a consequence of Gödel's incompleteness theorem (states that every logical system that axiomatises arithmetic cannot be both sound and complete [GS93]). A template logic will inherit the incompleteness of the target language.

[11] There are different frameworks for proof logics, such as natural deduction systems where proof is based on derivation trees and Gentzen systems where proof is based on the sequent calculus (proof inferences manipulate a structure called sequent) [TS00]. The proofs in this chapter are based on the sequent calculus.

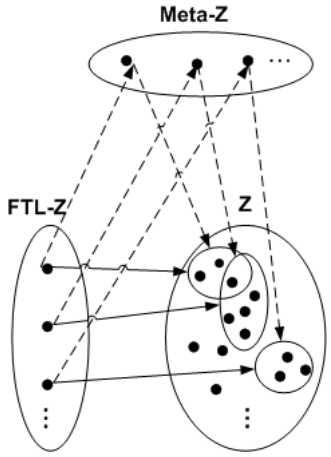


Figure 3.6: A Z template denotes one meta-Z object and a set of Z specifications. A meta-Z object denotes a set of Z specifications.

represent FTL templates of Z and denote sets of Z specifications. For example, the sequent,

$$\vdash \varnothing \in \mathbb{P}\llparenthesis ID \rrparenthesis$$

can be proved in meta-Z; suppose that in meta-Z there is a universal set $\mathbb{U}$, of which all sets in a Z specification are a subset of.[12] That formula is interpreted in meta-Z as:

$$\vdash \forall\, ID : \mathbb{P}\, \mathbb{U} \bullet \varnothing \in \mathbb{P}\ \ ID$$

Now, by the law of universal introduction,

$$ID \in \mathbb{P}\, \mathbb{U} \vdash \varnothing \in \mathbb{P}\ \ ID$$

which is trivially true.

The framework outlined here of proving each basic rule formally using the argument outlined above, and proving each derived rule by appeal to these basic rules and the rules of Z would ensure that the logic is sound. Typically, soundness proofs are based on an induction argument, involving a base case and an induction step. (Typically, the induction is based on the length of the proof or the number of proof steps that are required to conclude that a theorem is true.) The proof that the basic rules of characteristic instantiation are true (by appeal to the argument outlined above) constitutes the proof of the base case. The proof that each derived rule is true by appeal to basic rules and the rules of Z only, would constitute the proof of the induction step. Provided template proofs are based on the inference rules proved in this way no false theorems can be derived from the logic.

The following illustrates meta-proof with the template-Z logic, using the initialisation conjecture of the promoted ADT.

3.5.6 Proof with the template-Z logic

We now go back to the proof of the initialisation of the promoted ADT to illustrate proof with the template-Z logic.

[12] In fact, this is precisely the case in the ISO standard semantics of Z [ISO02b].

First, there is an inference rule for characteristic instantiation, which allows a variable to be replaced with a name referring to a set. This transformation uses the IC to replace all occurrences of a variable in a template formula with its substitution. For now, it is an unproved axiom of the template-Z logic:

$$\frac{\Gamma \vdash E_1 \lessdot S \gtrdot E_2}{S \in \mathbb{U};\ (\mathcal{IP}_{\mathcal{E}}(\Gamma \vdash E_1 \lessdot S \gtrdot E_2)\{S \mapsto \text{“}S\text{”}\})} \quad \begin{array}{l}[\text{T-I-PS}] \\ [S \notin FV(\Gamma)]\end{array}$$

An inference rule for Z templates for the schema calculus inference rule $\exists\ Sc$ (appendix B), is also required:[13]

$$\frac{\Gamma \vdash \exists\ [\ \lessdot ScD \gtrdot \mid \lessdot ScP \gtrdot\ ] \bullet \lessdot P \gtrdot}{\Gamma \vdash \exists\ \lessdot ScD \gtrdot \bullet \lessdot P \gtrdot \wedge \lessdot ScP \gtrdot} \quad [\text{T}\ \exists\ Sc]$$

We also need the one-point rule (appendix B) for templates:

$$\frac{\Gamma \vdash \exists\ \lessdot x \gtrdot : \lessdot t \gtrdot \bullet \lessdot P \gtrdot \wedge \lessdot x \gtrdot = \lessdot v \gtrdot}{\Gamma \vdash \lessdot P \gtrdot [\ \lessdot x \gtrdot := \lessdot v \gtrdot\ ] \wedge \lessdot v \gtrdot \in \lessdot t \gtrdot} \quad \begin{array}{l}[\text{T one-point}] \\ [\lessdot x \gtrdot \notin FV(\lessdot v \gtrdot)]\end{array}$$

These two inference rules are easily proved by using the axiom rules of the template-Z logic and the logic of Z (see appendix B).

The meta-proof of, $\vdash ?\ \exists\ \lessdot P \gtrdot Init \bullet \text{true}$, using these rules is:

$\equiv$ [by defn of $PInit$]
$\lessdot P \gtrdot == [\ \lessdot ids \gtrdot : \mathbb{P} \lessdot ID \gtrdot;\ \lessdot st \gtrdot : \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \mid$
$\quad \text{dom} \lessdot st \gtrdot = \lessdot ids \gtrdot \wedge \lessdot I \gtrdot\ ];$
$\lessdot P \gtrdot Init == [\ \lessdot P \gtrdot\ ' \mid \lessdot ids \gtrdot' = \varnothing \wedge \lessdot st \gtrdot' = \varnothing\ ]$
$\vdash \exists\ [\ \lessdot P \gtrdot\ ' \mid \lessdot ids \gtrdot' = \varnothing \wedge \lessdot st \gtrdot' = \varnothing\ ] \bullet \text{true}$
$\equiv$ [by $T\ \exists\ Sc$; sequent calculus]
$\lessdot P \gtrdot == [\ \lessdot ids \gtrdot : \mathbb{P} \lessdot ID \gtrdot;\ \lessdot st \gtrdot : \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \mid$
$\quad \text{dom} \lessdot st \gtrdot = \lessdot ids \gtrdot \wedge \lessdot I \gtrdot\ ]$
$\vdash \exists\ \lessdot P \gtrdot\ ' \bullet \lessdot ids \gtrdot' = \varnothing \wedge \lessdot st \gtrdot' = \varnothing \wedge \text{true}$
$\equiv$ [by $T\ \exists\ Sc$; sequent calculus; propositional calculus]
$\vdash \exists\ \lessdot ids \gtrdot' : \mathbb{P}\ \lessdot ID \gtrdot;\ \lessdot st \gtrdot' : \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \bullet$
$\quad \lessdot ids \gtrdot' = \varnothing \wedge \lessdot st \gtrdot' = \varnothing \wedge \text{dom}\ \lessdot st \gtrdot' = \lessdot ids \gtrdot' \wedge \lessdot I \gtrdot$
$\equiv$ [by $T\ one - point$]
$\vdash \varnothing \in \mathbb{P}\ \lessdot ID \gtrdot \wedge \varnothing \in \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \wedge \text{dom}\ \varnothing = \varnothing$
$\quad \wedge \lessdot I \gtrdot'[\lessdot ids \gtrdot' := \varnothing, \lessdot st \gtrdot' := \varnothing]$
$\equiv$ [by T-I-PS twice and set theory]
$ID \in \mathbb{U};\ S \in \mathbb{U}$
$\vdash \varnothing \in \mathbb{P}\ ID \wedge \varnothing \in ID \pfun S \wedge \lessdot I \gtrdot'[\lessdot ids \gtrdot' := \varnothing, \lessdot st \gtrdot' := \varnothing]$
$\equiv$ [by set theory and propositional calculus]
$\lessdot I \gtrdot'[\lessdot ids \gtrdot' := \varnothing, \lessdot st \gtrdot' := \varnothing]$

And if I is instantiated with *true*, then the formula reduces to true.

Note that in proof with templates placeholders may be substituted by other template formulas containing themselves placeholders. This is the case in the second and third steps of the proof above.

[13] Template inference rule names are preceded by T.

3.6 Discussion

The Z language supports generic structures, Z generics, but these are not a substitute for templates. Z generics are too restricted, they only allow parameterisation of Z constructs with sets (a parameter can be substituted by a set and set only). This restriction makes it impossible, for instance, to represent the template above with a Z generic.[14]

FTL was defined formally with the aim of mechanical meta-proof. However, this helped to clarify the language, to gain a better understanding of what templates are, what can be done with them and of meta-proof, and makes the construction of an FTL tool an extension of the work done so far.

It is interesting to compare FTL's IC with other calculus used in software engineering. There are some similarities with Dijkstra's calculus of weakest preconditions [Dij75], which calculates preconditions from program text (a collections of sentences in a programming language expressing transformations of state in a representation of some machine). The FTL semantics and IC calculate instances of templates given a total or partial instantiation. The IC is closer to the refinement calculus [Mor94], where a specification is refined by applying a sequence of design decisions until a program is obtained; this, usually, involves resolving non-determinism. In IC templates are transformed until a formula in some language is reached; this involves resolving the instantiation of parameters, lists, and choices. In fact, both the semantics and the IC give the meaning of FTL templates. This is also not new, the *B* specification language has its semantics based on a calculus based on weakest preconditions [Abr96].

Testing the FTL semantics and IC with the Z/Eves prover helped to uncover many errors. Placeholders are simple and easy to get right, but more complex constructs (such as lists) required a lot more testing to get their definitions right. The consistency between the calculus and the semantics was tested, and a full proof of consistency was not yet done at this stage. In fact, both the IC and the actual semantics give the semantics of FTL: the actual semantics does it in terms of total instantiations, whereas the IC does it in terms of partial instantiations.

As said above, inference rules related with *characteristic instantiation* are, for now, unproved in the template-Z logic, which would have to be proved for the logic to be claimed sound. We believe that this is proved by applications of the generalisation rule in a proper meta-world of Z (as discussed in section 3.5). The proof of these basic rules by appeal to meta-Z argument (see above) together with the proof of all derived rules of template-Z by appeal to those basic rules and the rules of Z would demonstrate that the template-Z logic is sound.

As the example shows, templates may need to be constrained so that useful results can be extracted. So, template design needs to consider what is to be described, and the results to be proved. The most attractive aspect of meta-proof is the property *true by construction.* Often, however, meta-proof gives a simplification of the original conjecture, which, in many cases, is sufficient to allow the prover to discharge the remaining formula automatically. Meta-Theorems capture our experience in proving theorems with instances of some structure; they formalise something done and perceived in practice.

Our approach tries to address several requirements. We want to capture the form of sentences of any language, trying to separate form from content, and use templates for reasoning. So, FTL has a very general semantics, and requires further work to be integrated with some

[14] [VTSK00] proposes *type-constrained* generics to soften the restrictions of normal Z generics, but even type-constrained generics are not as expressible as FTL templates.

target language for reasoning. There is a separation of concerns: on one side the language to describe form, on the other the approach to reason with those representations. This constitutes a pragmatic and non-intrusive approach, rather than extending a language to support templates, we designed a general language to capture form.

Formal development requires expertise in the use of proof tools. Our approach allows experts to build templates and prove meta-theorems (perhaps assisted by proof tools), so that software developers who are not experts in formal-methods can still build formal models that are proved consistent.

FTL evolved from an approach to syntactically decorate Z into a formal language. Early in the development of the object-oriented structuring (or style) for Z (next chapters), there was a need to represent commonly occurring structures of Z specifications built using that style. Templates were introduced in order to represent these commonly occurring structures that could not be represented in Z itself. Later, it became clear that some reasoning could be performed on template representations of Z. This was the motivation for both FTL and meta-proof. This evolution is similar, in many ways, to the evolution of the schema construct of the Z specification language [Woo89]. In the early stages of the development of Z, schemas were used informally as textual macros for pieces of mathematical text. Like macros, schemas could be expanded by replacing the name of the schema by its body. Only later did they evolve to become the main structural construct of the language, representing values in its own right and used to model state spaces and operations.

3.7 Related Work

Catalysis [DW98] proposes templates and hints at variable substitution for instantiation, but this is defined informally. Moreover, its template notation has fewer features than FTL (Catalysis has only placeholders; FTL has lists and choice).

As discussed in the previous chapter (page 43), there has been work on formalising patterns. There have been formalisations of OO patterns in several formal modelling languages, but FTL's treatment of patterns is more general than OO. The patterns of temporal logic [DvL96, DAC99], similar to *schematic* representations in logic, are closer to FTL. This is, however, less flexible than FTL, having fewer abstraction constructs (placeholders only), and mechanisation is not addressed.

The approach behind FTL and the IC is akin to term-rewriting systems [BKdV03], which are methods for replacing sub-terms of a formula with other terms based on rewriting rules. In our approach, the FTL semantics and IC define rules for the substitution of placeholders, lists and choice. Term-rewriting is used to capture the form of objects; *L-Systems* [PL90], for instance, is a string rewriting system designed to capture the form of plant growth, which has many application in computer graphics. FTL captures the form of formal language sentences.

The term *template* is sometimes used to refer to generics (e.g. C++ templates), which allow parameterisation based on types, and may involve subtyping and polymorphism. Generics are more restricted than FTL templates.

3.8 Conclusions

This chapter presents FTL, a language to express templates, a calculus of instantiations for FTL templates and an approach that explores templates of formal models for proof. FTL

allows the representation of structural patterns of modelling and proof and their mechanical instantiation. This enables reuse in formal development, which contributes to reducing the effort involved in formal modelling and verification.

The next two chapters use FTL and the meta-proof approach for Z to develop a catalogue of templates and meta-theorems for the $UML + Z$ framework. Those chapters show how FTL can be used to represent templates and generate Z models, and how meta-proof can be used to reduce the proof overhead associated with Z development.

At the moment when a person is faced with an act of design, he does not have time to think it from scratch.
[...] Even when a person seems to "go back to the basic problem," he is still always combining patterns that are already in his mind.

Christopher Alexander [Ale79]

4

ZOO: an Object-Oriented style for Z

The previous chapter developed an important tool to build GeFoRME frameworks: the language FTL and the meta-proof approach. This chapter starts using these tools to build a specific GeFoRME framework: $UML+Z$. This chapter does two things: (a) it develops ZOO, an object-oriented style for Z, which is the semantic domain of the $UML+Z$ framework; and (b) it shows how the catalogue of templates of $UML+Z$ can be used to generate ZOO models.

ZOO is a style of specification in the Z language. Z is flexible, simple, and has a mathematical rather than a computational semantics, which makes it adaptable to different computational paradigms. This chapter builds ZOO by adapting Z to the object-oriented (OO) computational paradigm. ZOO enables the expression of OO models in Z, and is used to give a Z meaning to UML diagrams of $UML+Z$.

The $UML+Z$ catalogue of FTL templates explores the fact that different ZOO models have many structures in common. ZOO may be used to build different OO models across different application domains, but all these models have the same underlying structures — they are *patterns*. FTL is used to represent these common patterns of ZOO models, which are assembled in the templates catalogue of $UML+Z$. This catalogue allows the generation of ZOO models by instantiating FTL templates. Some templates also include their meta-theorems, which are used to simplify (in some cases fully prove) conjectures of a ZOO model.

This chapter develops the core of ZOO and illustrates it in the context of $UML+Z$. This core includes foundational structures of OO, namely: classes, associations and systems. The next chapter extends this core with class inheritance (or *is-a* relationships).

The following starts by explaining core concepts of OO. Then, it explains the concept of Z *views*, which is used in the ZOO style. Next, it overviews ZOO by seeing how OO concepts are represented in Z, explains the approach to consistency-check ZOO models, and discusses some issues with the $UML+Z$ catalogue. Then, the ZOO style is illustrated in the context of $UML+Z$: a ZOO model of a trivial bank system is built by instantiating templates from the $UML+Z$ catalogue. Finally, the chapter discusses some issues regarding the design of the ZOO style and related work.

4.1 Core concepts of the OO paradigm

The OO paradigm is a model of computation centred around the notion of *object*. Like many concepts in computing, the object is related with the idea of *abstraction*.

Abstraction is a cognitive tool that is used to reduce the information content of a concept, in order to retain information that is relevant for a particular purpose. Hoare said that "abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on those similarities, and to ignore for the time being the differences [Hoa72a]."

An object is an instance of some abstraction, representing some object from the real world or some virtual domain. Objects are *individuals*. They are characterised by an *identity* (which distinguishes one object from all others) and observable properties. OO computing revolves around computer representations of objects, their relations and interactions.

As Jackson puts it "An abstraction is a structure, pure and simple — an idea reduced to its essential form [Jac06]." In the OO paradigm, these structures are classes, whose instances are objects. This chapter explores other structures that are useful in OO computations: *associations* and *systems*.

Class. A class is a representation of an abstraction. It defines a set of objects with common properties. For example, consider the class Pen, an abstraction of all pen objects from the real world; one particular object of this class may be the red-ink pen that is in my desk. In computing, an OO class embodies several concepts and ideas that evolved over the years: a class is both a *module* and an *abstract data-type*.

A module is a basic unit of system decomposition. It is based on the idea that a complex system should be decomposed into a number of self-contained modules that can be fitted together to make the whole. Modules should follow the principle of *information hiding* [Par72], which says that a module should hide as much internal details as possible from all others in order to minimise the impact of changes. An abstraction is said to be *encapsulated* when its internal details are hidden.

The abstract data-type (ADT) is a structure that defines a class of abstract objects, which are completely characterised by the operations available on them [LZ74]. This realises the idea of information-hiding, and *data abstraction*, which separates the abstract properties of some abstraction from its representation details (data).

So, a class is a module whose objects are defined by its ADT. Unlike objects of general ADTs, class objects are individuals: they have an identity, which distinguishes one object from all others. Objects of some class have their internal details hidden (they are *encapsulated*), and all that is made available to the outside world is a set of operations. Encapsulation and abstraction (which underlie the OO model of computing) are concepts that we are accustomed to in our everyday lives. To make a car move, we do not need to understand the details of mechanics; all we need to know, as drivers of cars, is that by pressing the accelerator pedal the car moves faster and that by pressing the brake pedal the car slows down. Cars, phones, watches, televisions, cash machines and computers are all abstractions that we learn how to work with, but that we do not necessarily know in detail how they actually work.

Association. The abstraction of the real world and the classes of any interesting system, are not isolated. There are several kinds of such relationships, this chapter explores *associations*, which are *has-a* relationships (*is-a* relationships are explored in the next chapter) and express

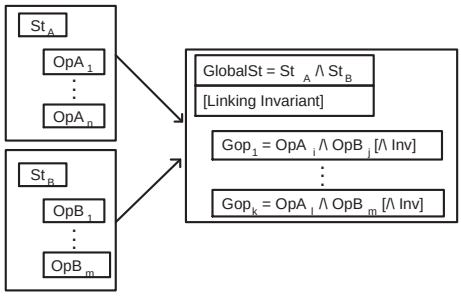


Figure 4.1: Z views and their composition using Z schema conjunction.

some semantic relation between otherwise unrelated classes. For example, a class Person is associated with a class Car through the association Has: Person Has Car.

System. A class constitutes a module (or component). In ZOO, associations are also components. We need a way to put these local components together, to bring about global system behaviour. This is done in the system structure.

4.2 Views and views structuring in Z

To articulate all properties of an OO system, ZOO is structured with *views*. A view is a partial model; it describes one particular aspect of a system. The whole model of the system is obtained by composing the different views. Views-based structuring is an approach to separate concerns, an application of the well-known principle of *divide and conquer*.

ZOO uses the views structuring approach for Z proposed by Jackson [Jac95a]. In this approach, a view is a partial specification of a program and includes a definition of a state space and a set of operations (a view is a Z ADT). A specification is made of a set of disjoint views, which are combined into composite structures through their states and their operations using the operators of the schema calculus (usually, schema conjunction). The composition of views usually involves linking invariants, which state the constraints that exist between the views being linked and that may be stated either at the level of state or operations. Figure 4.1 sketches the technique. First, the various aspects of the system are modelled in disjoint views, each consisting of one Z abstract data type (ADT) (St_A and St_B in figure 4.1). Then, these independent ADTs are composed to make another ADT (*GlobalSt* in figure 4.1).

Views-structuring was adopted in ZOO to separate concerns. The concept was useful in structuring ZOO because not all OO properties would fit into a single representation. For example, ZOO has three different representations of the class concept, atoms, intension, extension, which are represented independently in different views (see below).

4.3 The structure of the ZOO style

The *object* is the central concept in the OO paradigm. Objects are characterised by an *identity* (which distinguishes one object from all others) and observable properties [CD94]. A class is an abstraction representing a set of objects with a common structure and behaviour.

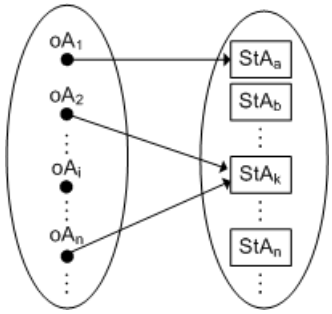


Figure 4.2: The set of all object atoms of class A, OA, the set of all possible object states StA (class intension), and the mapping from existing objects to their current states (class extension).

The building blocks of ZOO are object-based structures: namely, *class*, which represents a set of objects, *association*, which represents relations between classes of objects, and *system*, which represents the ensemble that includes classes and their associations. These structures are represented separately and independently as Z ADTs, composed of a state, initialisation, operations and a finalisation.

4.3.1 Object and class

The class structure is further divided in two, based on the dual meaning of a class. The class *intension* defines a class in terms of the properties shared by the objects of that class (for example, a class Person with properties *name* and *address*). The class *extension* defines a class in terms of the currently existing objects (instances) of the class (for example, Person is $\{MrSmith, MrAnderson, MsFitzgerald\}$). This duality is inspired by the definitions of a set in set theory [Hal60]. In ZOO, each class has one intensional and one extensional representation.

In ZOO's models, figure 4.2, objects are atoms (individuals represented in Z as elements of a given set). The class intension defines the set of all possible object states. A function maps the existing object atoms to their current states (the class extension). The representation of objects as atoms ensures that each object has a unique identity.

Class intension and extension are represented in separate views. The intension specifies the state space of objects and operations upon the state from the point of view of individual objects. The extension is specified using Z promotion [WD96, SPT03a] (the intension is promoted), where the individuals are put together to make a group of individuals. This ensures separation of concerns: objects, class intensions, and class extensions are separately represented.

4.3.2 Association

An association denotes a set of object tuples, where each tuple describes the objects being related (or linked). In the example above, (*MrSmith*, *AFerrari*) could be a tuple of Has, recording the fact that *MrSmith* (a Person) has a *Ferrari* (a Car).

ZOO represents associations as a Z relation between object atoms (figure 4.3). The operations manipulate the set of tuples of the associations (e.g., add tuple, remove tuple). An alternative representation of associations is discussed below (section 4.10).

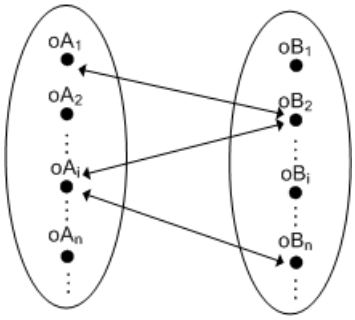


Figure 4.3: The sets of all object atoms of classes A and B, and the tuples of the association.

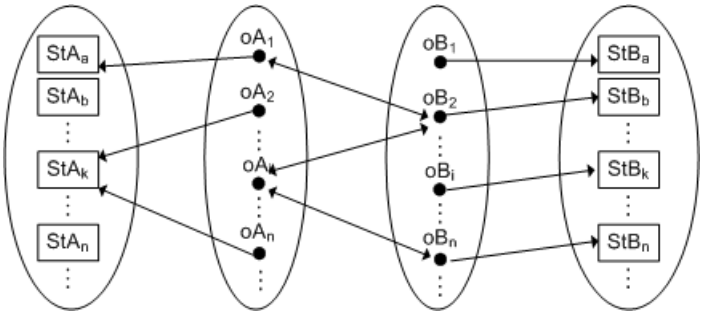


Figure 4.4: A system composed of classes A and B, and one association between them.

4.3.3 System

So far, we have seen classes and how to relate them using associations. We also need a way to make ensembles of classes and associations to bring about global behaviour (that makes sense only in terms of the ensemble). For example, consider the classes Person, Car, and the association Has relating them; there may be operations in Person and Car to add new Person and Car objects to the system, and an operation in Has to add tuples. But how to model an operation of the system to register the property of cars, that is, the creation of a Car object and its link with an existing or new Person in the association Has? This cannot possibly be expressed locally (in the scope of Car, Person or Has), so they need to be composed into an ensemble so that this operation can be expressed.

In ZOO, the system structure defines such ensembles. It is defined as a composition of classes and associations (figure 4.4), and includes global system constraints. System operations are also defined as a composition of component operations.

ZOO's models have at least one system structure, representing the system as a whole. Smaller systems may have just one system structure, composed of the system's classes and associations. But in larger systems, classes and associations may be composed into subsystems, which are then composed to make the system.

4.3.4 The views of the ZOO style

The views of ZOO are closely related to its structures (or building blocks). So, we have a *class* view for classes, a *relational* view for associations and a *global* view for systems (or ensembles). In addition, there is a *structural* view, which defines a structure to allow classes, associations and systems to be built; this view defines sets of object atoms, and captures properties of class structures from an atom perspective of classes. The class view is further divided in two views: the *intensional* and the *extensional* views. As said above, the structures of the ZOO style are represented in Z as ADTs.

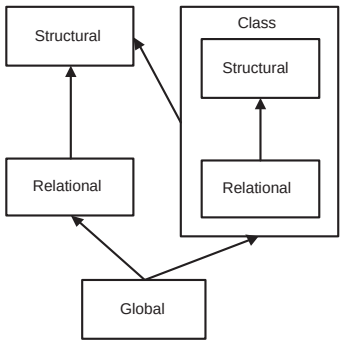


Figure 4.5: The views of ZOO and dependency relationships (arrow means dependency).

Figure 4.5 presents ZOO's views and the dependencies among them. The structural view is the only one that does not follow a Z state and operations style; it introduces global names, allowing classes and associations to be built independently whilst sharing the same vocabulary, and then to be linked in the global view. The remaining views are collections of ADTs.

ZOO uses several mechanisms to link views. The structural view defines global names, which are then used (directly or as a Z *parent* section) in the descriptions of the extensional, relational and global views. The extensional and intensional views are linked using Z promotion [WD96, SPT03a]: a class extension includes a collection of state intensions (as a mapping from object atom to state), the intensional operations are then promoted to be applicable to all the objects of the class. Finally, the link between the description of the global view and the ones from the relational and extensional views is established through operators of the Z schema calculus.

4.4 Consistency Checking in ZOO

Z models need to be checked for consistency, because models that are inconsistent cannot possibly have implementations that satisfy them. A ZOO model is consistent if all its components (or structures) are consistent. As said above, the components of a ZOO model are Z ADTs. So, checking component consistency amounts to a demonstration that the ADT is consistent using the usual Z approach.

In Z, an ADT is consistent if the state space, initialisation, operations and finalisation are consistent. The state space of a Z ADT is consistent if at least one instance of the state space does exist. In Z, this is proved by showing that the initial state of the ADT (defined in the initialisation) is a valid instance of the state space, which proves that both the state space and the initialisation are consistent. This proof obligation is expressed in Z as the conjecture:

$$\vdash? \ \exists \ StateInit \bullet \text{true}$$

where *StateInit* is the initialisation (see p.88 for an example).

The same approach is used for finalisation. So, a finalisation is consistent if it describes a valid instance of the state space, which is expressed as the conjecture:

$$\vdash? \ \exists \ StateFin \bullet \text{true}$$

where *StateFin* is the finalisation (see p.95 for an example).

The consistency of operations is checked by analysing their preconditions. The precondition of an operation describes the sets of states for which the outcome of the operation is properly defined. So, an operation is consistent if the precondition describes at least one state (the operation is satisfiable). In Z, this is demonstrated by proving the conjecture,

$$\vdash ?\ \exists \operatorname{pre} Op \bullet \text{true}$$

where *Op* is the operation specification (see p.93 for an example).

4.5 The *UML+Z* template catalogue

The *UML + Z* templates catalogue is given in appendix D. The templates of the catalogue use generics from the ZOO toolkit (appendix C). Every sentence of a ZOO model is generated by instantiating one of the templates of the catalogue. Associated with templates are meta-theorems, which simplify consistency conjectures.

Templates capture patterns, and a catalogue of templates is a repository of experience. The templates of the *UML + Z* catalogue emerged from sample ZOO models: recurring structures were identified, captured with templates and added to the catalogue. The catalogue then evolves as experience in building ZOO models increases: more templates are added to the catalogue, or existing templates are updated.

The catalogue is structured according to the views of the ZOO style and so there are templates for the structural (section D.1), class (section D.2), relational (section D.3) and global (section D.4) views. The class view is itself divided into intensional (section D.2.1) and extensional (section D.2.2) views. To generate a ZOO model from a UML model involves instantiating templates for each of these views.

To get stronger meta-theorems, some consistency conjectures are split in two: a *well-formedness* conjecture, and the usual consistency conjecture. The meta-theorems then draw on the truth of the *well-formedness* conjecture to get stronger results for the consistency conjecture. In most cases, well-formedness conjectures are proved automatically in the Z/Eves prover [Saa97].

The well-formedness conjecture is a consequence of the typing rules of Z, which are defined in terms of maximal sets. For example, consider the state space and initialisation:

$$St == [\ x : \mathbb{N} \mid x \bmod 2 = 0\ ]$$
$$StInit == [\ St\ ' \mid x' = -2\ ]$$

This is type-correct because the maximal type of $\mathbb{N}$ is $\mathbb{Z}$, but the initialisation is not consistent because -2 is not a natural number. Such errors are captured when proving consistency conjectures. In the proof of the initialisation conjecture for the definitions above, after applying the one-point rule (see appendix B.4), we are left with,

$$\vdash -2 \in \mathbb{N} \land -2 \bmod 2 = 0$$

which is *false*. The one-point rule always gives a conjunction formed by a substitution predicate and a membership predicate. By introducing the well-formedness conjecture we are splitting this conjunction in two: well-formedness captures the membership predicate, the consistency conjecture captures the substitution. So that if the well-formedness conjecture is

proved then its proof is not required in the consistency conjecture. In this setting, to prove the initialisation conjecture we would be required to prove first that, $\vdash -2 \in \mathbb{N}$ (well-formedness), and then $\vdash -2 \bmod 2 = 0$ (consistency). This separation is useful to define meta-theorems, because, in many cases, the proof of consistency is reduced to a trivial well-formedness proof.

The templates of the *UML + Z* catalogue generate Z that follows its ISO standard definition [ISO02b]. There is, however, one exception. This thesis borrows a notation from Z/Eves [Saa97] to refer to the substitution of variables by values in a Z expression. In the following and throughout, $E[x := a]$ refers to the substitution of the variable x by the value a in the Z expression E.

4.6 Generation of ZOO models

The following section generates ZOO models for a case study using the templates of appendix D, where templates are instantiated with data coming from diagrams or explicitly provided by the user. Some terminology needs to be introduced at this point. The following is used to refer to the Z that is generated from templates:

- *fully generated* — the Z is fully generated from templates with instantiation information coming from diagrams. If we had a tool, the Z would be automatically generated.
- *partially generated* — the instantiation depends on information not coming from UML diagrams, which needs to be explicitly added by the user.

And the following is used to refer to the results of proving consistency conjectures:

- *true by construction* — proved as a meta-theorem of the ZOO catalogue;
- *trivially true* — the Z/Eves prover can do it automatically with our library of Z/Eves laws, which extends the laws for the Z toolkit and includes laws for the ZOO toolkit;
- *provable* — can be proved but Z/Eves requires the user's intervention.

The *fully generated* and the *true by construction* are very closely related because more fully generated Z gives more proof for free. This is because the more it can be predicted and represented as templates the more it can be explored with proof at the meta-level (before actual Z models are generated). This becomes clearer in the following section.

The templates catalogue of appendix D comprises many templates, which are designed to represent several OO concepts under a variety of circumstances. Each template description briefly explains the context in which the template is to be used. Some of these templates are *fully generative*, others require user intervention to yield a Z description (partially generative) and other templates may be *fully or partially generative* depending on the context. Table 4.1 comments on this full vs partial generation of Z for each view of the ZOO style.

4.7 The Bank case study

The following discusses the problem (the Bank case study) and presents some UML diagrams, then formalisation into ZOO is discussed.

View	Generation of templates
Structural	All templates are *fully generated*.
Class, intensional view	If the class has no extra constraints (other than those expressed in diagrams), then state and initialisation are *fully generated*; otherwise *partially generated*. Operations are partially generated
Class, extensional view	Most definition are *fully generated*; only those operations that act on the set objects as a whole are *partially generated*.
Associations (relational view)	All *fully generated*.
System (global view)	State and initialisation is *fully generated* provided there are no global constraints; otherwise it is partially generated. Operations are partially generated.

Table 4.1: *Full* and *partial* generation in the templates of each view of the ZOO style.

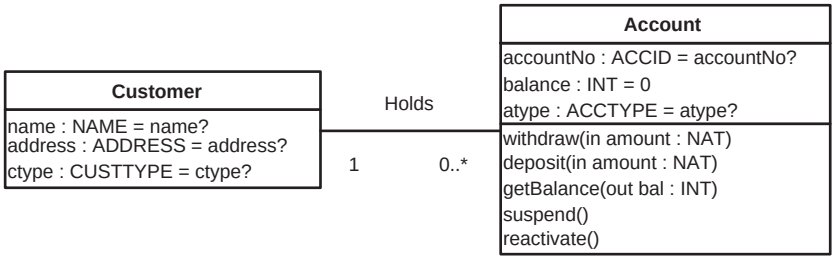


Figure 4.6: The UML class diagram of the trivial Bank system.

4.7.1 Problem and UML diagrams

The ZOO style is illustrated in the context of *UML + Z* with the case study of a trivial Bank system. The UML class diagram of figure 4.6 captures the static structure of this system.

- Customer represents the bank's customers; the attributes record the *name* of a customer, its *address*, and *type* (either *company* or *personal*).
- Account represents the accounts managed by the bank; the attributes record the account number (*accountNo*), the *balance*, and the *type* of account (either *current* or *savings*).
- The association Holds relates Customers and their Accounts; a Customer may have zero or more accounts; an Account must have one customer.

The global operations of the trivial Bank system are given in table 4.2. At the level of classes, this requires in the class Account the operations withdraw, deposit, getBalance, suspend and reactivate (see figure 4.6).

The constraints of the system are documented in table 4.3. As we can deduce from the operations and constraints, the objects of Account have distinct states, depending on whether the account is suspended or not (OP9, OP10 and C4); furthermore the operation *suspend* (OP9) brings accounts into a *suspended* state, and the operation *reactivate* (OP10)

OP1	*New Customer*	Create a new bank customer in the system.
OP2	*Open Account*	Open a new account for an existing bank customer.
OP3	*Deposit*	Deposit money into one account.
OP4	*Withdraw*	Withdraw money from one account.
OP5	*Get Balance*	Get the balance of one account.
OP6	*Get Customer Accounts*	Get all the accounts of a certain bank customer.
OP7	*Get Accounts in Debt*	Get all the accounts that are in debt.
OP8	*Close Account*	Delete one account from the system.
OP9	*Suspend Account*	Suspend an account, disabling account withdrawals.
OP10	*Reactivate Account*	Re-activate an account, re-enabling account withdrawals.

Table 4.2: The global operations of the trivial bank system.

C1	Savings accounts cannot have negative balances.
C2	The total balance of all the bank's accounts must not be negative.
C3	Customers of type *company* cannot hold savings accounts.
C4	Account withdrawals are not allowed when an account is suspended.
C5	A customer may close an account provided its balance is zero.
C6	To open a savings account, the customer must already hold a current account with the bank.

Table 4.3: The constraints of the trivial bank system.

brings accounts back into an *active* state, withdrawals are not allowed when the account is *suspended* (constraint C4). This behaviour of Account objects is described in the statechart of figure 4.7. The statechart also captures constraint C5; that is, account objects may be deleted (when the account is closed) provided the balance is 0, which is expressed in the diagram as a precondition of the transitions into the final state.

The diagrams of figures 4.6 and 4.7 describe the static structure of the system, some of its behaviour and constraints. The ZOO model that is *fully generated* from templates of appendix D with data coming from these diagrams is given appendix F.1; all consistency conjectures associated with this model are *true by construction*. The section that follows builds the full ZOO model (only illustrative components are given), which includes what is represented with diagrams and what is represent in ZOO but that cannot not be represented with diagrams.

4.8 ZOO model: state space

The state space is constructed from the five views. ZOO uses systematic naming conventions, based on [BSC94, chap.8]. A name starting with the letter $\mathbb{S}$ (stands for *set of objects*, from Hall [Hal90b]) designates a concept from the extensional view, and a name starting with the letter $\mathbb{A}$ (stands for *association*) designates a concept from the relational view.

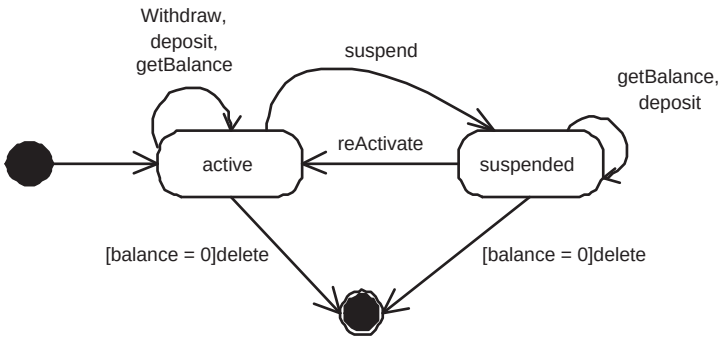


Figure 4.7: The UML statechart of the Account class in the trivial Bank system.

4.8.1 Structural View

The definitions of this view are *fully generated*. First, there is a section for the new model, which is defined by instantiating template T1:

section $Bank\_model$ parents $ZOO\_toolkit$

Also from template T1, is the definition of the set of all classes of a model as Z free type:

$$CLASS ::= CustomerCl \mid AccountCl$$

This defines a set of class atoms.

The following is discussed in detail in the next chapter, it is related with inheritance. For now, we are saying that there are no inheritance relations in our model.

$$\begin{array}{|l} subCl : CLASS \leftrightarrow CLASS \\ abstractCl : \mathbb{P}\ CLASS \\ rootCl : \mathbb{P}\ CLASS \\ \hline subCl = \{\} \\ abstractCl = \{\} \\ rootCl = CLASS \setminus \mathrm{dom}\ subCl \end{array}$$

Each class has a set of object atoms. The ZOO toolkit (appendix C) defines a type for all possible object atoms of any class, OBJ, as a given set. The set of objects of each class is a subset of OBJ, and to extract these sets there are the functions $\mathbb{O}_x$ and $\mathbb{O}$ that take a class atom and return subsets of OBJ; these functions are generated from template T2:

$$\begin{array}{|l} \mathbb{O}_x : CLASS \rightarrow \mathbb{P}_1\ OBJ \\ \mathbb{O} : CLASS \rightarrow \mathbb{P}_1\ OBJ \\ \hline \mathrm{disjoint}\ \mathbb{O}_x \\ \forall\, cl : CLASS \bullet \mathbb{O}\ cl = \mathbb{O}_x\ cl \cup \bigcup(\mathbb{O}_x(\!|\ (subCl^+)^{\sim}(\!|\ \{cl\}\ |\!)\ |\!)) \\ \forall\, cl, cl' : CLASS \mid cl \mapsto cl' \in subCl \bullet \mathbb{O}\ cl \subseteq \mathbb{O}\ cl' \end{array}$$

$\mathbb{O}_x$ gives all possible direct objects of a class, hence, it excludes those objects of its subclasses. $\mathbb{O}$ gives all possible objects of a class, which includes those objects of its subclasses. These

functions are total because each class has sets of possible objects, and they give non-empty sets. Each class has its own set of direct objects, which is non-empty. These functions are discussed in more detail in the next chapter. For now, the set of objects of a class, say Customer, can be obtained with, $\mathbb{O}_x$ *CustomerCl*.

4.8.2 Class View

This view defines the classes of the system. The following defines the classes Customer and Account of the trivial bank system.

Customer class, intensional view

The intension defines the state space of the objects of the class. That is, the state attributes of the class and, where required, an invariant.

The attribute types required to define Customer are *fully generated* from template T3; *NAME* and *ADDRESS* are defined as given sets, *CUSTTYPE* as a free type:

$$[NAME, ADDRESS]$$

$$NAME \neq \varnothing \land ADDRESS \neq \varnothing$$

$$CUSTTYPE ::= company \mid personal$$

The intensional state space and initialisation of Customer with consistency conjectures is *fully generated* from template T5:

Customer

$$name : NAME$$
$$address : ADDRESS$$
$$ctype : CUSTTYPE$$
---
true

CustomerInit

$$Customer'$$
$$name? : NAME$$
$$address? : ADDRESS$$
$$ctype? : CUSTTYPE$$

$$name' = name?$$
$$address' = address?$$
$$ctype' = ctype?$$

$$\vdash? \ \forall name? : NAME; \ address? : ADDRESS; \ ctype? : CUSTTYPE \bullet$$
$$name? \in NAME \land address? \in ADDRESS \land ctype? \in CUSTTYPE$$

$$\vdash? \ \exists CustomerInit \bullet \text{true}$$

The well-formedness conjecture is *trivially true*. The consistency conjecture is *true by construction*, because the instantiation of the meta-theorem cl-init-ni of T5 gives the theorem:

$$\frac{\Gamma \vdash \text{true}, \Gamma \vdash \exists \ CustomerInit \bullet \text{true}}{\text{true}}$$

Customer class, extensional view

In this view, the state space defines the set of all *existing* objects (a subset of the class's object set), and a function that maps object atoms to their state intensions. This promotion

is expressed in the $\mathbb{S}$CL generic of the ZOO domain toolkit (generic G2, appendix C). Actual class state extensions are defined as instantiations of this generic; the template captures the instantiation of the generic.

The extensional state space and initialisation of Customer is *fully-generated* from template T19:

$$\mathbb{S}Customer == \mathbb{S}\mathrm{CL}[\mathbb{O}CustomertCl, Customer][sCustomer/os, stCustomer/oSt]$$
$$\mathbb{S}CustomerInit == [\ \mathbb{S}Customer\ ' \mid sCustomer' = \varnothing \land stCustomer' = \varnothing\ ]$$

Above, the components of the generic are renamed in order to avoid name clashing when class extensions are composed to make the system schema. The initialisation assigns both the set of existing objects and the set of object to state mappings to the empty set: in the initial state there are no objects.

Here, a specific proof of the initialisation conjecture is not required; by meta-theorem cl-ext-init of template T19 this conjecture is *true by construction*.

Account class, intensional view

The required attribute types are *fully generated* from template T3; $ACCID$ is defined as a non-empty given set, and $ACCTYPE$ as a Z free type:

$$[ACCID] \qquad\qquad ACCTYPE ::= current \mid savings$$
$$\vdash ACCID \neq \varnothing$$

The class Account has a state diagram, and this affects the representation of state intension. So, a different template from the one used for Customer is required. Template T9 captures class state intensions for classes with a state diagram. This template requires the definition of a Z free-type to capture all possible states of the statechart, which is *fully generated* from the statechart:

$$AccountST ::= active \mid suspended$$

There is the usual definition of state space, initialisation and consistency conjecture. The system constraint C1 is represented here as an intensional invariant, because it constrains individual accounts. Because of this constraint, the template T9 definition of state space is *partially generated*; the user needs to provide the invariant (variable $\ll CLI \gg$ in the template), all the rest is instantiated from information coming from the UML and state diagrams. The intensional state space, initialisation and consistency conjectures of Account are:

Account

$$accountNo : ACCID$$
$$balance : \mathbb{Z}$$
$$atype : ACCTYPE$$
$$st : AccountST$$
$$atype = savings \Rightarrow 0 \leq balance$$

AccountInit

$$Account\ '$$
$$accountNo? : ACCID$$
$$atype? : ACCTYPE$$
$$accountNo' = accountNo?$$
$$balance' = 0$$
$$atype' = atype?$$
$$st = active$$

$$\vdash? \ \forall\, accountNo? : ACCID;\ atype? : ACCTYPE \bullet accountNo? \in ACCID \land 0 \in \mathbb{Z} \land atype? \in ACCTYPE$$

$$\vdash? \ \exists\, AccountInit \bullet \text{true}$$

Above, the *st* attribute and its initial value come from the statechart; all the rest comes from the class diagram. The first conjecture, *well-formedness*, is *trivially true* in Z/Eves. The second is the usual initialisation conjecture, which, is reduced by appeal to meta-theorem cl-stc-init of T9 to,

$$\vdash \exists\, accountNo? : ACCID;\ type? : ACCTYPE \bullet type? = savings' \Rightarrow 0 \leq 0$$

which is easily *provable* in Z/Eves.

Account class, extensional view

The extensional state space and initialisation of Account is *fully-generated* from template T19:

$$\mathbb{S}Account == \mathbb{S}\mathrm{CL}[\mathbb{O}AccountCl, Account][sAccount/os, stAccount/oSt]$$
$$\mathbb{S}AccountInit == [\ \mathbb{S}Account\ ' \mid sAccount' = \varnothing \land stAccount' = \varnothing\ ]$$

Again, the proof of the initialisation conjecture is not required; by meta-theorem cl-ext-init of template T19, it is *true by construction.*

4.8.3 Relational View

An association denotes a set of object links or tuples. This is described as a Z relation between the object sets of the classes being related, which denotes a set of object-tuple pairs (a set of object links).

The state space and initialisation of the association Holds is *fully generated* from template T47; the state space defines the relation between the object sets of the classes Customer and Account, and the initialisation sets the set of links to the empty set:

$$\mathbb{A}Holds == [\ rHolds : \mathbb{O}CustomerCl \leftrightarrow \mathbb{O}AccountCl\ ]$$
$$\mathbb{A}HoldsInit == [\ \mathbb{A}Holds\ ' \mid rHolds' = \varnothing\ ]$$

An initialisation conjecture is not required because by meta-theorem assoc-init of template T47, it is always true.

Also *fully generated* from template T47 is the *association link invariant*, which links the objects referred by the association to existing objects and expresses the multiplicity constraint

of the association. This is used to build the system structure (below) based on the Name Predicates pattern [SPT03b]. The link invariant of Holds is:

$$\begin{array}{|l}
\hline
\textit{Link}\mathbb{A}\textit{Holds} \\
\mathbb{S}\textit{Customer};\ \mathbb{S}\textit{Account};\ \mathbb{A}\textit{Holds} \\
\hline
\mathsf{mult}(\textit{rHolds}, \textit{sCustomer}, \textit{sAccount}, \textit{om}, \varnothing, \varnothing) \\
\hline
\end{array}$$

This includes the extensions of the participating classes and the association. The predicate uses the mult generic of the ZOO toolkit (see generic G3, appendix C), which constrains the relation *rHolds* to use *existing* objects and to have the appropriate multiplicity (1 to $*$, *om*, as defined in the UML diagram). The instantiation of the mult generic says that the inverse of the relation is a total function from the set of existing Account objects to the set of existing Customer objects, hence giving the required multiplicity.

4.8.4 Global View

The global view defines system structures, which are ensembles of classes and associations. The state space of the system comprises the state spaces of the system's components (classes and associations), association link invariants, and global constraints. The system initialisation is defined as the initialisation of the system's components. This is captured by template T54.

The global view expresses those constraints that cannot be confined to the scope of local components. These are expressed in separate schemas following the Name Predicates pattern [SPT03b]. The system constraint $C2$ is *partially-generated* from template T53:

$$\begin{array}{|l}
\hline
\textit{ConstSumBalsGEQZ} \\
\mathbb{S}\textit{Account} \\
\hline
0 \leq \Sigma\{a : \textit{sAccount} \bullet a \mapsto (\textit{stAccount}\ a).\textit{balance}\} \\
\hline
\end{array}$$

(See appendix C for the definition of Σ.)

Also global and *partially-generated* from template T53, the system constraint $C3$ involves the classes Customer, Account and the association Holds:

$$\begin{array}{|l}
\hline
\textit{ConstCompanyNoSavings} \\
\mathbb{S}\textit{Customer};\ \mathbb{S}\textit{Account};\ \mathbb{A}\textit{Holds} \\
\hline
\{oC : \textit{sCustomer} \mid (\textit{stCustomer}\ oC).\textit{type} = \textit{company}\} \\
\quad \lhd \textit{rHolds} \rhd \{oA : \textit{sAccount} \mid (\textit{stAccount}\ oA).\textit{type} = \textit{savings}\} \\
= \varnothing \\
\hline
\end{array}$$

The schema that collects all system constraints is generated from template T54, which is defined as the conjunction of all association link invariants and global constraints:

$$\textit{SysConst} == \textit{Link}\mathbb{A}\textit{Holds} \land \textit{ConstSumBalsGEQZ} \land \textit{ConstCompanyNoSavings}$$

Finally, and also from template T54, is the system state space, initialisation and initialisation conjecture. The state space includes all class extensions and associations (declarations), and the constraints schema (predicate). The initialisation is defined as the initialisation of

the system's components, expressed as conjunction of the after state of the system and all component initialisations:

System

$\mathbb{S}Customer$; $\mathbb{S}Account$; $\mathbb{A}Holds$

SysConst

$$SysInit == System\,' \land \mathbb{S}CustomerInit \land \mathbb{S}AccountInit \land \mathbb{A}HoldsInit$$

$$\vdash ?\ \exists\ SysInit \bullet \text{true}$$

In the initialisation, the inclusion of the after state of the *System* in the conjunction ensures that all constraints are taken into account in the system initialisation. The conjecture is reduced, by appeal to meta-theorem sys-init of template T54, to a conjunction of two predicates (one for each global constraint),

$$\varnothing \lhd \varnothing \rhd \varnothing = \varnothing \land 0 \leq \Sigma\ \varnothing$$

which is *trivially true* in Z/Eves.

This completes the ZOO model of the state, which captures everything about state in the UML class diagram of figure 4.6 (the operations are defined in the next section) and more: initial state as defined in the statechart and system constraints. System constraint C1 was expressed locally in the scope of Account; constraints C2 and C3 were expressed as global constraints.

4.9 ZOO model: operations

A system operation in an OO system comprises operations on a number of objects of different classes. In ZOO, this is specified by first defining local component operations (class and association) and then by defining system operations as a composition of local operations. For example, the system operation OP2 (open account) involves the creation of a new Account object and the addition of a link, made of the new account and its customer, to the association Holds; hence, the actual system operation would be defined as the composition of these two local operations. Local operations are defined in the intensional, extensional and relational views, and then they are composed in the global view to form system operations.

Naming conventions are used to distinguish update (change state) from observe (do not change state) operations. Following Z conventions, names of update operations include the letter Δ (here subscripted), whereas observe ones include the letter Ξ.

4.9.1 Class view

The class view is subdivided into intensional and extensional views. The operations of the intensional view describe either (a) an object state-transition, that is, a change in the state of a single object of the class (*update* operations), or (b) an observation upon the state of a single object (*observe* operations). In some cases, a specification of *finalisation* is required; this expresses a condition for the objects of a class to cease their existence (that is, to be removed from their extension). The operations from this view are *partially generated* from templates, with some instantiation information coming from diagrams.

The extensional view defines operations that are applicable to all existing objects of a class. In most cases, this involves *promoting* an operation from the intensional view. All promoted operations are *fully generated* from the templates of the catalogue.

Z Promotion uses *framing* schemas to form promoted operations. This has the effect of promoting local operations to a global state. ZOO uses framing schemas customised to class state extensions, drawing on the work on promotion patterns [SPT03a, SPT03b]. There is one framing schema for each kind of operation: the usual *new*, *update*, and *delete* framing schemas, and ZOO introduces the *observe* framing schema.

The following defines the operations for the classes Customer and Account.

Class Customer

The class customer does not have intensional operations. But we need to build an operation to create new Customer objects, which promotes the intensional initialisation. This is used to build operation OP1.

First, there is the definition of the new promotion frame for Customer, which is *fully generated* from template T20:

$$
\begin{array}{|l}
\Phi\mathbb{S}CustomerN \\
\Delta\mathbb{S}Customer \\
Customer' \\
oCustomer! : \mathbb{O}_x\, CustomerCl \\
\hline
oCustomer! \in \mathbb{O}_x\, CustomerCl \setminus sCustomer \\
sCustomer' = sCustomer \cup \{oCustomer!\} \\
stCustomer' = stCustomer \cup \{oCustomer! \mapsto \theta Customer'\} \\
\hline
\end{array}
$$

The new class operation, formed by promotion, is *fully generated* from template T21:

$$\mathbb{S}_\Delta CustomerNew == \exists\ Customer' \bullet \Phi\mathbb{S}CustomerN \land CustomerInit$$

The pre-condition of this operation is simplified by appeal to meta-theorem cl-ext-nop-pre of template T21 to:

$$[\ \mathbb{S}Customer \mid \mathbb{O}_x\, CustomerCl \setminus sCustomer \neq \varnothing\ ]$$

The proof of the consistency conjecture is not required for this operation, because by meta-theorem cl-ext-nop-epre of template T21 it is always true.

Class Account, intensional view

The system operations OP3 (deposit) and OP4 (withdraw) change the state of account objects (balance is incremented or decremented by some amount), and so do operations OP8 (suspend account) and OP9 (re-activate account). As Account has a statechart, these operations are generated from the template T10 (intensional update operations with a statechart). The deposit and withdraw operations are *partially generated* from this template:

Account$_\Delta$*Deposit*

$\Delta Account$
$amount? : \mathbb{N}$

$st = active \land st' = active$
$\quad \lor\ st = suspended \land st' = suspended$
$accountNo' = accountNo$
$type' = type$
$balance' = balance + amount?$

Account$_\Delta$*Withdraw*

$\Delta Account$
$amount? : \mathbb{N}$

$st = active \land st' = active$
$accountNo' = accountNo$
$type' = type$
$balance' = balance - amount?$

Here, schema declarations come from the class diagram, the values of the *st* attribute come from the statechart, and the rest is provided by the user. System constraint C4, expressed in the statechart, is formalised here: a withdraw may occur only when the account is active.

The consistency conjectures are also generated from template T10. The well-formedness conjectures of these operations are both *trivially true*; the one for *withdraw* is:

$$\vdash?\ \forall\, Account;\ amount? : \mathbb{N} \bullet \\ accountNo \in ACCID \land type \in ACCTYPE \land balance - amount? \in \mathbb{Z}$$

The precondition of *deposit* is simplified with meta-theorem cl-stc-uop-pre of template T10 and Z/Eves to give:

$$\text{pre}\ Account_\Delta Deposit = [Account;\ amount? : \mathbb{N} \mid \text{true}]$$

The consistency conjecture,

$$\vdash?\ \exists\ \text{pre}\ Account_\Delta Deposit \bullet \text{true}$$

is easily *provable* in Z/Eves.

The precondition of *withdraw* is calculated from meta-theorem cl-stc-uop-pre and Z/Eves to give:

$$\text{pre}\ Account_\Delta Withdraw = \\ [Account;\ amount? : \mathbb{N} \mid st = active \land type = savings \Rightarrow balance - amount? \geq 0\ ]$$

The consistency conjecture is easily *provable* in Z/Eves.

Similarly, the operations suspend and re-activate are also partially generated from template T10 (the declarations come from the class diagram, the values of the *st* attribute from the statechart and the rest is provided by the user):

Account$_\Delta$*Suspend*

$\Delta Account$

$st = active \land st' = suspended$
$accountNo' = accountNo$
$atype' = atype$
$balance' = balance$

Account$_\Delta$*ReActivate*

$\Delta Account$

$st = suspended \land st' = active$
$accountNo' = accountNo$
$atype' = atype$
$balance' = balance$

The consistency conjectures are also generated from template T10. The well-formedness conjectures of these operations are both *trivially true*. The preconditions of the operations are calculated with meta-theorem cl-stc-uop-pre and Z/Eves to give:

$$\text{pre}\, Account_{\Delta} Suspend = [Account \mid st = active\ ]$$
$$\text{pre}\, Account_{\Delta} ReActivate = [Account \mid st = suspended\ ]$$

The precondition consistency conjectures for these operations are both easily *provable* in Z/Eves.

The system operation OP5 (get balance) queries the *balance* of Account objects; this is *partially generated* from template T11 with the schema declarations coming from the UML diagram, the values of the *st* attribute from the statechart, and the expression that gives the value of the output from the user:

$$Account_{\Xi} GetBalance == [\ \Xi Account;\ balance! : \mathbb{Z} \mid st = active \lor st = suspended \land balance! = balance\ ]$$

The consistency conjectures are also generated from T11. The well-formedness conjecture for this operation is *trivially true*. Its precondition gives a *true* predicate using meta-theorem cl-stc-oop-pre and Z/Eves. The precondition consistency conjecture is *true by construction* by appeal to meta-theorem cl-stc-oop-epre-np of template T11.

The finalisation is *fully-generated* from template T12 with instantiation information coming from the statechart:

$$AccountFin == [\ Account \mid st = active \lor st = suspended \land balance = 0\ ]$$
$$\vdash ?\ \exists\, AccountFin \bullet \text{true}$$

This formalises system constraint C4 expressed in the statechart, which says that accounts may be closed (or deleted) provided their balance is zero.

The consistency conjecture is reduced by appeal to meta-theorem cl-stc-fin of template T12 to something that is easily *provable* in Z/Eves.

Class Account, extensional view

For system operation OP2 (open account), an operation to create Account objects is required. The new promotion frame for Account is *fully generated* from template T20:

$$\Phi\mathbb{S} AccountNI$$
$$\Delta\mathbb{S} Account$$
$$Account\ '$$
$$oAccount! : \mathbb{O}_x AccountCl$$
$$oAccount! \in \mathbb{O}_x AccountCl \setminus sAccount$$
$$sAccount' = sAccount \cup \{oAccount!\}$$
$$stAccount' = stAccount \cup \{oAccount! \mapsto \theta Account\ '\}$$

The actual new operation is also *fully generated* from template T21:

$$\mathbb{S}_{\Delta} AccountNew == \exists\, Account\ ' \bullet \Phi\mathbb{S} AccountNI \land AccountInit$$

The pre-condition of this operation is generated from meta-theorem cl-ext-nop-pre of template T20:

$$[\, \mathbb{S}Account \mid \mathbb{O}_x AccountCl \setminus sAccount \neq \varnothing \,]$$

It is not required to prove the consistency conjecture for this operation, because by meta-theorem cl-ext-nop-epre it is always true.

The operations deposit, withdraw, suspend and re-activate change the state of an Account object, and so they need to be promoted here. The update framing schema for Account is *fully generated* from template T22:

$\Phi\mathbb{S}AccountUI$

$$\Delta\mathbb{S}Account$$
$$\Delta Account$$
$$oAccount? : \mathbb{O}_x AccountCl$$

$$oAccount? \in sAccount$$
$$\theta Account = stAccount\ oAccount?$$
$$sAccount' = sAccount$$
$$stAccount' = stAccount \oplus \{oAccount? \mapsto \theta Account'\}$$

The actual class operations, formed by promotion, are *fully generated* from template T23:

$$\mathbb{S}_\Delta AccountDeposit == \exists \Delta Account \bullet \Phi\mathbb{S}AccountUI \land Account_\Delta Deposit$$
$$\mathbb{S}_\Delta AccountWithdraw == \exists \Delta Account \bullet \Phi\mathbb{S}AccountUI \land Account_\Delta Withdraw$$
$$\mathbb{S}_\Delta AccountSuspend == \exists \Delta Account \bullet \Phi\mathbb{S}AccountUI \land Account_\Delta Suspend$$
$$\mathbb{S}_\Delta AccountReActivate == \exists \Delta Account \bullet \Phi\mathbb{S}AccountUI \land Account_\Delta ReActivate$$

The precondition of these operations is simplified by meta-theorem cl-ext-uop-pre of template T23 to:

$$\mathrm{pre}\, \mathbb{S}_\Delta AccountDeposit =$$
$$[\, \mathbb{S}Account;\ oAccount? : \mathbb{O}_x AccountCl;\ amount? : \mathbb{N} \mid oAccount? \in sAccount \,]$$
$$\mathrm{pre}\, \mathbb{S}_\Delta AccountWithdraw =$$
$$[\, \mathbb{S}Account;\ oAccount? : \mathbb{O}_x AccountCl;\ amount? : \mathbb{N} \mid$$
$$oAccount? \in sAccount \land (stAccount\ oAccount?).st = active$$
$$\land\ (stAccount\ oAccount?).type = savings$$
$$\Rightarrow (stAccount\ oAccount?).balance - amount? \,]$$
$$\mathrm{pre}\, \mathbb{S}_\Delta AccountSuspend =$$
$$[\, \mathbb{S}Account;\ oAccount? : \mathbb{O}_x AccountCl \mid$$
$$oAccount? \in sAccount \land (stAccount\ oAccount?).st = active \,]$$
$$\mathrm{pre}\, \mathbb{S}_\Delta AccountReActivate =$$
$$[\, \mathbb{S}Account;\ oAccount? : \mathbb{O}_x AccountCl \mid$$
$$oAccount? \in sAccount \land (stAccount\ oAccount?).st = suspended \,]$$

The proof of consistency conjectures for update operations is not required: they are always true, provided the operation being promoted is consistent (meta-theorem cl-ext-uop-epre of template T23).

The observe operation to get balance of an account also needs to be promoted. The *observe* promotion frame for Account is *fully generated* from template T33:

$\Phi\mathbb{S}AccountO$

$\Xi\mathbb{S}Account$
$\Xi Account$
$oAccount? : \mathbb{O}AccountCl$

$oAccount? \in sAccount$
$\theta Account = stAccount\ oAccount?$

The actual operation is also *fully generated* by instantiating template T34:

$$\mathbb{S}_{\Xi}AccountGetBalance == \exists\ \Xi Account \bullet \Phi\mathbb{S}AccountO \land Account_{\Xi}GetBalance$$

Again, it is not required to prove the consistency conjecture because it is always true (meta-theorem cl-ext-oop-epre of T33).

System operation OP8 requires an operation to delete Account objects, which promotes the intensional finalisation of Account (see above). The delete promotion schema is *fully generated* from template T35:

$\Phi\mathbb{S}AccountDI$

$\Delta\mathbb{S}Account$
$Account$
$oAccount? : \mathbb{O}_x AccountCl$

$oAccount? \in sAccount$
$\theta Account = stAccount\ oAccount?$
$sAccount' = sAccount \setminus \{oAccount?\}$
$stAccount' = \{oAccount?\} \lhd stAccount$

Also *fully generated* is the actual delete operation, which is instantiated from template T36:

$$\mathbb{S}_{\Delta}AccountDelete == \exists\, Account \bullet \Phi\mathbb{S}AccountDI \land AccountFin$$

The precondition of this operation is simplified with meta-theorem cl-ext-dop-pre to:

$$[\ \mathbb{S}Account;\ oAccount? : \mathbb{O}_x AccountCl \mid oAccount? \in sAccount \\ \land\ (stAccount\ oAccount?).balance = 0]$$

The consistency conjecture is always true (meta-theorem cl-ext-dop-epre).

Finally, there is an operation that is not a promotion. The operation OP7 (get accounts in debt) is *partially-generated* from template T42:

$\mathbb{S}_{\Xi}AccountGetDebtAccounts$

$\Xi\mathbb{S}Account$
$osAccount! : \mathbb{P}(\mathbb{O}AccountCl)$

$osAccount! = \{\ a : sAccount \mid (stAccount\ a).balance < 0\ \}$

The well-formedness conjecture (instantiated from the same template) is *trivially true*. The precondition yields true in the predicate (meta-theorem cl-ext-onp-pre) and the consistency conjecture is always true by appeal to meta-theorem cl-ext-onp-epre.

4.9.2 Relational View

Operations in the relational view change or observe the state of associations, by adding to or removing tuples from the association relation, and performing queries. All the operations are *fully-generated* from templates of the catalogue.

In the trivial bank system, operation OP2 (open account) requires the addition of a tuple (link), consisting of a Customer object and an Account object, to the association Holds. The operation is *fully-generated* from template T48:

$$
\begin{array}{|l}
\hline
\mathbb{A}_{\Delta} HoldsAdd \\
\Delta \mathbb{A} Holds \\
oCustomer? : \mathbb{O} CustomerCl \\
oAccount? : \mathbb{O} AccountCl \\
\hline
holds' = holds \cup \{ oCustomer? \mapsto oAccount? \} \\
\hline
\end{array}
$$

The precondition gives a predicate that is *true* (meta-theorem assoc-atop-pre of template T48). The consistency conjecture is always true (meta-theorem assoc-atop-epre).

Operation OP8 (close account) requires that the link that exists between the account to delete and its customer (association Holds) must also be deleted. The deletion of tuples of the association Holds, given a set of Account objects, is defined by instantiating template T50:

$$
\begin{array}{|l}
\hline
\mathbb{A}_{\Delta} HoldsDelAccount \\
\Delta \mathbb{A} Holds \\
osAccount? : \mathbb{P}(\mathbb{O} AccountCl) \\
\hline
holds' = holds \rhd osAccount? \\
\hline
\end{array}
$$

The consistency conjecture is always true (meta-theorem assoc-rsop-epre of template T50).

Operation OP6 (get customer accounts) requires a list of all the accounts held by a customer, an observation on the state of *Holds*. This is defined by instantiating template T51:

$$
\begin{array}{|l}
\hline
\mathbb{A}_{\Xi} CustomerAccounts \\
\Xi \mathbb{A} Holds \\
oCustomer? : \mathbb{O} CustomerCl \\
osAccount! : \mathbb{P}(\mathbb{O} AccountCl) \\
\hline
osAccount! = holds(\!| \{ oCustomer? \} |\!) \\
\hline
\end{array}
$$

The consistency conjecture is always true by appeal to meta-theorem assoc-qfop-epre of template T51.

4.9.3 Global View

The global view defines system operations, which, as the name indicates, act upon the state of the system as a whole. System operations are defined by composition: class and association operations are composed using operators of the schema calculus, usually schema conjunction, to make a system operation. First, we see how system operations are composed in the global view. Then, we show how they are generated from templates and consistency results proved by using meta-theorems.

Composition of system operations

Composition of system operations is illustrated with system operation OP2 (open account). This involves one operation from the class view (to create a new account), and one from the relational view (to associate the new account with an existing customer). So, the composition is naturally specified as a conjunction of these two local operations:

$$SysOpenAccount == \mathbb{S}_\Delta AccountNew \land \mathbb{A}_\Delta HoldsAdd$$

Here, however, we need to adjust the way the two components communicate with each other, because $\mathbb{S}_\Delta AccountNew$ outputs the name $oAccount!$, but $\mathbb{A}_\Delta HoldsAdd$ expects the input $oAccount?$. This is resolved with renaming; the input is renamed to an output:

$$SysOpenAccount == \mathbb{S}_\Delta AccountNew \land \mathbb{A}_\Delta HoldsAdd[oAccount!/oAccount?]$$

This allows the two component operations to communicate properly, but the global constraints of the system are not being taken into account in the operation. This means that this operation, as it is stated, would violate the constraints of the system. For example, as the system constraint $C3$ is not taken into account, the operation would allow a customer of type *company* to open a *savings* account. To solve this, local component operations are *lifted*[1] to system operations by adding $\Delta System$ to the conjunction:

$$SysOpenAccount == \Delta System \land \mathbb{S}_\Delta AccountNew \land \mathbb{A}_\Delta HoldsAdd$$

This takes global constraints into account in the system operation, but introduces a *frame* problem [BMR95]. Local operations specify the change of state of their local components, but a system may include other components whose states should be unchanged by the operation. In this example, the operation should change the states of the class Account and the association Holds, but the class Customer, introduced by adding $\Delta System$, should remain unchanged. However, in Z, the state of any component that is not explicitly addressed is undetermined after the operation; so any state of $\mathbb{S}Customer$ would satisfy the specification.

This is solved by defining frames for system operations, which make explicit what is to change and what is to remain unchanged. These frames are formed by conjoining $\Delta System$ with the Ξ (nothing changes) of every system component whose state is to remain unchanged. The names of system operation frames are prefixed by Ψ (by analogy to Φ promotion frames). The frame for this example is (instantiated from template T57):

$$\Psi SysAccountHolds == \Delta System \land \Xi\mathbb{S}Customer$$

The system operation specification now becomes:

$$\begin{aligned} SysOpenAccount == {} & \Psi SysAccountHolds \land \mathbb{S}_\Delta AccountNew \\ & \land \mathbb{A}_\Delta HoldsAdd[oAccount!/oAccount?] \end{aligned}$$

In common with the constraints added to the global view of the system state, some constraints can only be expressed in the global view in terms of operations. These constraints are expressed in separate schemas, which are then used to form the system operation. In our

[1] Lifting is similar to Z *promotion*, because local operations are promoted to a global state, but the mechanism that is used to achieve this is different (here schema conjunction).

example, the constraint C5 (*to open savings account a customer must already hold a current account*) can be expressed in this way (instantiated from template T55)[2]:

$$
\begin{array}{|l}
\hline
\textit{OpConstIfSavingsHasCurrent} \\
\mathbb{S}\textit{Account};\ \mathbb{A}\textit{Holds} \\
\textit{atype}? : \textit{ACCTYPE} \\
\textit{oCustomer}? : \mathbb{O}\textit{CustomerCl} \\
\hline
\textit{atype}? = \textit{savings} \Rightarrow \\
\quad \textit{oCustomer}? \in \textit{rHolds}^{\sim}(\!| \{\textit{oA} : \mathbb{O}\textit{AccountCl} \mid (\textit{stAccount}\ \textit{oA}).\textit{atype} = \textit{current}\} |\!) \\
\hline
\end{array}
$$

The formulation of the system operation can now be completed. This and the consistency conjecture are obtained by instantiating template T59 (which captures those system operations that create an object and then add a link with a reference to the object to an association):

$$
\begin{array}{l}
\textit{SysOpenAccount} == \Psi\textit{SysAccountHolds} \land \textit{OpConstIfSavingsHasCurrent} \\
\qquad \land\ \mathbb{S}_{\Delta}\textit{AccountNew} \land \mathbb{A}_{\Delta}\textit{HoldsAdd}[\textit{oAccount}!/\textit{oAccount}?] \\
\vdash ?\ \exists \operatorname{pre} \textit{SysOpenAccount} \bullet \text{true}
\end{array}
$$

Generation of system operations and consistency checking

Above, we have seen how system operations are constructed in ZOO. We now use the *UML+Z* catalogue to generate system operations, and their preconditions. The precondition of all system operations are given in table 4.4. These were obtained by using meta-theorems of the catalogue and Z/Eves.

In the system operation to open a new account (OP2), the precondition can be simplified with meta-theorem sys-op-no-at-pre of template T59, to give:

$$
\begin{array}{l}
[\ \textit{System} \mid \operatorname{pre} \mathbb{S}_{\Delta}\textit{AccountNew} \\
\quad \land\ \exists\, \textit{oAccount}! : \mathbb{O}\textit{AccountCl};\ \textit{Account}\ ' \mid \textit{oAccount}! \notin \textit{sAccount} \bullet \\
\qquad (\textit{SysConst}' \land \textit{CondIfSavingsHasCurrent})[\theta\mathbb{S}\textit{Customer}\ ' := \theta\mathbb{S}\textit{Customer}, \\
\qquad\quad \textit{sAccount}' := \textit{sAccount} \cup \{\textit{oAccount}!\}, \\
\qquad\quad \textit{stAccount}' := \textit{stAccount} \cup \{\textit{oAccount}! \mapsto \theta\textit{Account}\ '\}, \\
\qquad\quad \textit{rHolds}' := \textit{rHolds} \cup \{\textit{oCustomer}? \mapsto \textit{oAccount}!\}] \\
\qquad \land\ \textit{AccountInit}\ ]
\end{array}
$$

After performing all the required substitutions, this is simplified with Z/Eves to:

$$
\begin{array}{l}
[\ \textit{System};\ \textit{oCustomer}? : \mathbb{O}\textit{CustomerCl};\ \textit{accountNo}? : \textit{ACCID};\ \textit{atype}? : \textit{ACCTYPE} \mid \\
\quad \mathbb{O}\textit{AccountCl} \setminus \textit{sAccount} \neq \varnothing \land \textit{oCustomer}? \in \textit{sCustomer} \\
\quad \land\ (\textit{atype}? = \textit{savings} \\
\qquad \Rightarrow (\textit{stCustomer}\ \textit{oCustomer}).\textit{ctype} \neq \textit{company} \\
\qquad \land\ (\textit{rHolds}(\!| \{\textit{oCustomer}?\} |\!)) \\
\qquad\quad \cap\{\textit{oA} : \mathbb{O}\textit{AccountCl} \mid (\textit{stAccount}\ \textit{oA}).\textit{atype} = \textit{current}\} \neq \varnothing)\ ]
\end{array}
$$

The existence proof for the state described by this precondition is *provable* in Z/Eves.

[2] There is also a well-formedness conjecture, which is *trivially true*; it is omitted here

OP1	$\mathbb{O}_x\, Customer \setminus sCustomer \neq \varnothing$
OP2	$\mathbb{O}_x\, AccountCl \setminus sAccount \neq \varnothing \land oCustomer? \in sCustomer$ $atype? = savings \Rightarrow$ $\quad (stCustomer\ oCustomer?).ctype \neq company$ $\quad oCustomer? \in (rHolds^{\sim}(\!\mid \{oA : \mathbb{O} AccountCl \mid (stAccount\ oA).type = current\} \mid\!))$
OP3	$oAccount? \in sAccount$
OP4	$oAccount? \in sAccount \land (stAccount\ oAccount?).st = active$ $(stAccount\ oAccount?).type = savings \Rightarrow$ $\quad (stAccount\ oAccount?).balance - amount? \geq 0$ $\Sigma\{\ a : sAccount \bullet a \mapsto (stAccount\ a).balance\ \} \geq amount?$
OP5	$oAccount? \in sAccount$
OP6	$true$
OP7	$true$
OP8	$oAccount? \in sAccount \land (stAccount\ oAccount?).balance = 0$
OP9	$oAccount? \in sAccount \land (stAccount\ oAccount?).st = active$
OP10	$oAccount? \in sAccount \land (stAccount\ oAccount?).st = suspended$

Table 4.4: Preconditions of system operations (calculated with meta-theorems and Z/Eves).

The system operation OP1 (Create new customer) and its consistency conjecture are *partially generated* from template T58:

$$\Psi Customer == \Delta System \land \Xi \mathbb{S} Account \land \Xi \mathbb{A} Holds$$
$$SysNewCustomer == \Psi Customer \land \mathbb{S}_{\Delta} CustomerNew$$
$$\vdash ?\ \exists \mathrm{pre}\ SysNewCustomer \bullet \mathrm{true}$$

The precondition of this operation is simplified by appeal to meta-theorems sys-op-no-pre of template T58, which is further simplified with Z/Eves to:

$$[\ System;\ name? : NAME;\ address? : ADDRESS;\ ctype? : CUSTTYPE \mid$$
$$\quad \mathbb{O}\ CustomerCl \setminus sCustomer \neq \varnothing\ ]$$

The consistency conjecture is *provable* in Z/Eves.

The system operations deposit (OP3), withdraw (OP4), suspend (OP9) and re-activate (OP10) involve one update class operation. The frames of these operations are generated from template T57 and the actual operation from template T60:

$$\Psi Withdraw == \Delta System \land \Xi \mathbb{S} Customer \land \Xi \mathbb{A} Holds$$
$$SysWithdraw == \Psi Withdraw \land \mathbb{S}_{\Delta} AccountWithdraw$$
$$\Psi Deposit == \Delta System \land \Xi \mathbb{S} Customer \land \Xi \mathbb{A} Holds$$
$$SysDeposit == \Psi Deposit \land \mathbb{S}_{\Delta} AccountDeposit$$
$$\Psi Suspend == \Delta System \land \Xi \mathbb{S} Customer \land \Xi \mathbb{A} Holds$$
$$SysSuspend == \Psi Suspend \land \mathbb{S}_{\Delta} AccountSuspend$$
$$\Psi ReActivate == \Delta System \land \Xi \mathbb{S} Customer \land \Xi \mathbb{A} Holds$$
$$SysReActivate == \Psi ReActivate \land \mathbb{S}_{\Delta} AccountReActivate$$

The consistency of these operations is provable in Z/Eves (see table 4.4 for preconditions).

System observe operations do not require a specific frame; nothing changes in the system, so they can be simply conjoined with $\Xi System$. The system operations *get balance* (OP5) and *get accounts in debt* (OP7) are generated by instantiating template T63:

$$SysGetBalance == \Xi System \land \mathbb{S}_{\Xi} AccountGetBalance$$
$$SysGetDebtAccounts == \Xi System \land \mathbb{S}_{\Xi} AccountGetDebtAccounts$$

The observe operation *get customer accounts* (OP5) is instantiated from template T64:

$$SysGetCustAccounts == \Xi System \land \mathbb{A}_{\Xi} CustomerAccounts$$

The consistency conjectures for these operations are all *true by construction* by appeal to the meta-theorems of templates T63 and T64 (see table 4.4 for preconditions).

The system operation to delete an account (OP9) is defined in a similar way, but requires a rather more elaborate adjustment to the communication between components. This operation involves deleting one Account object (operation $\mathbb{S}_{\Delta} AccountDelete$) and deleting the links involving this object in the association Holds (operation $\mathbb{A}_{\Delta} HoldsDelAccount$). The first operation takes as input one account object, but the second operation expects a set of objects. The adjustment is made in a *connector schema*, which transforms the single output of $\mathbb{S}_{\Delta} AccountDelete$ to a singleton set (instantiated from template T56):

ConnAccountOs

$$osAccount? : \mathbb{P}(\mathbb{O} AccountCl)$$
$$oAccount? : \mathbb{O} AccountCl$$
$$osAccount? = \{oAccount?\}$$

The connector is added to the system operation specification to form the correct composition, but the input *osAccount*? is hidden (its value is derived from the input *oAccount*? and not the environment); this kind of operations is captured by the template T62:

$$SysDeleteAccount == (\Psi SysAccountHolds \land \mathbb{S}_{\Delta} AccountDelete \land ConnAccountOs \land \mathbb{A}_{\Delta} HoldsDelAccount) \setminus (osAccount?)$$

The consistency conjecture of this operation is provable in Z/Eves (see table 4.4 for precondition).

4.10 Discussion

There are several design decisions involved in the ZOO style and its templates catalogue. First, those decisions that relate to the design of a Z model for OO are discussed, and then those related to the design of the ZOO templates catalogue.

4.10.1 Object-orientation in Z

There is a single type to represent all object atoms (OBJ) (section 4.8). If we just consider the structures presented here (core of ZOO), it would be possible to assign a type to each set

of objects of each class; this would give a more restricted typing and with it the advantage that more specification errors could be detected by type-checking (e.g. an object atom of some class being wrongly used as an object of another class would be detected by type-checking). However, since Z does not support subtyping, this would not work when inheritance is introduced (next chapter): a subclass object-set is a subset of object-sets of all its super-classes; so, if a class has, say, two super-classes, it is not possible to define a set that is a subset of two distinct types. Hence, the decision to go for a single type.

The definition of the association, in the relational view, uses *all possible* rather than *existing* objects (section 4.8). The link between the two is established in the association link schema. It would be possible to define all this in just one schema; the state of the association Holds would be defined as:

$$
\begin{array}{|l}
\hline
\mathbb{A}Holds \\
\mathbb{S}Customer;\ \mathbb{S}Account \\
rHolds : \mathbb{O}CustomerCl \leftrightarrow \mathbb{O}AccountCl \\
\hline
\mathsf{mult}(rHolds, sCustomer, sAccount, om, \varnothing, \varnothing) \\
\hline
\end{array}
$$

But this has some problems. Logically, it breaks the independence between the relational and extensional views. But the most serious issue is that we lose control over the frame. Every time $\Delta\mathbb{A}Holds$ is used in operations, that means that the association relation can change, but the class extensions are also allowed to change. And if $\Xi\mathbb{A}Holds$ is used, that means that neither the relation nor the class extensions are allowed to change. So, it has been decided to keep the definitions of the associations and class extensions separate and then link them (via association link schemas) when making the ensemble in the global view. And it is this separation that allows the nice modular solution for frames of system operations in the global view (section 4.9); when a component is preceeded with Ξ, it means that only that component's state is to remain unchanged and nothing else. Such a simple solution would not be possible in a representation of associations that included class extensions.

ZOO uses the relational interpretation of associations. Alternatively, a representation where associations are interpreted as properties of a class could be devised [AP03]. In this setting, a relational view would not be required, and associations would be represented as class attributes in the intensional view. The relational interpretation has been chosen because it is more abstract; the aim is to represent abstract UML models, which can then be refined to a structure where associations are represented as class attributes.[3]

The OO model of the trivial bank system has two classes and one association, and so just one ensemble (the system) is required. In larger systems, however, more than one ensemble may be required; in such cases, the system is defined as a collection of subsystems, where each subsystem comprises classes and associations; invariants may then be added at the level of the subsystem or the full system as appropriate.

4.10.2 *UML + Z* Templates catalogue

Earlier versions of the ZOO style allowed an extra invariant in class extensions and associations.[4] While developing the templates catalogue, however, it was decided that extra

[3] Preliminary research, not included in this thesis, indicates that a model with associations represented as attributes is a refinement of one that uses relations.

[4] In fact, it was so in [APS05], where system constraint $C2$ was part of the Account extension.

invariants of extensions and associations would be allowed only as global constraints. At first, it might seem counter-intuitive to do so, because constraints are being moved from the scope of local components to the scope of the global system, which is usually not a good thing to do. But there are some reasons to do so. The first is that it simplifies meta-proof over templates, because the number of entry points for extra constraints is reduced, only one is now allowed. The second reason, the main one, is that, because precondition calculation in Z is not compositional with respect to schema conjunction,[5] there is not really much to be gained, in terms of proof, by keeping the constraints in the scope of components — what is proved in the scope of components needs to be reproved when reasoning in terms of the ensemble. So, it was decided to keep the local components simple to simplify meta-proof at the level of the ensemble.

The need for connector schemas could be avoided by defining more kinds of operations in the relational view. But the number of templates can be reduced if the operations are kept a bit more generic, and then the connection between components is adjusted with connectors.

The templates of system operations are quite specific, not leaving much room for variation. They capture one possible kind of system operation with a very specific action (e.g. an operation that creates an object and then adds a link with the new object to an association). It would be possible to provide more generic templates, say a template for update system operations, but this would render meta-proof support infeasible. The calculation of preconditions of system operations is hard even without meta-proof; if too much variation is placed on the templates, then the task of proving results for the template becomes infeasible. So, it was decided to design more specific templates that capture more specific kinds of operations. This requires a larger number of templates, but, in this case, it was a price worth paying, because the meta-theorems of system operations have proved to be useful.

The catalogue of templates constitutes a repository of knowledge and experience in building ZOO models. When there is no template to support a particular description, then it needs to be defined and investigated with meta-proof. Once this is done it can be added to the catalogue; the new template becomes part of the repository of knowledge.

4.11 Related Work

Several authors have worked on views (or viewpoints) for Z. [ACWG94] and [BDBS99] propose a similar approach to views specification: (a) views are specified as separate Z specifications (each constitutes a partial specification) and then (b) composed by building an amalgamated Z specification, which is a refinement of all partial specifications. The work of Jackson [Jac95a], which has similar aims to those approaches, is the one used here, because it is about views-structuring within the context of a single Z specification; each view is a Z ADT and views are composed using Z schema conjunction.

Hall [Hal90b, Hal94] introduced the dual representation of a classes as intension and extension. ZOO builds on this work, and its most significant contribution is specification of operations, but there other smaller improvements. ZOO introduces the idea of views to achieve a better separation of concerns: intensions, extensions and associations are represented separately from each other on each view and then they are composed. This facilitates the

[5] This is a consequence of the fact that existential quantification does not distribute through conjunction. This is also a consequence of the well known and much discussed problem that the Z schema operators have poor monotonicity properties [DHR03].

construction of composite structures, which had been overlooked in the work of Hall; ZOO specifies class operations as promotions (a class is a promoted ADT), and systems as (schema) conjunction of classes and associations.

Utting and Wang [UW03] propose an OO Z style based on axiomatic definitions, rather than the schema calculus. Objects are atoms, and the relationship between atoms and state fields is given by axiomatically-defined functions, with one function per state field. Operations are also defined axiomatically, as a relation between an object and operation inputs and outputs. One problem with axiomatic-based descriptions is that it is easy to introduce accidental contradictions (especially with complex operations), and a contradictory description renders the whole model unsatisfiable. This approach may be cumbersome and difficult to use in practice because the resulting models are not succinct, lacking modularity, and because it is not as easy to compose axiomatic definitions as it is to compose schema-based ones.

It is interesting to consider the formal modelling language Alloy [JSS01, Jac06]. Alloy's semantics is similar to Z's; its main structuring construct, the signature, is very similar to the Z schema. They differ in that an Alloy signature denotes a set of atoms, its fields (or state components) are also atoms, and signature atoms and field atoms are linked through relations. Unlike Z schemas, this effectively gives identity to signature instances. In Z, a schema is a record, a mapping from names to values; so instances of a Z schema with the same value for its fields denote the same schema object. ZOO overcomes this by representing an object atom separately from its state (a schema), and then there is a function that maps object atoms to their states. In the end, the two object models are not so different.

Object-Z's [Smi95, Smi00, DR00, DB01] model of objects is similar to that of ZOO. Objects are atoms (called object identifiers) and there is a function that maps identifiers to object states. As in ZOO, calling an operation upon an object involves promotion. So, what in Object-Z is hidden (or given some syntactic sugar), in ZOO is explicit. ZOO, however, is more flexible, because concepts may be tailored to different application domains, and alternative representations of semantics concepts may be provided. Moreover, Z is a more mature language than Object-Z and has better tool-support.

ZOO is also more flexible than OO formalisations in B. Those approaches that use the B machine structuring (e.g [Ngu98]) are much more restricted than ZOO: associations must be unidirectional (the bidirectional association of the Bank case study may not be represented) and it is not easy to compose operations. Those approaches that formalise all properties into a single B machine [BDG05, SB06] result in a clumsy structuring (and the bigger the model the more clumsy it gets), which is likely to affect negatively proof and refinement.

As in ZOO, Meyer sees classes from a dual viewpoint [Mey97]. In Meyer's terms, the *type* view and the *module* view. This is reflected in the object model of the Eiffel programming language. The type view corresponds to the ZOO intensional view; it defines a set of values of an object with operations. The module view corresponds to the extensional one; it defines the services offered to the outside world.

4.12 Conclusions

This chapter developed the core of the ZOO style, the semantic domain of the $UML + Z$ framework. It also showed how ZOO models can be generated from the FTL templates of the $UML + Z$ catalogue, and consistency-checking proofs of ZOO models can be simplified by using the catalogue's meta-theorems. The next chapter extends ZOO with inheritance.

The use of hierarchy is an important component of object-oriented design. Hierarchy allows the use of type families, in which higher level supertypes capture the behaviour that all of their subtypes have in common. For this methodology to be effective, it is necessary to have a clear understanding of how subtypes and supertypes are related. This paper takes the position that the relationship should ensure that any property proved about supertype objects also holds for its subtype objects.

Barbara Liskov and Jeannnette Wing [LW94]

5 Inheritance in the ZOO style

The previous chapter has developed the core of the ZOO style. This enables the expression of important OO-related concepts, such as objects, classes, associations and systems. We have also seen how ZOO can be used as the semantic domain of the $UML{+}Z$ framework. $UML{+}Z$'s catalogue of FTL templates and meta-theorems capture ZOO's structure, and this is used to generate ZOO models from UML class diagrams made up of classes and associations.

This chapter extends ZOO with another important OO concept: inheritance. This will enable users to express inheritance hierarchies in the context of the $UML{+}Z$ framework. This chapter follows a similar structure to the previous chapter. First, the machinery for expressing inheritance in ZOO is developed. Then, this is illustrated in the context of $UML + Z$, where ZOO models with inheritance are generated from templates of the $UML + Z$ catalogue.

This chapter is divided in two parts. The first part focusses on inheritance concepts and how ZOO is extended to support them. The second part illustrates the $UML + Z$ framework in the context of OO inheritance. The first part starts with a brief explanation of inheritance and related concepts, including behavioural inheritance and its relation to an importance concept of formal development: data refinement. Then, it shows how ZOO is extended with inheritance and ZOO's approach to behavioural inheritance. Next, inheritance in ZOO is illustrated with a simple case study of Queues with an emphasis on behavioural inheritance and multiple inheritance. In the second part, the Bank case study used in the previous chapter is extended with an inheritance hierarchy to illustrate inheritance in the $UML{+}Z$ framework. Again we see how a ZOO model can be generated by instantiating templates of the catalogue and how consistency-checks can be simplified by appeal to meta-theorems, but this time in the context of OO inheritance. Finally, the work presented in this chapter is discussed and compared with related work.

5.1 Inheritance and related concepts

In any interesting and realistic system, classes are not isolated. Associations, introduced in the previous chapter, establish structural relations (with some conceptual meaning) between

unrelated classes (called *has-a* relations). Inheritance, another mechanism to relate classes, establishes relations of *specialisation* between them. This specialisation feature realises an important concern of software engineering that has already been discussed in this thesis: reuse.

As mentioned in the previous chapter, a class is a representation of some abstraction whose instances are objects. Inheritance (*subclassing*, or *specialisation*) is a hierarchical structuring mechanism with two related but slightly different features: *abstraction specialisation* and *incremental definition*. Abstraction specialisation refers to the *is-a* nature of inheritance relations between classes: an abstraction (subclass, child or descendant) *is-a* kind of another (superclass, parent or ancestor) with perhaps extra properties, but with a strong conceptual link with the parent abstraction; an object of a descendant is at the same time also an object of the parent class (and the parent class includes all objects that are its own direct instances plus those of its descendants). Incremental definition emphasises reuse because inheritance allows the definition of a new abstraction (subclass) from the definition of another (superclass), where all the features of the superclass are available in the subclass, which may add extra features of its own or adjust the inherited features.

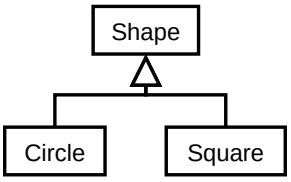


Figure 5.1: The *Shape* inheritance hierarchy.

Inheritance enables reuse and extendibility. It allows concepts to be organised in a hierarchy of classes and subclasses, where a class at any level in the hierarchy inherits all the properties of the classes higher up in the hierarchy. For example (figure 5.1), consider the class Shape with subclasses Square and Circle, where Shape is an abstraction representing a general geometric shapes, and Square and Circle represent concrete shapes.

An important property of inheritance is *substitutability*, a consequence of the *is-a* semantics. Substitutability says that subclass objects may replace superclass objects whenever superclass objects are expected. In our example, Circles and Squares may be used whenever a Shape is expected.

One now may ask: substitutability looks neat, but what is it used for? Inheritance is a mechanism of specialisation, whose purpose is to have abstractions (specialisations) that conform to some structure and behaviour, but provide slight variations. In the shapes example, we can imagine the class Shape and its subclasses for the purpose of drawing shapes on the screen, so when an operation to draw some shape (draw()) is called, then it should draw the appropriate shape, a circle or a square. This is where substitutability comes to play: to draw shapes one does not need to know whether the shape is a circle or a square, just that it is some shape with a draw operation. In OO, the mechanism that brings these variations of behaviour is called *polymorphism*; polymorphism gives the ability to select the proper behaviour (an operation) of some object from among related behaviours. In OO programming languages, polymorphism is realised through a mechanism called *dynamic binding*.

In the shapes example, the class Shape looks a little bit odd. We know how to draw a circle and a square, but how to do draw a shape? The thing is: provided we know which concrete shape it is, we know how to draw it. This leads to the idea of *abstract* (or *deferred*) class. Sometimes we just want to factor commonality in one class, and defer its concrete realisation to its subclasses; this is done with abstract classes. An abstract class has no direct instances; all its objects are direct objects of one of its subclasses. Abstract classes are used to represent some generic concept; their operations are polymorphic: they set some generic behaviour, which is realised under different forms in the subclasses. Shape is a natural abstract class.

There are two kinds of inheritance. The first, *single inheritance*, is more restricted, where a subclass is allowed to have one immediate superclass only. The second, *multiple inheritance*, is more relaxed, where a subclass is allowed to have many superclasses.

5.1.1 Behavioural Inheritance and data refinement

As said above, substitutability is an important property of inheritance. There are different ways to enforce it. Most OO systems require *interface conformity*, which gives the ability to ask subclass objects to perform the services (operations) that a superclass offers. This is checked by comparing the signatures of the operations using type-checking: provided the signatures of the subclass operations match those of its superclass then the subclass is conformant. This has a problem because one may ask a Circle to do what a Shape does, but the circle can go along and do something completely different. Suppose that a Shape has a fill operation to colour the interior of the shape, we could imagine a situation where a Shape is asked to be filled with *blue*, but Circle fills the interior with *red*. This problem is addressed by *behavioural conformity* (which coined a flavour of inheritance known as *behavioural inheritance*, after the seminal paper [LW94]), which further constrains *interface conformity* by requiring that a subclass must have a behaviour that is conformant with that of its superclass. This is enforced by exploring semantics. In the example above, if the Shape is asked to be filled with the colour blue, then also the Circle must be filled with the colour blue, not red. This is, of course, more difficult to achieve because the language being used must have a formal semantics.

Behavioural inheritance and substitutability are related to an important concept of formal development: *data refinement*. Data refinement studies how abstract representations of a data type can be refined into more concrete representations in a series of steps towards an implementation; this is related to the notion of stepwise development [Wir71]. In a formal setting, data refinement studies the conditions for a data type to correctly refine another; in a seminal paper [HHS86] data refinement is defined as:

> One datatype (call it concrete) is said to refine another data type [...] (call it abstract), if in all circumstances and for all purposes the concrete type can be validly used in place of the abstract one.

In data refinement, there is also a concern with substitutability, and an obvious connection with behavioural inheritance. This chapter devises an approach to check behavioural conformance in ZOO models that uses the Z theory of data refinement.

Next, we discuss how to extend the ZOO style to support inheritance.

5.2 Inheritance in ZOO

There are different properties of inheritance that need to be represented in ZOO. These can be divided into: (a) inheritance hierarchy, (b) class specialisation, (c) abstract classes, (d) polymorphism and (e) behavioural inheritance. This section focuses on structure, so it discusses how the first four items are represented structurally in ZOO. The next section discusses behavioural inheritance.

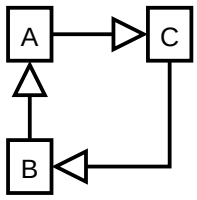


Figure 5.2: An invalid inheritance hierarchy.

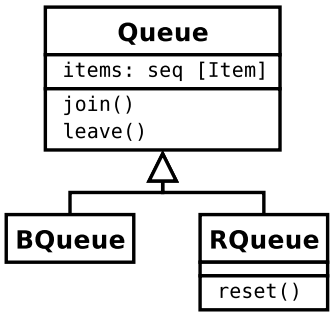


Figure 5.3: An inheritance hierarchy of Queues: Queue (unbounded queue), BQueue (bounded queue) and RQueue (resettable queue).

5.2.1 Inheritance Hierarchy

Inheritance hierarchies have a well-formedness condition: they must not have cycles. This is to avoid situations where a class is a descendant of itself (see figure 5.2 for an example).

This can be captured nicely in ZOO's structural view. There is a relation that represents the inheritance relationships between classes. The well-formedness condition is checked by proving a conjecture: the graph described by the relation must form a tree (in the case of single inheritance) or a DAG (in the case of multiple inheritance), both disallow cycles.

It is time to introduce our simple Queues hierarchy of figure 5.3. This is represented in ZOO's structural view as follows. The set of class atoms is defined in the usual way:

$$CLASS ::= QueueCl \mid BQueueCl \mid RQueueCl$$

Then there is a relation that represents the inheritance hierarchy; this is defined axiomatically as a relation on the $CLASS$ set:

$$\begin{array}{|l} subCl : CLASS \leftrightarrow CLASS \\ \hline subCl = \{BQueueCl \mapsto QueueCl, RQueueCl \mapsto QueueCl\} \end{array}$$

The conjecture that checks well-formedness is one of the two (depending on whether the inheritance is single or multiple):

$$\vdash ?\ subCl \in \mathsf{Tree} \qquad\qquad \vdash ?\ subCl \in \mathsf{Dag}$$

Tree and Dag are generics from the ZOO toolkit (appendix C, generics G1). Both these conjectures are true.[1]

[1]For the hierarchy of figure 5.2, however, the conjectures that check well-formedness are both false.

5.2.2 Subclass specialisation

The previous chapter discussed the dual meaning of a class, intension and extension, and how that is reflected in ZOO's representation of classes. Classes were seen as individual components that could be externally related with associations. Inheritance breaks this separation that exists between classes by allowing them to be extended to make other classes. We need to see how this affects the intension and extension of a class.

The following need to be addressed:

- *Subclasses extend their superclasses.* In intension, a class is defined in terms of the properties that are shared by all its objects. When subclasses enter into the picture, we have to consider how superclasses can be extended to make subclasses: a subclass inherits the properties of the superclass and may add some properties of its own.
- *Every subclass object is also a superclass object.* A class extension denotes a set of objects. With inheritance, however, this set needs to be extended. A class extension includes all its direct objects plus those that are direct instances of its subclasses (every subclass object is also a superclass object). There is, therefore, a subset relation: the extension of a subclass is a subset of the extensions of its superclasses.

The way this is realised in ZOO is better explained through the Queues example (figure 5.3 above). The class Queue with its two operations, enter (to join the queue) and leave (to allow the element at the front to leave the queue), is represented intensionally in ZOO as:

$$\begin{array}{|l} \textit{Queue}\ [\textit{Item}] \\ \hline items : \operatorname{seq} \textit{Item} \\ \hline \end{array} \qquad \begin{array}{|l} \textit{QueueInit}\ [\textit{Item}] \\ \hline \textit{Queue}\ '[\textit{Item}] \\ \hline items' = \langle\rangle \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{Queue}_\Delta \textit{Enter}\ [\textit{Item}] \\ \hline \Delta \textit{Queue}[\textit{Item}] \\ item? : \textit{Item} \\ \hline items' = items \frown \langle item? \rangle \\ \hline \end{array} \qquad \begin{array}{|l} \textit{Queue}_\Delta \textit{Leave}\ [\textit{Item}] \\ \hline \Delta \textit{Queue}[\textit{Item}] \\ item! : \textit{Item} \\ \hline item! = head\ items \\ items' = tail\ items \\ \hline \end{array}$$

The class RQueue (resettable Queue) specialises Queue by adding an operation to clear the queue (resetting); apart from this both classes present a similar behaviour. Specialisation (or extension) is expressed in Z using Z schema conjunction (or schema inclusion). Each subclass intensional definition (state space, initialisation, operations and finalisation) conjoins (or includes) the definitions of its superclasses with the definition of its own specific properties. The intension of RQueue is defined as follows:

$$\begin{array}{|l} \textit{RQueue}\ [\textit{Item}] \\ \hline \textit{Queue}[\textit{Item}] \\ \hline \end{array} \qquad \begin{array}{|l} \textit{RQueueInit}\ [\textit{Item}] \\ \hline \textit{RQueue}\ '[\textit{Item}] \\ \textit{QueueInit}[\textit{Item}] \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{RQueue}_\Delta \textit{Enter}\ [\textit{Item}] \\ \hline \Delta \textit{RQueue}[\textit{Item}] \\ \textit{Queue}_\Delta \textit{Enter}[\textit{Item}] \\ \hline \end{array} \qquad \begin{array}{|l} \textit{RQueue}_\Delta \textit{Leave}\ [\textit{Item}] \\ \hline \Delta \textit{RQueue}[\textit{Item}] \\ \textit{Queue}_\Delta \textit{Leave}[\textit{Item}] \\ \hline \end{array}$$

$$
\begin{array}{|l}
\hline
RQueue_{\Delta} Reset\ [Item] \\
\Delta RQueue[Item] \\
\hline
items' = \langle\rangle \\
\hline
\end{array}
$$

In the extensional view, the state spaces of superclass and subclasses are defined in the usual way: by instantiating the $\mathbb{S}$CL Z generic. The state extensions Queue and RQueue are:

$$\mathbb{S}Queue[Item] == \mathbb{S}\mathrm{CL}[\mathbb{O}QueueCl, Queue[Item]][sQueue/os, stQueue/oSt]$$
$$\mathbb{S}RQueue[Item] == \mathbb{S}\mathrm{CL}[\mathbb{O}RQueueCl, RQueue[Item]][sRQueue/os, stRQueue/oSt]$$

Then, there is a separate schema to express the dependencies between the extension of the subclass and that of its superclasses; namely: (a) the set of existing objects of a subclass is a subset of that of its superclasses, and (b) the mapping functions of both classes must be consistent. This constraint is expressed for the subclassing of Queue by RQueue as:

$$
\begin{array}{|l}
\hline
\mathbb{S}RQueueIsQueue\ [Item] \\
\mathbb{S}Queue[Item];\ \mathbb{S}RQueue[Item] \\
\hline
sRQueue \subseteq sQueue \\
\forall\, oRQueue : sRQueue \bullet \\
\quad (\lambda\ RQueue[Item] \bullet \theta Queue)(stRQueue\ oRQueue) = stQueue\ oRQueue \\
\hline
\end{array}
$$

The subclassing schemas are then added to the state of the system (see below).

In the extensional view, class operations are built in the usual way with promotion. However, promotion frames of subclass operations need to be changed in order to maintain the *subsetting* constraint.

Assuming the usual promotion frame (template T22), the update Queue operations are:

$$\mathbb{S}_{\Delta} QueueEnter[Item] == \exists \Delta Queue[Item] \bullet \Phi\mathbb{S}QueueU[Item] \wedge Queue_{\Delta} Enter[Item]$$
$$\mathbb{S}_{\Delta} QueueLeave[Item] == \exists \Delta Queue[Item] \bullet \Phi\mathbb{S}QueueU[Item] \wedge Queue_{\Delta} Leave[Item]$$

The promotion frames of subclasses are also built by extension. There is an intermediate frame to specify the action in the superclass, there are intermediate frames in the subclasses that extend the superclass frame. The intermediate frames for Queue and RQUeue are:

$$
\begin{array}{|l}
\hline
\Phi\mathbb{S}QueueUI_0\ [Item] \\
\Delta\mathbb{S}Queue[Item] \\
\Delta Queue[Item] \\
oQueue? : \mathbb{O}QueueCl \\
\hline
sQueue' = sQueue \\
stQueue' = stQueue \\
\quad \oplus\{oQueue? \mapsto \theta Queue\ '\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\Phi\mathbb{S}RQueueUI_0\ [Item] \\
\Phi\mathbb{S}QueueUI_0[Item][oRQueue?/oQueue?] \\
\Delta\mathbb{S}RQueueIsQueue[Item] \\
\Delta RQueue[Item] \\
oRQueue? : \mathbb{O}RQueueCl \\
\hline
sRQueue' = sRQueue \\
stRQueue' = stRQueue \\
\quad \oplus\{oRQueue? \mapsto \theta RQueue\ '\} \\
\hline
\end{array}
$$

The subclass final promotion frame extends the subclass intermediate frame by adding the required precondition:

$\Phi\mathbb{S}RQueueUI\ [Item]$

$\Phi\mathbb{S}RQueueUI_0[Item]$

$oRQueue? \in sRQueue \cap \mathbb{O}_x RQueueCl$

$\theta RQueue = stRQueue\ oRQueue?$

The promoted RQueue operations use the final frame:

$$\mathbb{S}_\Delta RQueueEnter[Item] == \exists \Delta RQueue[Item] \bullet \Phi\mathbb{S}RQueueUI[Item] \land RQueue_\Delta Enter[Item]$$
$$\mathbb{S}_\Delta RQueueLeave[Item] == \exists \Delta RQueue[Item] \bullet \Phi\mathbb{S}RQueueUI[Item] \land RQueue_\Delta Leave[Item]$$

Note that the promotion frames of the subclass ensures that satisfaction of the subsetting constraint: whenever an object is added to the subclass it is also added to the superclass. Subclass promotion frames are discussed further in the illustrations.

5.2.3 Abstract Class

The fact that a class is abstract is expressed in the class's extensional view, by saying that a class extension has no direct objects. For example, if Queue were an abstract class its extension would be defined as:

$\mathbb{S}Queue\ [Item]$

$\mathbb{S}\mathrm{CL}[\mathbb{O}QueueCl, Queue[Item]][sQueue/os, stQueue/oSt]$

$sQueue \cap \mathbb{O}_x QueueCl = \varnothing$

This says the set of *Queue* objects that are alive, must not have direct instances of *Queue*.

5.2.4 Polymorphism

As we have seen in the previous chapter, a class is a promoted ADT. In that respect, a subclass is just like any other class; its operations promote operations from the intensional view (where the superclass definitions are extended). This brings about class-specific and subclass-specific behaviour, not polymorphism.

Polymorphic operations occur in two distinct cases: (a) the class is abstract and its operations offer a choice of subclass behaviours (described as operations); (b) the class is not abstract and its operations offer its own behaviour plus those of its subclasses.

So, superclasses that are not abstract have two ADTs at the extensional level: one for their own operations, promoted from the intensional view, and another for those operations that are polymorphic. This duality is often not perceived because it remains hidden in the abstraction mechanisms of OO languages.

The polymorphic leave operation in Queue is defined as a disjunction (choice) between its own operation and the one of RQueue:

$$\mathbb{S}_\Delta QueueLeave_P[Item] == \mathbb{S}_\Delta QueueLeave[Item]$$
$$\quad \lor \mathbb{S}_\Delta RQueueLeave[Item][oQueue?/oRQueue?]$$

This is discussed further below.[2] Note that the disjunction is exclusive; that is one operation or the other is executed, but not both. This is because the object on which this operation is

[2] There is a frame problem in this definition.

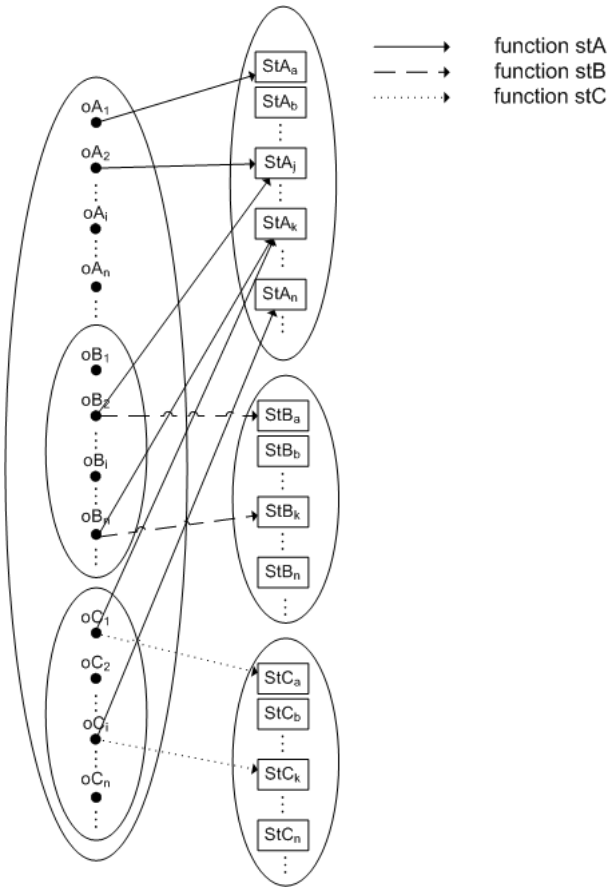


Figure 5.4: The set of all object atoms of the class A, and its subclasses B and C; A includes its exclusive objects (oAs) and the objects of its subclasses (oBs and oCs). Each class includes a set of all possible states of its objects ($STAs$, $STBs$, $STCs$), the class intension (the state of subclasses extends its superclasses). The extension defines a mapping function from existing object atom to its state; each class has its own mapping function.

executed must belong to the set of direct possible objects of either Queue ($\mathbb{O}_x QueueCl$), or RQueue ($\mathbb{O}_x RQueueCl$), but not both.

5.2.5 Overview

Figure 5.4 depicts the structure of classes in ZOO with inheritance. A class includes the set of all possible objects, of which some are alive (or existing); those objects that are alive have a state. The class intension defines the set of all possible object states; the state definition of subclasses extend those of its superclasses (not shown in the figure). In each class, there is a function mapping object atoms to their states.

Next, we discuss ZOO's approach to check behavioural conformance.

5.3 Behavioural inheritance and refinement

As said above, a ZOO class is a promoted Z ADT. Promoted ADTs are composite structures, which encapsulate another type. In ZOO, the encapsulated (or inner) type is the class intension, which defines all possible states of class objects; the outer type is the class extension. This separation is important also for behavioural inheritance.

Behavioural inheritance in ZOO is based on Z's theory of data refinement. The idea is to see a subclass as a refinement of its superclass; behavioural conformance can then be checked by proving the correctness of the refinement. Z's theory of refinement is based on ADTs, but a ZOO class comprises two ADTs. So, the question is: which ADT is supposed to be refined? The inner type (class intension)? The outer type (class extension)? Or both?

The answer is: both ADTs are supposed to be refined. However, in most cases, behavioural conformance reduces to proving the refinement of the inner type (class intension). This is because there is an important property of Z promotion with respect to refinement.

Promotion is compositional with respect to refinement provided it is free [Lup90, WD96, DB01]. In most cases promotions are free and so to demonstrate that a promoted ADT refines another it is sufficient to show that there is a refinement between the two ADTs being promoted (the inner types); the relationship holds for the rest of the system. To be more precise: suppose promoted ADTs PC and PA, which promote, respectively, C and A, then, in most cases, to show that PC refines PA, it is sufficient to show that C refines A.

So, for the purpose of behavioural inheritance, the critical ADT is the inner type (the class intension). The following develops the setting for behavioural inheritance in ZOO by discussing first class intension and then class extension.

5.4 Behavioural inheritance in the intensional view

As discussed above, essentially, behavioural conformance needs to be checked at the level of class intension (the inner type). This because there is an important property of Z promotion that tells us that, in most cases, the refinement proved at the level of the inner type is also valid at the level of class extension without the need for further prof.

In Z, the correctness of a refinement is demonstrated by proving certain conjectures, which are known as simulation rules. Appendix E gives a brief overview of Z's theory of data refinement. Without going into much details, the setting for refinement in Z is as follows: the user needs to find a refinement relation (also called retrieve or simulation relation) relating concrete and abstract data types; then the correctness of refinement is demonstrated by proving the appropriate conjectures. These conjectures vary with the type of relation and the setting of refinement.

A refinement relation may may be forwards (or downwards) if the relation is from the concrete type to the abstract one, and backwards (or upwards) if it is the other way around. There are two settings for refinement in Z, *non-blocking* (or contractual) and *blocking* (or behavioural), which differ on the way they interpret what happens outside the precondition. The former interprets an operation as a contract and so outside the precondition anything may happen. The latter says that outside the precondition an operation is blocked.

ZOO's refinement relation for behavioural inheritance is always the same. As discussed above, ZOO defines the state space of a subclass by extending the state space of its superclasses. Consider a class A (abstract) and its subclass C (concrete). Then the state of C

(concrete), subclass of A, is defined in the intensional view by this formula of the Z schema calculus (X represents the extra state of C):

$$C == A \land X$$

These two state definitions can be refinement related by the following function:

$$\lambda\, C \bullet \theta A$$

This is a total function from concrete to abstract, which given a concrete type returns the abstract type.

Z's refinement conjectures assume a general refinement relation and they vary depending on whether the relation is forwards or backwards. But given that ZOO's refinement relation for behavioural inheritance is known and it has nice properties (it is a total function), Z's refinement rules can be specialised for the purpose of behavioural inheritance. The goal is to obtain simpler refinement rules.

5.4.1 Specialising general Z data refinement for behavioural inheritance

The refinement function presented above was used to derive simpler refinement rules for behavioural inheritance. This is done in appendix E (section E.2). For this refinement relation the rules of blocking and non-blocking settings of refinement still vary, but the rules of backwards and forwards simulation reduce to the same set of rules in each of the settings. This differs from the general setting where forwards and backwards simulation have slightly different sets of rules in either of the settings.

The simulation rules are as follows. Let A and C be class intensions defined in Z such that C extends A (i.e. $C = A \land X$). Let A and C have initialisation schemas AI and CI, operations AO and CO, and finalisation schemas AF and CF. C conforms with the behaviour of A, written $C \sqsupseteq A$, in the non-blocking setting, if and only if:

1. $\vdash?\ \forall\, C' \bullet CI \Rightarrow AI$ (Initialisation)
2. $\vdash?\ \forall\, C;\ i? : I \bullet \mathrm{pre}\, AO \Rightarrow \mathrm{pre}\, CO$ (Applicability)
3. $\vdash?\ \forall\, C';\ C;\ i? : I;\ o! : O \bullet \mathrm{pre}\, AO \land CO \Rightarrow AO$ (Correctness)
4. $\vdash?\ \forall\, C \bullet CF \Rightarrow AF$ (Finalisation)

If the finalisation is total (the ADT does not have a finalisation condition) the fourth rule reduces to *true*.

The blocking setting is as the non-blocking, with the exception of the correctness rule:

1. $\vdash?\ \forall\, C';\ C;\ i? : I;\ o! : O \bullet CO \Rightarrow AO$ (Correctness)

It is now time to apply these rules.

5.4.2 The restrictions of refinement

Adding operations to the subclass

Above, we have defined one subclassing relation. The class RQueue specialised Queue by adding a new operation. But, we need to check that the behaviour of RQueue conforms with that of Queue.

In both the non-blocking and blocking settings, the initialisation conjecture is *trivially true* in Z/Eves. The correctness and applicability conjectures for the operations Enter and Leave are also *trivially true*. But how to prove behavioural conformance for the new subclass operation, reset? What should we compare it against? Is the refinement-check really required for this operation?

Yes, the refinement-check is required. Refinement is based on simulation, by comparing concrete and abstract types, for every execution in the concrete type there must be a corresponding step in the abstract. So, this new reset operation of RQueue needs to be simulated in Queue.

In formal development, this is resolved by adding an operation to the abstract type that does nothing, a *stuttering step* or a skip operation (here it would be simply $\Xi Queue$), and then prove that the new concrete operation refines skip. The idea of skip is this: if we view each operation of an ADT as buttons of a machine, then adding a skip button to the abstract data type does no harm because when pressed, from the environment, it has no effect in terms of the abstract system. So, the original contract is still valid, someone may press the new button in the abstract type but it does nothing.

The problem is that the operation reset in RQueue does not refine skip ($\Xi Queue$). The correctness conjecture is not provable. In the skip approach, concrete operations are allowed to change the extra state of the concrete type only. In the example, reset changes the *items* component, which comes from the abstract type; so it cannot possibly refine $\Xi Queue$.

Again there is conflict between the restrictions of refinement, and the use of common OO ideas, such as abstraction specialisation and incremental modification. The addition of operations to a subclass is a very common practice in OO development.

The problem of subclass constraints

It is now time to define the other class of figure 5.3, BQueue. BQueue extends Queue by adding a bound on the size of the queue and apart from this they present a similar behaviour. The intension of BQueue is defined in ZOO as follows:

$$maxQ : \mathbb{N}_1$$

BQueue [*Item*]

$$Queue[Item]$$

$$\#items \leq maxQ$$

BQueueInit [*Item*]

$$BQueue\ '[Item]$$
$$QueueInit[Item]$$

BQueue$_\Delta$*Enter* [*Item*]

$$\Delta BQueue[Item]$$
$$Queue_\Delta Enter[Item]$$

BQueue$_\Delta$*Leave* [*Item*]

$$\Delta BQueue[Item]$$
$$Queue_\Delta Leave[Item]$$

The problem is that BQueue is not behavioural conformant with its superclass. The applicability conjecture does not hold for enter: the precondition of BQueue.enter is $\#items < maxQ$, that of Queue is *true*, and

$$true \not\Rightarrow \#items < maxQ$$

In refinement, the concrete type may widen (or weaken) the precondition, not narrow (or strengthen) it. This subclassing could be implemented in any OO programming language. However, the type *BQueue* is not a refinement of type *Queue*.

That refinement failed for a good and logical reason. An operation is a contract to the outside world. The abstract operation enter (in Queue) said that it would do its job under any circumstance (the precondition is *true*). The concrete operation, however, said that it would only do its job provided some condition is met. This violates the principle of substitutability, because the concrete type when used in place of the abstract does not provide the same behaviour under the same circumstances. To understand how important this applicability restriction is for refinement, imagine a braking system of a car, where the abstract type says "upon brake slow down in any circumstance", and the concrete type says, "upon brake slow down only when the speed is less than 160 Km per hour".

The inheritance involving Queue and BQueue is a typical example of incremental definition in OO development, but which is not a refinement. We can turn around this problem by making a refactoring on the model: BQueue is promoted to superclass, Queue is eliminated, and subclasses of Queue (like RQueue), become subclasses of BQueue. But if we have to do this kind of refactoring every time we face such a problem, our use of inheritance becomes too restricted, and the inheritance hierarchies that we get will look rather stiff.

5.4.3 Reaching a compromise: relaxing the refinement constraints

An approach to behavioural inheritance that is strictly based on the refinement rules above is overly restrictive. It would render invalid many subclassings that are common in object-oriented programming, having a negative impact on the use of inheritance for reuse. To be true to refinement, and the overall goal of abstraction and reuse that underlies inheritance, we need to make a compromise. This compromise will come in the form of a relaxation to the behavioural inheritance refinement rules, which is the result of a careful analysis of behavioural inheritance and data refinement.

The theory of data refinement is given in a general setting, which assumes that ADTs are exposed to the environment. The real world interacts with the system by using them. However, that is not always the case. Some ADTs are concealed from the environment and only used internally. This is the case with the inner type of promotion and our class intension: the inner ADT is concealed and what is exposed to outside world is the outer ADT. This can be explored to relax the restrictions of behavioural inheritance.

The following shows how the refinement rules given above can be relaxed so that we may overcome the two problems mentioned above.

Relaxing with virtual operations

The restrictions on the addition of new operations are relaxed by exploring what is visible to the outside world. The aim is to allow the addition of new operations to the subclass in

a way that is sound with respect to refinement, but overcoming the restrictions of the skip approach.

The skip operation (or button) ensures true substitutability to the environment. If skip is pressed on the abstract type then it does nothing. If it is pressed on the concrete type it does something, but still it respects the behaviour set by the abstract type. Because of restrictions of simulation in Z refinement, we need to find a button that performs a similar role, but this button can be more relaxed than skip, because it is not made available to the environment.

The idea is to find a virtual operation in the superclass that simulates the operation in the subclass. The operation is virtual because it is never executed. But can such an operation be found?

Yes, it can. Besides, given the nice properties of our retrieve relation, the virtual operation can be found by calculation. The details are given in appendix E (section E.2). Briefly, given the behavioural inheritance refinement relation,

$$f = \lambda\, C \bullet \theta A$$

and a subclass operation (concrete),

$$co = \{\, CO \bullet \theta C \mapsto \theta C\,' \}$$

then the virtual superclass operation (abstract) is given by the formula:

$$ao = f^{\sim} \fatsemi co \fatsemi f$$

Appendix E.2 shows that for any concrete operation (co), the calculated abstract operation (ao) is refined by the concrete. This means that, we may add extra operations to subclasses freely. That is, no refinement proofs are required.

By using this argument, RQueue is behavioural conformant with Queue. The extension of Queue provides only two operations to the outside world, enter and leave; RQueue provides those operations as well, and in addition a third operation reset. The virtual operation that simulates reset in the intension of Queue is not available to the outside world (it is not promoted); it is there just to demonstrate the correctness of the refinement.

Relaxing by using abstract classes

The rules of behavioural inheritance may be relaxed by exploring the special properties of abstract classes. Abstract classes have no direct instances, and their operations are not like the ones of ordinary classes. In ZOO, as in OO programming, operations may be defined for an abstract class with the aim of reuse and further specialisation by descendants. However, these operations are never executed. Instead, an operation of an abstract class is always polymorphic, offering a choice of subclass behaviours. So those defined operations are never actually executed, they are also virtual. This can be explored to relax the behavioural inheritance constraints.

The relaxation is very simple: *if a subclass inherits from an abstract class, then applicability proofs are not required.* This is because the operations of an abstract class bind a behaviour for its descendants (so we need to prove correctness), but subclass (concrete) operations do not have to be applicable whenever the abstract operation is applicable, because the abstract operation is never executed. With abstract classes there is no real substitutability, because there are no abstract objects, and so substitutability does not actually take place.

This argument can be made clearer by using the *buttons* metaphor. The applicability conjecture is there to ensure that whenever the button of the abstract machine is allowed to be pressed, then it should also be allowed on the concrete machine; the concrete type may allow more situations under which the button is pressed (precondition is weakened or is less restrictive), but it cannot be made stronger (or more restrictive), because that would mean that the operation could be run on the abstract type and not on the concrete one. The special case of subclasses of an abstract class, is that only they offer those buttons to the environment. So, the subclass is not violating the contract of its superclass class, because there is none.

Of course, one needs to be careful with this relaxation and not rely on the precondition that is set by the intensional operations of an abstract class. The example of the braking system given above illustrates what can go wrong if one is not careful.

In the example above, by making Queue abstract, the need to prove applicability conjectures is lifted, and all the required correctness conjectures are provable. So BQueue conforms with the behaviour of its abstract superclass Queue.

5.5 Inheritance in the extensional view

Above, we have seen the required conditions for behavioural inheritance at the level of class intension (the inner type). Now we need to investigate the class extension (the outer-type). Below, this is discussed separately for two cases: (a) the case where the superclass and subclass promote operations and (b) the case of polymorphic class operations.

5.5.1 Promoted class operations

We have already talked about the important property of compositionality of promotion with respect to refinement provided the promotion is free. In ZOO, this means that, in most cases, checking behavioural conformance in the intensional view is sufficient to conclude behavioural conformance in terms of the whole system.

A Z promotion is free if the outer type does not impose extra constraints upon the inner type. If the inner type is constrained from the outside then the compositionality of promotion with respect to refinement may not hold. By exploring the structure of ZOO, and the way subclasses extensions and operations are built, we may relax the freeness condition so that the compositionality property applies to more cases.

The relaxation is this: *any subclass intension should not have external constraints that do not apply to its superclasses.* If that is not the case, the behaviour of subclass objects may *diverge* from the behaviour of their superclass counterparts, even if behavioural conformance has been proved at the intensional level. This becomes clearer with a simple example.

Suppose we have the intensions of classes Queue and RQueue above, and that (for now) Queue is not abstract. The behavioural conformance of RQueue by Queue has been proved at the level of class intension. We now build the class extensions and play with external constraints upon the internal states of objects (constraints upon class intensions).

$$\mathbb{S}Queue[Item] == \mathbb{S}\mathrm{CL}[\mathbb{O}QueueCl, Queue[Item]][sQueue/os, stQueue/oSt]$$

$$\begin{array}{|l}
\mathbb{S}RQueue\ [Item] \\
\hline
\mathbb{S}\mathrm{CL}[\mathbb{O}RQueueCl, RQueue[Item]][sRQueue/os, stRQueue/oSt] \\
\hline
\forall\, o : sRQueue \bullet \#(stRQueue\ o).items < 2 \\
\hline
\end{array}$$

Above, the internal states of RQueue objects are constrained, but Queue (the superclass) objects are not constrained.

That constraint breaks behavioural conformance between RQueue and Queue, because it creates *divergence*: the behaviour of subclass objects diverges from its superclass counterparts, violating the substitutability principle. Let us assume we have the extensional operations of Queue and RQueue defined above (page 112). Suppose that we create two objects of classes Queue, oQ and RQueue, oRQ, so their queues (sequences) are both empty. If we make two consecutive runs of the operation Enter on both objects, they behave similarly, adding the requested item to the sequence; on the third run, however, they diverge. oQ allows the object to be added to the sequence, but in oRQ since this behaviour is outside the precondition then either any outcome may happen (non-blocking interpretation), or the operation blocks when the operation is run (blocking interpretation). This violates the substitutability principle: oRQ cannot be used in place of oQ.

Suppose the constraint in RQueue is moved to Queue:

$$\begin{array}{|l}
\mathbb{S}Queue\ [Item] \\
\hline
\mathbb{S}\mathrm{CL}[\mathbb{O}QueueCl, Queue[Item]][sQueue/os, stQueue/oSt] \\
\hline
\forall\, o : sQueue \bullet \#(stQueue\ o).items < 2 \\
\hline
\end{array}$$

$$\mathbb{S}RQueue[Item] == \mathbb{S}\mathrm{CL}[\mathbb{O}RQueueCl, RQueue[Item]][sRQueue/os, stRQueue/oSt]$$

Then, there is no divergence anymore. This is because a superclass extension includes all objects of its subclasses. So, all subclass objects are equally affected by the constraint: when the precondition on superclass objects fails, it also fails on the objects of its subclasses.

The guideline is: constraints that only affect a subclass and not its superclasses, should be avoided, because they may cause divergence and the violation of behavioural conformance. If this is not followed, then the compositionality property no longer holds, and a proof of behavioural conformance at the extensional level may be required.

5.5.2 Polymorphic operations

Above, we analysed substitutability for promoted superclass and subclass operations. We saw that, in most cases, the substitutability proved at the intensional level propagates to the extensional level, and so subclass objects may be used whenever an object of its superclass is expected. But what about substitutability when we have polymorphic operations?

Above, it has been mentioned that non-abstract superclasses have two ADTs. The one that offers only their operations and the polymorphic ADT. The former has been discussed above; for behavioural conformance, the subclass must be a refinement of the superclass. However, with the polymorphic ADT it is the other way around: the polymorphic superclass operation should conform with the behaviour of the subclass operations that it offers, and so

the polymorphic superclass ADT should refine the subclass ADT. This is discussed further in the context of the two case studies.

Next, the simple case study of queues is explored further to illustrate ZOO's approach to inheritance.

5.6 Multiple inheritance in ZOO: the Queues case study

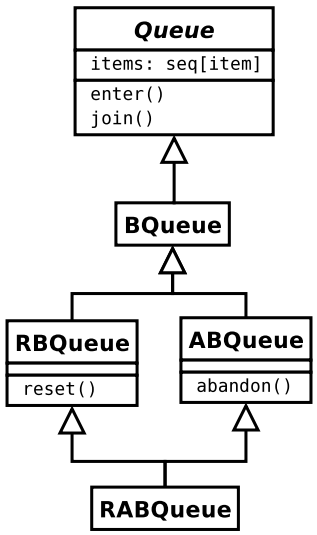


Figure 5.5: The multiple-inheritance model of Queues.

ZOO's approach to multiple-inheritance is illustrated here with the Queues case study. Above, the inheritance of Queue by BQueue (bounded-queue) was made valid by making Queue abstract. But if we try to add the class RBQueue (resettable-bounded queue) to the hierarchy so that it inherits both BQueue and RQueue, the inheritance refinement does not hold. The problem is similar to the inheritance of Queue by Bqueue discussed above: RBQueue inherits the invariants of BQueue (bound on size of queue) making the applicability conjecture of $RBQueue.enter \sqsupseteq RQueue.enter$ false. To solve this, either RQueue is made abstract, or the model is refactored.

This time, the refactoring option is the one chosen (figure 5.5). The class RQueue is eliminated and RBQueue is made a subclass of BQueue. There are two new classes: the class ABQueue (abandonable-bounded queue) allows any element to abandon the queue regardless of its position in the queue, it is a subclass of BQueue; and the class RABQueue (resettable-abandonable-bounded queue), which subclasses both RBQueue and ABQueue.

The following builds the ZOO model for this inheritance hierarchy.

5.6.1 Structural view

This view describes the class structure of the model, used to check the well-formedness of the inheritance structure. This view also defines the set of all possible objects of each class.

The following Z axiomatic definition introduces the set $CLASS$, which defines the set of class atoms of a model, the $subCl$ relation, which defines the subclass relationships of the

model, and the *rootCl* set, which is derived from the *subCl* relation and gives the classes that are not specialisations of some other class:

$$
\begin{array}{l}
CLASS ::= QueueCl \\
\quad | \; BQueueCl \\
\quad | \; RBQueueCl \\
\quad | \; ABQueueCl \\
\quad | \; RABQueueCl
\end{array}
\qquad
\begin{array}{|l}
subCl : CLASS \leftrightarrow CLASS \\
abstractCl : \mathbb{P}\, CLASS \\
rootCl : \mathbb{P}\, CLASS \\
\hline
subCl = \{BQueueCl \mapsto QueueCl, \\
\quad RBQueueCl \mapsto BQueueCl, \\
\quad ABQueueCl \mapsto BQueueCl, \\
\quad RABQueueCl \mapsto RBQueueCl, \\
\quad RABQueueCl \mapsto ABQueueCl\} \\
abstractCl = \{QueueCl\} \\
rootCl = CLASS \setminus \mathrm{dom}\, subCl
\end{array}
$$

A multiple-inheritance hierarchy, such as this one, is well-formed provided the graph representation of the hierarchy (here the *subCl* relation) forms a Directed-Acyclic Graph (DAG). To check this, the user is required to prove that the *subCl* relation is a DAG:

$$\vdash ?\ subCl \in \mathsf{Dag}$$

Dag is a generic from the ZOO toolkit (appendix D, generic G1). The conjecture is *trivially true* in the JaZA Z animator [Utt].[3]

In inheritance, the set of objects of a class includes its own objects and those of its subclasses. The $\mathbb{O}$ function gives all possible objects of a class. There is also a function to obtain the direct set of objects of a class, $\mathbb{O}_x$, which excludes the objects of the subclasses. These two functions are defined as:

$$
\begin{array}{|l}
\mathbb{O}_x : CLASS \to \mathbb{P}_1\, OBJ \\
\mathbb{O} : CLASS \to \mathbb{P}_1\, OBJ \\
\hline
\mathrm{disjoint}\ \mathbb{O}_x \\
\forall\, cl : CLASS \bullet \mathbb{O}\ cl = \mathbb{O}_x\ cl \cup \bigcup(\mathbb{O}_x(\!|\ (subCl^+)^{\sim}(\!|\ \{cl\}\ |\!)\ |\!)) \\
\forall\, cl, cl' : CLASS \mid cl \mapsto cl' \in subCl \bullet \mathbb{O}\ cl \subseteq \mathbb{O}\ cl'
\end{array}
$$

Above, the first constraint says that the direct set of objects of each class (exclude those objects of its subclasses) are mutually disjoint. The second defines $\mathbb{O}$ in terms of $\mathbb{O}_x$, and says that the set of objects of a class includes its own objects and those of its descendants. The third says that the set of all possible objects atoms of a subclass is a subset of every superclass-object set.

5.6.2 Class, intensional view

Classes Queue, BQueue and RBQueue

The intensional definitions of Queue and BQueue are given above (pages 111 and 117). The class RQueue (page 111) is slightly changed to make RBQueue: it inherits from BQueue rather

[3]This is not easy to prove in Z provers, such as Z/Eves, CADIZ or ProofPower Z, because the transitive closure in Z has a very abstract definition, which does not favour computation. The best way to prove this in, say, Z/Eves, would be to define the transitive closure recursively, hence favouring computation, and then prove that the alternative definition is equivalent to the Z one. JaZa has a recursive transitive closure.

than Queue. The initialisation and inheritance refinement conjectures for these classes are all *trivially true* in Z/Eves.

Class ABQueue

The state and initialisation of ABQueue extend the ones of BQueue:

$$ABQueue\,[Item] \quad BQueue[Item]$$

$$ABQueueInit\,[Item] \quad ABQueue\,'[Item] \quad BQueueInit[Item]$$

The initialisation conjecture is trivially true.

In the intensional view there are two types of subclass operations: (a) those that specialise (extend) a superclass operation, and (b) those that exist only in the subclass.

The operations leave and enter of ABQueue extend the ones of BQueue:

$$ABQueue_{\Delta}Enter\,[Item] \quad \Delta ABQueue[Item] \quad BQueue_{\Delta}Enter[Item]$$

$$ABQueue_{\Delta}Leave\,[Item] \quad \Delta ABQueue[Item] \quad BQueue_{\Delta}Leave[Item]$$

Finally, AQueue provides an extra operation, abandon:

$$ABQueue_{\Delta}Abandon\,[Item] \quad \Delta ABQueue[Item] \quad item? : Item$$

$$\exists\, s1, s2 : \operatorname{seq} Item \bullet items = s1 \frown \langle item? \rangle \frown s2 \land items' = s1 \frown s2$$

The conjectures that demonstrate that AQueue refines BQueue were proved in Z/Eves:

- the initialisation conjecture is trivially true;
- the applicability and correctness conjectures of enter are trivially true;
- the applicability and correctness conjectures of leave are trivially true.
- abandon is a subclass-only operation, so no refinement-related proofs are required.

Class RABQueue

The class RABQueue subclasses both RBQueue and ABQueue, so its state and initialisation extends the ones of both classes:

$$RABQueue\,[Item] \quad RBQueue[Item] \quad ABQueue[Item]$$

$$RABQueueInit\,[Item] \quad RABQueue\,'[Item] \quad RBQueueInit[Item] \quad ABQueueInit[Item]$$

The required initialisation conjecture is trivially true.

Likewise, the operations leave and enter of RBQueue extend the ones of RBQueue and ABQueue:

$RABQueue_{\Delta} Leave\ [Item]$

$\Delta RABQueue[Item]$
$RBQueue_{\Delta} Leave[Item]$
$ABQueue_{\Delta} Leave[Item]$

$RABQueue_{\Delta} Enter\ [Item]$

$\Delta RABQueue[Item]$
$RBQueue_{\Delta} Enter[Item]$
$ABQueue_{\Delta} Enter[Item]$

The operation reset extends the one of RBQueue:

$$RABQueue_{\Delta} Reset[Item] == \Delta RABQueue[Item] \land RBQueue_{\Delta} Reset[Item]$$

And abandon the one of ABQueue:

$$RABQueue_{\Delta} Abandon[Item] == \Delta RABQueue[Item] \land ABQueue_{\Delta} Abandon[Item]$$

In Z/Eves, it was proved that RABQueue refines both RBQueue and ABQueue:

- The two initialisation conjectures are trivially true.
- The four applicability and correctness conjectures for the operation enter are trivially true; the same holds for the operation leave.
- The applicability and correctness conjectures for reset ($RABQueue \sqsupseteq RBQueue$) are trivially true.
- The applicability and correctness conjectures for abandon ($RABQueue \sqsupseteq ABQueue$) are trivially true.

5.6.3 Class, extensional view

The following builds the promoted ADTs and then polymorphic operations.

State space and initialisation

The extensional state space of any class is defined by instantiating the $\mathbb{S}$CL generic. The extensional state space and initialisation of Queue are:

$\mathbb{S} Queue\ [Item]$

$\mathbb{S}\mathrm{CL}[\mathbb{O} QueueCl, Queue[Item]][sQueue/os, stQueue/oSt]$

$sQueue \cap \mathbb{O}_x QueueCl = \varnothing$

$$\mathbb{S} QueueInit[Item] == [\ \mathbb{S} Queue\ '[Item] \mid sQueue' = \varnothing \land stQueue' = \varnothing\ ]$$

Note that the state invariant above states an important property of abstract classes (recall that Queue is abstract): such a class may not have objects of its own (direct instances) and its objects are those of its subclasses (the set of exclusive object atoms of Queue and the existing objects of Queue must be disjoint). The initialisation conjecture is easily *provable* in Z/Eves.

The state space and initialisation of BQueue, a subclass of Queue, are similarly defined:

$$\mathbb{S} BQueue[Item] == \mathbb{S}\mathrm{CL}[\mathbb{O} BQueueCl, BQueue[Item]][sBQueue/os, stBQueue/oSt]$$
$$\mathbb{S} BQueueInit[Item] == [\ \mathbb{S} BQueue\ '[Item] \mid sBQueue' = \varnothing \land stBQueue' = \varnothing\ ]$$

The state definition does not include the disjointness constraint because BQueue is not an abstract class. The initialisation conjecture is easily *provable* in Z/Eves.

Unlike superclasses, subclasses needs to take into account the dependencies that exist between their own extension and those of their superclasses. This is expressed in a *subclassing schema*, which says that (a) the set of existing objects of the subclass is a subset of the superclass, and (b) that the object to state mappings in subclass and superclass are consistent. Each subclass extension needs to define one subclassing schema for each subclassing, which added to global model. The subclassing of BQueue by Queue is described extensionally as:

$\mathbb{S}BQueueIsQueue\ [Item]$

$\mathbb{S}Queue[Item];\ \mathbb{S}BQueue[Item]$

$sBQueue \subseteq sQueue$

$\forall\, oBQueue : sBQueue \bullet$
$\quad (\lambda\ BQueue[Item] \bullet \theta Queue)(stBQueue\ oBQueue) = stQueue\ oBQueue$

Promoted operations

A consequence of what is described in the subclassing schema (above) is that the superclass extension needs to change every time the subclass extension changes. That is, whenever a subclass object is created, deleted or updated, not only does the extension of the subclass need to be updated, but also all the extensions of its superclasses need to be updated. So, the way promotion frames are defined needs to be changed, but the way promoted operations are formed remains the same.

Promotion frames for inheritance are formed by extension: the subclass frame extends the superclass one. To simplify precondition calculation, ZOO uses intermediate schemas to define the action of the frame; the final promotion schema defines the constraints of the frame (including the link to the before state of the promoted object, if applicable). This defines promotions in a modular fashion, so that the underlying pattern can be captured with templates.

New promotion frames for inheritance. To specify new class operations in the Queue hierarchy, there is an intermediate frame that specifies the action in the extension of Queue, which is extended by its subclasses. This is defined as:

$\Phi\mathbb{S}QueueNI_0\ [Item]$

$\Delta\mathbb{S}Queue[Item]$
$Queue\ '[Item]$
$oQueue! : \mathbb{O}QueueCl$

$sQueue' = sQueue \cup \{oQueue!\}$
$stQueue' = stQueue \cup \{oQueue! \mapsto \theta Queue\ '\}$

The intermediate frame of BQueue, subclass of Queue, extends the intermediate frame of its superclass and includes the subclassing schema (the subclassing invariant needs to be taken into account by the operation):

$\Phi\mathbb{S}BQueueNI_0\ [Item]$

$\Phi\mathbb{S}QueueNI_0[Item][oBQueue!/oQueue!]$
$BQueue\ '[Item]$
$oBQueue! : \mathbb{O}BQueueCl$

$sBQueue' = sBQueue \cup \{oBQueue!\}$
$stBQueue' = stBQueue \cup \{oBQueue! \mapsto \theta BQueue\ '\}$

Above, the output of the superclass frame is renamed to include the name of the subclass.

The final promotion frame extends the subclass intermediate frame by adding the required constraint:

$\Phi\mathbb{S}BQueueNI\ [Item]$

$\Phi\mathbb{S}BQueueNI_0[Item]$

$oBQueue! \in \mathbb{O}_x BQueueCl \setminus sBQueue$

The predicate states the constraint of the frame by saying that the new object must be in the exclusive object-set of the class and not be one of the existing objects.

The new operation of BQueue is formed in the usual way:

$$\mathbb{S}_\Delta BQueueNew[Item] == \exists\, BQueue\ '[Item] \bullet \Phi\mathbb{S}BQueueNI[Item] \land BQueueInit[Item]$$

The new promotion frame for RBQueue, subclass of BQueue, includes an intermediate definition, which extends the superclass one:

$\Phi\mathbb{S}RBQueueNI_0\ [Item]$

$\Phi\mathbb{S}BQueueNI_0[Item][oRBQueue!/oBQueue!]$
$\Delta\mathbb{S}RBQueue[Item]$
$RBQueue\ '[Item]$
$oRBQueue! : \mathbb{O}RBQueueCl$

$sRBQueue' = sRBQueue \cup \{oRBQueue!\}$
$stRBQueue' = stRBQueue \cup \{oRBQueue! \mapsto \theta RBQueue\ '\}$

The final frame definition extends the intermediate one by adding the required constraint:

$\Phi\mathbb{S}RBQueueNI\ [Item]$

$\Phi\mathbb{S}RBQueueNI_0[Item]$

$oRBQueue! \in \mathbb{O}_x RBQueueCl \setminus sRBQueue$

The new class operation of RBQueue is then defined in the usual way:

$$\mathbb{S}_\Delta RBQueueNew[Item] == \exists\, RBQueue\ '[Item] \bullet \Phi\mathbb{S}RBQueueNI[Item] \land RBQueueInit[Item]$$

Update promotion frames. Update promotion frames are defined in the same way. The class Queue has an intermediate update promotion frame, specifying the action. The intermediate promotion frame of class BQueue extends the one of Queue:

$\Phi\mathbb{S}QueueUI_0\ [Item]$

$\Delta\mathbb{S}Queue[Item]$
$\Delta Queue[Item]$
$oQueue? : \mathbb{O}QueueCl$

$sQueue' = sQueue$
$stQueue' =$
$\quad stQueue \oplus \{oQueue? \mapsto \theta Queue\ '\}$

$\Phi\mathbb{S}BQueueUI_0\ [Item]$

$\Phi\mathbb{S}QueueUI_0[Item][oBQueue?/oQueue?]$
$\Delta BQueue[Item]$
$oBQueue? : \mathbb{O}BQueueCl$

$sBQueue' = sBQueue$
$stBQueue' =$
$\quad stBQueue \oplus \{oBQueue? \mapsto \theta BQueue\ '\}$

And the final update promotion frame for BQueue extends the intermediate frame by adding the required constraints, including the link to the before state of the promoted object:

$\Phi\mathbb{S}BQueueUI\ [Item]$

$\Phi\mathbb{S}BQueueUI_0[Item]$

$oBQueue? \in sBQueue \cap \mathbb{O}_x BQueueCl$
$\theta BQueue = stBQueue\ oBQueue?$

Having defined the frame, the update subclass operations can be defined in the usual way:

$$\mathbb{S}_\Delta BQueueEnter[Item] == \exists \Delta BQueue[Item] \bullet \Phi\mathbb{S}BQueueUI[Item] \\ \land\ BQueue_\Delta Enter[Item]$$

Polymorphic Operations

The class operations defined above are applicable to the exclusive objects of the class only. They cannot be applied to an object of its subclasses; this is what is enforced by the constraints of the frames involving $\mathbb{O}_x$. To allow operations to be applicable to all objects of a class (including those of subclasses) the operations need to be made polymorphic.

A polymorphic operation offers a choice of alternative behaviours. This choice is resolved based on the class of the object (a class operation is applied on an object). This choice of behaviours is naturally specified in Z as schema disjunction. Two cases need to considered: (a) the class is abstract, so the operation offers the behaviours of its subclasses; and (b) the class is not abstract, so the operation offers the choice between its own behaviour and that of its subclasses.

Polymorphic operations are built in a bottom-up fashion. So, to define the polymorphic version of the operation enter(), we start with the classes RBQueue and ABQueue. RBQueue is not abstract, so its polymorphic enter() is defined as choice between enter() on itself and on RABQueue:

$$\mathbb{S}_\Delta RBQueueEnter_P[Item] == \mathbb{S}_\Delta RBQueueEnter[Item] \\ \lor\ \mathbb{S}_\Delta RABQueueEnter[Item][oRBQueue?/oRABQueue?]$$

This poses, however, a frame problem. The frame of the polymorphic operation says that either of the two classes may change. However, only if the operation is run on the subclass are both extensions changed. This frame problem can be solved as:

$$\mathbb{S}_\Delta RBQueueEnter_P[Item] == \mathbb{S}_\Delta RBQueueEnter[Item] \land \Xi\mathbb{S}RABQueue[Item] \\ \lor \mathbb{S}_\Delta RABQueueEnter[Item][oRBQueue?/oRABQueue?]$$

The operation enter() on ABQueue is built in the same way:

$$\mathbb{S}_\Delta ABQueueEnter_P[Item] == \mathbb{S}_\Delta ABQueueEnter[Item] \land \Xi\mathbb{S}RABQueue[Item] \\ \lor \mathbb{S}_\Delta RABQueueEnter[Item][oABQueue?/oRABQueue?]$$

In BQueue, the polymorphic enter() offers a choice between enter() on itself and on its subclasses, RBQueue and ABQueue:

$$\mathbb{S}_\Delta BQueueEnter_P[Item] == \\ \mathbb{S}_\Delta BQueueEnter[Item] \land \Xi\mathbb{S}RBQueue[Item] \land \Xi\mathbb{S}ABQueue[Item] \\ \land \mathbb{S}_\Delta RBQueueEnter_P[Item][oBQueue?/oRBQueue?] \land \Xi\mathbb{S}ABQueue[Item] \\ \land \mathbb{S}_\Delta ABQueueEnter_P[Item][oBQueue?/oABQueue?] \land \Xi\mathbb{S}RBQueue[Item]$$

Here the solution of the frame problem differs slightly from the one above. The frame includes the superclass and its subclasses. So, if the operation is run on the superclasses, then its subclasses do not change; but if the operation runs on one of the subclasses, then it is the other subclass that is not allowed to change.

Finally, enter() on Queue offers the choice of the polymorphic operations of its subclass only because Queue is abstract:

$$\mathbb{S}_\Delta QueueEnter[Item] == \mathbb{S}_\Delta BQueueEnter_P[Item][oQueue?/oBQueue?]$$

5.6.4 Global View

The global view builds the system structure, which comprises all the local components (classes and associations), and include constraints of global scope.

The global constraints are conjoined to make the global constraints schema:

$$SysConst[Item] == \mathbb{S}BQueueIsQueue[Item] \land \mathbb{S}RBQueueIsBQueue[Item] \\ \land \mathbb{S}ABQueueIsBQueue[Item] \land \mathbb{S}RABQueueIsRBQueue[Item] \\ \land \mathbb{S}RABQueueIsABQueue[Item]$$

The system schema includes all class extensions and the global constraints schema:

System [*Item*]

$\mathbb{S}Queue[Item]$; $\mathbb{S}BQueue[Item]$; $\mathbb{S}RBQueue[Item]$
$\mathbb{S}ABQueue[Item]$; $\mathbb{S}RABQueue[Item]$

$SysConst[Item]$

The system initialisation is built by conjoining the initialisations of all class extensions:

$$SysInit[Item] == System\ '[Item] \land \mathbb{S}QueueInit[Item] \land \mathbb{S}BQueueInit[Item] \\ \land \mathbb{S}RBQueueInit[Item] \land \mathbb{S}ABQueueInit[Item] \land \mathbb{S}RABQueueInit[Item]$$

The system initialisation conjecture is easily provable in Z/Eves.

Likewise, system operations are defined by composing local operations. The system operation *enter* is:

$$SysEnter[Item] == \Delta System[Item] \land \mathbb{S}_\Delta QueueEnter[Item]$$

5.6.5 Discussion

The promotion frames have to be changed to support the requirements of inheritance. Whenever a subclass extension changes (an object is created, updated or deleted) its superclasses must also change, since this object is also in their extension. To accommodate this, a modular scheme of building promotion frames has been designed, where the subclass frame extends the superclass one and the constraints of the frame are separated from its actions. This is modular and easily captured with templates (see next section).

Note also that the usual precondition of update promotion frames had to be changed. For example, in the case BQueue the precondition,

$$oBQueue? \in sBqueue$$

would allow the operation to run on objects that are not direct instances of BQueue. And this would break the consistency that should exist between the states of objects in superclass and subclass (see subclassing schemas); that is, the state of an object in the superclass would have one state and in the subclass another. For example, consider an existing object $oRBQ$ of class RBQueue, subclass of BQueue. This object belongs to the set $sBqueue$, so if we keep the simple pre-condition (above), we could run an update operation of BQueue upon this object; this means that the state of $oRBQ$ will change in $\mathbb{S}BQueue$ but it will not change in $\mathbb{S}RBQueue$. We break the usual subclass consistency invariant. It is to avoid this problem that the precondition is strengthened to,

$$oBQueue? \in sBqueue \cap \mathbb{O}_x BQueueCl$$

so that only direct objects of the class are allowed.

As this example shows, behavioural conformance may require that inheritance class hierarchies be refactored. In the example above, BQueue could be made a subclass of Queue by making Queue abstract. Also, the class RBQueue could be added by refactoring the structure of the hierarchy; it was not possible to add RBQueue as a subclass of both BQueue and RQueue and to keep BQueue non-abstract.

This example (and the next) also show how useful the abstract class relaxation is. By making certain classes abstract, we are able to design flexible inheritance hierarchies that can take advantage of polymorphism. Without that relaxation the structures that could be made behavioural conformant would look inflexible. For example, in the example the classes *Queue* and *BQueue* would have to be merged. But abstract class relaxation enables a useful feature of inheritance, *incremental definition*, and allows us to respect the principles of behavioural inheritance.

The same holds for the relaxation that allows the addition of extra operations in the subclass. If these extra subclass operations had to refine skip, then the use of inheritance would be severely limited, and would not reflect the use that inheritance has in OO programming. Thus, these two relaxations allows us to capture a wider variety of subclassings that are common in OO programming (see next case study).

The example also shows that inheritance refinement proofs are trivial. In the example above, they are all *trivially true* in the Z/Eves theorem prover.

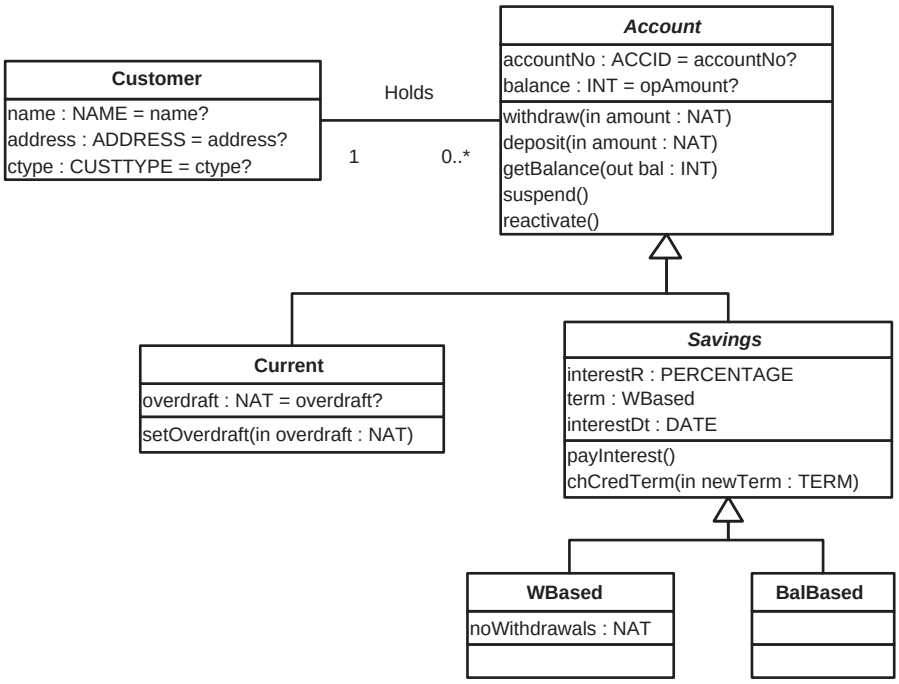


Figure 5.6: The class diagram of the Bank system with inheritance.

5.7 The Bank case study with inheritance

The previous chapter used the case study of a trivial *Bank* system to illustrate the core of the ZOO style (p.84). Here, the Bank case study is extended to support different kinds of bank accounts, which have properties in common, but also have properties of their own. This is naturally modelled in an inheritance hierarchy. The modified class diagram for this version of the Bank case study is given in figure 5.6. This version makes the following modifications:

- Account is an abstract class; it represents what is common to all kinds of account.
- Current specialises Account by allowing an overdraft.
- Savings is an abstract class that specialises Account and represents a general savings account; the attributes record the interest rate (interestR), the term (monthly, quarterly, or yearly) for crediting interest into the account, and the date when the next credit of interest due (interestDt).
- WBased specialises the class Savings by allowing the interest rate to decreases with the number of account withdrawals; so it adds an attribute to count how many withdrawals have been made since the last credit of interest (noWithdrawls).
- BalBased specialises the class Savings by allowing the actual interest rate to depend on the account's balance.

The extended Bank system adds three extra system operations, documented in table 5.1 (original operations in table 4.2, p. 86). It also adds extra system constraints, documented in

OP11	*Set Overdraft*	Set the overdraft of a current account.
OP12	*Pay Interest*	Pays due interest by crediting the savings acccount.
OP13	*Change Credit Term*	Change the term for crediting interest in a savings accounts.

Table 5.1: The extra operations of the extended bank system.

C7	The balance of Current accounts must not go below the allowed overdraft.
C8	Each kind of account requires a minimum opening amount.
C9	Savings accounts have a maximum allowed initial interest rate, which depends on the opening amount.
C10	The interest rate of WBased accounts may be decreased by a factor that depends on the number of withdrawals made.
C11	The interest rate of BalBased depends on the account's balance.

Table 5.2: The extra constraints of the extended bank system.

table 5.2 (the original constraints are in table 4.3, p. 86). The statechart of the class Account remains the same; it is as defined in figure 4.7 (p. 87).

The following presents part of the ZOO model of the extended Bank system, which is generated from templates of the ZOO catalogue (appendix D). The full generated ZOO model is given in appendix F.2. The presentation is divided into the static ZOO model (state space) and model of behaviour (operations).

5.7.1 ZOO model, state space

The following shows how illustrative components of the ZOO model are generated by instantiating templates of the ZOO catalogue for this version of the Bank system. The class Customer and the relational view are as defined in the previous chapter.

Structural view

This view defines the global names that are used in the other views. All definitions are *fully generated* from templates with information coming from the UML class diagram.

The Z that follows is generated from template T1. It starts by declaring a Z section for the ZOO model. Then, there are several definitions that capture the structure of the class model and conjectures to check that the class structure is well-formed:

section $BankInh\_model$ parents $ZOO\_toolkit$

$CLASS ::= CustomerCl \mid AccountCl \mid CurrentCl \mid SavingsCl \mid WBasedCl \mid BalBasedCl$

$$
\begin{array}{|l}
subCl : CLASS \leftrightarrow CLASS \\
abstractCl : \mathbb{P}\ CLASS \\
rootCl : \mathbb{P}\ CLASS \\
\hline
subCl = \{CurrentCl \mapsto AccountCl, \\
\quad SavingsCl \mapsto AccountCl, \\
\quad WBasedCl \mapsto SavingsCl, \\
\quad BalBasedCl \mapsto SavingsCl\} \\
abstractCl = \{AccountCl, SavingsCl\} \\
rootCl = CLASS \setminus \mathrm{dom}\ subCl
\end{array}
$$

$\vdash ?\ subCl \in \mathsf{Tree}$

$\vdash ?\ abstractCl = abstractCl \cap \mathrm{ran}\ subCl$

CLASS defines the set of all classes of the model (represented as atoms), *subCl* defines the set of all subclass relationships of the class diagram, and *rootCl*, derived from *subCl*, gives all the root classes of the model (that is, those that do not specialise other classes).

Since the model is to be used in a setting of single inheritance, the first conjecture requires that the inheritance hierarchy forms a tree (Tree is a generic of the ZOO toolkit). The conjecture is easily *provable* in the JaZA Z animator [Utt]. The second conjecture requires that each abstract class of the model has at least one descendant, which is *trivially true* in JaZA and easily provable in Z/Eves.

Generated from template T2, the functions $\mathbb{O}$ and $\mathbb{O}_x$ give, respectively, the set of all possible objects of a class and the exclusive set of objects of each class:

$$
\begin{array}{|l}
\mathbb{O} : CLASS \rightarrow \mathbb{P}_1\ OBJ \\
\mathbb{O}_x : CLASS \rightarrow \mathbb{P}_1\ OBJ \\
\hline
\mathrm{disjoint}\ \mathbb{O}_x \\
\forall\ cl : CLASS \bullet \mathbb{O}\ cl = \mathbb{O}_x\ cl \cup \bigcup(\mathbb{O}_x(\!|\ (subCl^+)^\sim(\!|\ \{cl\}\ |\!)\ |\!)) \\
\forall\ cl, cl' : CLASS \mid cl \mapsto cl' \in subCl \bullet \mathbb{O}\ cl \subseteq \mathbb{O}\ cl'
\end{array}
$$

Class Account, intension

In this view, the subclass ADTs extend the ADTs of their superclasses. That is, the subclass definitions extend the superclass one with what is specific to the subclass. This is expressed in Z using schema conjunction.

Account's definition of the previous chapter (p. 89) needs to be modified. The *atype* attribute (type of account) and its Z type are removed, because now there is an inheritance hierarchy representing all kinds of accounts. Also for this reason, the state invariant restricting the *balance* is moved to the class Savings. The initialisation is also modified because accounts are now opened with some initial amount.

Fully generated from template T9, the intensional state space, initialisation and conjectures for Account are:

$AccountST ::= active \mid suspended$

Account

$st : AccountST$
$accountNo : ACCID$
$balance : \mathbb{Z}$

AccountInit

$Account\ '$
$AccountInit_I$

$st' = active$
$accountNo' = accountNo?$
$balance' = opAmount?$

AccountInit$_I$

$accountNo? : ACCID$
$opAmount? : \mathbb{N}$

$\vdash?\ \forall\, AccountInit_I \bullet accountNo? \in ACCID \land opAmount? \in \mathbb{Z}$

$\vdash?\ \exists\, AccountInit \bullet \text{true}$

The state space (*Account*) includes the attribute *st*, which comes from the state diagram; all other attributes come from the class diagram. In the initialisation, the initial values come from the state (*st*) and class (all others) diagrams; the input opening amount (*opAmount*?) represents the account's opening amount.

Above, the first conjecture, *well-formedness*, is *trivially true* in Z/Eves. The second, initialisation, is *true by construction* by appeal to meta-theorem cl-stc-init of template T9 (there is no class invariant).

Class Account, extension

Fully generated from template T19, the extensional state space and initialisation for Account are:

$$\mathbb{S}Account == [\ \mathbb{S}\mathrm{CL}[\mathbb{O}AccountCl, Account][sAccount/os, stAccount/oSt] \mid sAccount \cap \mathbb{O}_x AccountCl = \varnothing$$

$$\mathbb{S}AccountInit == [\ \mathbb{S}Account\ ' \mid sAccount' = \varnothing \land stAccount' = \varnothing\ ]$$

$$\vdash?\ AccountCl \in rootCl$$

$$\vdash?\ AccountCl \in abstractCl \Leftrightarrow \mathbb{S}Account \Rightarrow sAccount \cap \mathbb{O}_x AccountCl = \varnothing$$

The well-formedness conjectures are *trivially true* in Z/Eves. The initialisation consistency conjecture is always true by meta-theorem cl-ext-init of template T19.

Class Current, intension

System constraint C8 (table 5.1) says that *each kind of account requires a minimum opening amount*. To express this in Current and other classes, we need a function that gives, for each kind of account, its minimum opening amount. So, first, there is a Z free type to represent the possible accounts that can be opened (current, withdraw-based and balance-based), which is *partially generated* from template T3:

$$ACCTY ::= current \mid wbased \mid balbased$$

The actual function is *partially generated* from template T4 (generates free axiomatic definitions); it is a total function that for each type of account gives the minimum opening amount (a natural number):

$reqOpAmount : ACCTY \rightarrow \mathbb{N}$

This is used to define the actual constraint (below).

Partially generated from template T14, the state space, initialisation and consistency conjectures for Current, subclass of Account, are:

Current$_0$

$overdraft : \mathbb{N}$

Current

$Account$
$Current_0$

$balance + overdraft \geq 0$

CurrentInit$_I$

$AccountInit_I$
$overdraft? : \mathbb{N}$

CurrentInit

$Current\ '$
$AccountInit$
$CurrentInit_I$

$opAmount? \geq reqOpAmount\ current$
$overdraft' = overdraft?$

$\vdash?\ \forall\, CurrentInit_I \bullet overdraft? \in \mathbb{N}$

$\vdash?\ \exists\, CurrentInit \bullet \text{true}$

The extra attributes of Current are defined in a separate schema (which come from the class diagram). The state space definition extends Account with the extra attributes and includes an invariant (from the user) representing constraint C7 (table 5.1). The initialisation also extends Account initialisation, sets the extra attributes to their initial values (from the class diagram) and describes constraint C8 (from the user).

Above, the well-formedness conjecture is *trivially true*. The initialisation conjecture is easily *provable* in Z/Eves after the simplification given by meta-theorem scl-init of template T14. It is not required to prove the initialisation refinement conjecture because it is always true by meta-theorem scl-init-ref of template T14.

Class Current, extension

Fully generated from template T24, the extensional state space, initialisation and subclassing schema for Current are:

$$\mathbb{S}Current == \mathbb{S}\mathrm{CL}[\mathbb{O}CurrentCl, Current][sCurrent/os, stCurrent/oSt]$$
$$\mathbb{S}CurrentInit == [\ \mathbb{S}Current\ ' \mid sCurrent' = \varnothing \wedge stCurrent' = \varnothing\ ]$$

$\mathbb{S}CurrentIsAccount$

$\mathbb{S}Account;\ \mathbb{S}Current$

$sCurrent \subseteq sAccount$
$\forall\, oCurrent : sCurrent \bullet$
$\quad (\lambda\ Current \bullet \theta Account)(stCurrent\ oCurrent) = stAccount\ oCurrent$

The subclassing schema expresses the required dependency constraints between the extensions of Current and Account. The initialisation includes the subclassing schema. It is not required to prove the initialisation conjecture because it is always true by meta-theorem scl-ext-init of template T24.

Class Savings

The class Savings requires the types, *PERCENTAGE*, *TERM* and *DATE*, which are *fully generated* from template T3:

$[DATE]$

$$DATE \neq \varnothing$$

$PERCENTAGE == 1 \,..\, 100$

$TERM ::= month \mid quarter \mid year$

Partially generated from template T14, the state space, initialisation and conjectures for Savings, subclass of Account, are:

Savings$_0$

$interestR : PERCENTAGE$
$term : TERM$
$interestDt : DATE$

Savings

$Account$
$Savings_0$

$balance \geq 0$

SavingsInit$_I$

$AccountInit_I$
$interestR? : PERCENTAGE$
$term? : TERM$
$today? : DATE$

SavingsInit

$Savings\ '$
$AccountInit$
$SavingsInit_I$

$interestR' = interestR?$
$term' = term?$
$interestDt' = today?$

$\vdash?\ \forall\, SavingsInit_I \bullet term? \in \mathbb{N} \land today? \in DATE$

$\vdash?\ \exists\, SavingsInit \bullet \text{true}$

The extra attributes of Savings are defined in a separate schema (from the class diagram), which is included in the final definition of state space. The state space invariant describes system constraint C1 (from the user). The initialisation extends the initialisation of Account by giving an initial value to the extra attributes (from the user).

Above, the well-formedness conjecture is trivially true in Z/Eves. The initialisation conjecture is easily *provable* in Z/Eves after the simplification given by meta-theorem scl-init of template T14. The proof of the refinement initialisation conjecture is not required because by meta-theorem scl-init-ref it is always true.

The extension of Savings is generated from template T24. The class definition is similar to the one of Account (because Savings is abstract, see above) and the subclassing schema is similar to the one of Current (above). See appendix F (section F.2.5, page 286) for the full extensional definition of Savings.

Class WBased

Constraint C9 says that each kind of savings account has a maximum initial interest rate (constraint C9), which depends on the account's opening amount. To represent this, there

is a total function that given a type of savings account and an opening amount, returns the initial interest rate; this is *partially generated* from template T4:

$$maxInitInterest : (\{wbased, balbased\} \times \mathbb{Z}) \rightarrow PERCENTAGE$$

Partially generated from template T14, the state space and initialisation of WBased, subclass of Savings, are:

$WBased_0$

$noWithds : \mathbb{N}$

$WBased$

$Savings$

$WBased_0$

$WBasedInit_I$

$SavingsInit_I$

$WBasedInit$

$WBased\ '$

$SavingsInit$

$WBasedInit_I$

$opAmount? \geq reqOpAmount\ wbased$

$interestR? \leq maxInitInterest(wbased, balance')$

$noWithds' = 0$

The extra state space of WBased, defined in a separate schema, adds the attribute *noWithds* (from the class diagram). The state space extends Savings with the extra attributes. In the initialisation, the counter of withdrawals is initialised to 0 (comes from the user), the opening amount is constrained to be at least the minimum for WBased accounts (constraint C8, from the user), and the interest rate is constrained to be less than the maximum allowed for the opening amount (constraint C9, from the user).

The well-formedness conjecture is *trivially true* in Z/Eves. The initialisation conjecture is easily *provable* in Z/Eves after the simplification given by meta-theorem scl-init of template T14. The proof of the initialisation refinement conjecture is not required because by meta-theorem scl-init it is always true.

The extension of WBased is *fully generated* from T24 (as in Current above); see appendix F.2.6 for further details.

Class BalBased

Similarly, the state, initialisation and consistency conjectures for BalBased are *partially generated* from template T14:

BalBased

$Savings$

BalBasedInit$_I$

$SavingsInit_I$

BalBasedInit

$BalBased\ '$
$SavingsInit$
$BalBasedInit_I$

$opAmount? \geq reqOpAmount\ balbased$
$interestR? \leq maxInitInterest(balbased, balance')$

Above, constraints C8 and C9 are expressed in the initialisation (from the user).

Proof of required consistency conjectures is as for class WBased (see above). The extension of BalBased is *fully generated* from T24 (as above for Current); see F.2.7 for the full ZOO definition of class BalBased.

Global View

This view describes the global constraints of the system. Constraint C2 is as defined in the previous chapter (p. 91). Constraint C3 (*companies cannot have savings accounts*), however, needs to be re-written, because it is affected by the new structure; it is *partially generated* from template T53:

ConstCompanyNoSavings

$\mathbb{S}Customer;\ \mathbb{S}Savings;\ \mathbb{A}Holds$

$\{oC : sCustomer \mid (stCustomer\ oC).type = company\} \lhd holds \rhd sSavings = \varnothing$

The system state space and initialisation is *fully generated* from template T54:

$$SysConst == Link\mathbb{A}Holds \land ConstSumBalsGEQZ \land ConstCompanyNoSavings$$

System

$\mathbb{S}Customer;\ \mathbb{S}Account;\ \mathbb{A}Holds;\ \mathbb{S}Current;\ \mathbb{S}Savings;\ \mathbb{S}WBased;\ \mathbb{S}BalBased$

$SysConst$

$$SysInit == System' \land \mathbb{S}CustomerInit \land \mathbb{S}AccountInit \land \mathbb{A}HoldsInit \\ \land \mathbb{S}CurrentInit \land \mathbb{S}SavingsInit \land \mathbb{S}WBasedInit \land \mathbb{S}BalBasedInit$$

$$\vdash?\ \exists SysInit \bullet \text{true}$$

The system initialisation conjecture is simplified by using meta-theorem sys-init and what is left (see p. 92) is *trivially true* in Z/Eves.

This completes the ZOO model of state, which captures everything in the UML class diagram (fig. 4.6) and more (initialisations as defined in class diagram and statechart, and system constraints). System constraint C1 is expressed locally in the intension of Savings; constraints C2 is expressed as a global constraint; constraint C7 is expressed locally in the intension of Current; constraints C8 and C9 are expressed locally in the intensional initialisations of Current, WBased and BalBased.

5.7.2 ZOO model, operations

The operations of the class Customer and the operations of the relational view are as defined in the previous chapter. The following presents some operations of the Account hierarchy (class view) and global view.

Class Account, intensional view

Some subclass operations are specialisations of some superclass operation. In the intensional view, these are defined by extending the superclass operation. So, the superclass operation defines the common behaviour of all its objects, which are specialised in the subclasses. The subclass operation *extends* the superclass by defining its own behaviour.

Account is an abstract class; it defines the behaviour that is shared by its child classes. The shared behaviour of the operation withdraw is *partially generated* from template T10:

$Account_{\Delta}\, Withdraw$

$$\Delta Account$$
$$amount? : \mathbb{N}$$
$$st = active \land st' = active$$
$$accountNo' = accountNo$$
$$balance' = balance - amount?$$

The well-formedness conjecture is *trivially true* in Z/Eves. The precondition is simplified with meta-theorem cl-stc-uop-pre of template T10 to give:

$$[\ Account;\ amount? : \mathbb{N} \mid st = active\ ]$$

The precondition consistency conjecture is *true by construction* (meta-theorem cl-stc-uop-epre-np).

Account has a statechart. In this version of *UML + Z* statecharts on subclasses are not allowed. So, to avoid undesired changes to the behaviour prescribed by the statechart of Account, there is an *history invariant* to be extended by the extra operations of the subclasses of Account, which disallows changes to the attribute *st*. This is *fully generated* from template T13:

$$AccountH == [\ \Delta Account \mid st = st'\ ]$$

Class Account, extensional view

Account is abstract, so it does not have operations of its own; its operations are polymorphic. However, Account needs to define inheritance promotion frames to be extended by its subclasses.

Fully generated from templates T25 and T29, the intermediate new and update promotion frames for Account are:

$\Phi\mathbb{S}AccountNI_0$

$\Delta\mathbb{S}Account$
$Account\ '$
$oAccount! : \mathbb{O}AccountCl$

$sAccount' = sAccount \cup \{oAccount!\}$
$stAccount' =$
$\quad stAccount \cup \{oAccount! \mapsto \theta Account\ '\}$

$\Phi\mathbb{S}AccountUI_0$

$\Delta\mathbb{S}Account$
$\Delta Account$
$oAccount? : \mathbb{O}AccountCl$

$sAccount' = sAccount$
$stAccount' =$
$\quad stAccount \oplus \{oAccount? \mapsto \theta Account\ '\}$

Delete promotion frames also involve intermediate and final definitions. The intermediate definitions for Account is *fully generated* from template T37 to give:

$\Phi\mathbb{S}AccountDI_0$

$\Delta\mathbb{S}Account$
$Account$
$oAccount? : \mathbb{O}AccountCl$

$sAccount' = sAccount \setminus \{oAccount?\}$
$stAccount' = \{oAccount?\} \lhd stAccount$

Class Current, intensional view

The operation withdraw in Current extends the one of Account. Its definition and conjectures are *partially generated* from template T15 to give:

$Current_\Delta\ Withdraw$

$\Delta Current$
$Account_\Delta\ Withdraw$

$overdraft' = overdraft$

$\vdash?\ \forall \Delta Current \bullet overdraft \in \mathbb{N}$

$\vdash?\ \exists \text{pre}\ Current_\Delta\ Withdraw \bullet \text{true}$

$\vdash?\ AccountCl \notin abstractCl$
$\quad \Rightarrow \forall(\text{pre}\ Account_\Delta\ Withdraw \Rightarrow \text{pre}\ Current_\Delta\ Withdraw) \bullet \text{true}$

Above, the first conjecture checks well-formedness and is *trivially true* in Z/Eves. The second is the usual precondition consistency conjecture. And the third is the conditional applicability refinement conjecture. The precondition of the operation is simplified with meta-theorems scl-uopx-pre of template T15 and cl-stc-uop-pre of template T10 to give:

$\text{pre}\ Current_\Delta\ Withdraw$

$Current$
$amount? : \mathbb{N}$

$st = active$
$balance + overdraft \geq amount?$

The precondition consistency conjecture (second, above) is *provable* in Z/Eves after the simplification of the precondition. The conditional applicability refinement conjecture is *true by*

construction by appeal to meta-theorem scl-uopx-refa-pa of template T15 because the superclass (Account) is abstract. It is not required to prove the correctness inheritance refinement conjecture because it is always true in both the non-blocking (meta-theorem scl-uopx-refc-nbl) and blocking interpretations (meta-theorem scl-uopx-refc-bl) of refinement, hence this conjecture is not even generated.

Subclass observe operations that are specialisations are also defined by extension. The operation to get the balance in Current is *fully generated* from template T16:

$$Current_{\Xi}GetBalance == \Xi Current \land Account_{\Xi}GetBalance$$

It is not required to prove conjectures for these operations because all the required conjectures are trivially true (see template T16 for further details).

The class Current has its own attribute, overdraft. So there is one operation to set the value of this attribute and another to observe it. The former is *partially-generated* from template T17:

$Current_{\Delta}SetOverdraft$

$$\Delta Current$$
$$AccountH$$
$$\Xi Account$$
$$overdraft? : \mathbb{N}$$

$$overdraft' = overdraft?$$

Note that the history invariant is included.

The well-formedness conjecture is *trivially true* in Z/Eves. The precondition of the operation is simplified with meta-theorem scl-uopn-pre of template T17 and Z/Eves to give:

$$[\ Current;\ overdraft? : \mathbb{N} \mid balance + overdraft? \geq 0\ ]$$

The precondition consistency conjecture is *provable* in Z/Eves after this simplification on the precondition. This operation does not specialise, hence, there are no refinement conjectures to prove.

The operation to observe the state of overdraft is *partially generated* from template T7:

$Current_{\Xi}GetOverdraft$

$$\Xi Current$$
$$overdraft! : \mathbb{N}$$

$$overdraft! = overdraft$$

Here, the well-formedness conjecture is *trivially true*. The precondition gives a true predicate by appeal to meta-theorem cl-stc-oop-pre of template T7. The precondition consistency conjecture is *true by construction* by appeal to meta-theorem cl-stc-oop-epre-np.

The finalisation of subclasses extends the finalisation of the superclass, where the subclass may add constraints of its own. The finalisation of *Current*, subclass of Account, is *fully generated* from template T18:

$CurrentFin$

$$AccountFin$$

The consistency conjecture is easily *provable* in Z/Eves after the simplification given by meta-theorem scl-fin of template T18. The finalisation refinement conjecture is not required because it is always true by meta-theorem scl-fin-ref.

Class Current, extensional view

The new intermediate promotion frame for Current is *fully generated* from template T26:

$\Phi\mathbb{S}CurrentNI_0$

$\Phi\mathbb{S}AccountNI_0[oCurrent!/oAccount!]$
$\Delta\mathbb{S}Current$
$Current\ '$
$oCurrent! : \mathbb{O}AccountCl$

$sCurrent' = sCurrent \cup \{oCurrent!\}$
$stCurrent' = stCurrent \cup \{oCurrent! \mapsto \theta Current\ '\}$

The final new promotion frame, *fully generated* from template T27, extends the intermediate frame with the frame's constraint:

$\Phi\mathbb{S}CurrentNI$

$\Phi\mathbb{S}CurrentNI_0$

$oCurrent! \in \mathbb{O}_x CurrentCl \setminus sCurrent$

The new Current operation and conjectures are *fully generated* from template T28:

$\mathbb{S}_\Delta CurrentNew == \exists\ Current\ ' \bullet \Phi\mathbb{S}CurrentNI \land CurrentInit$
$\vdash?\ CurrentCl \notin rootCl \land CurrentCl \notin abstractCl$

The well-formedness conjecture is *trivially true* in Z/Eves (Current is neither a root nor abstract). The precondition of the operation is obtained by instantiating meta-theorem scl-ext-nop-epre of template T28 and the precondition meta-theorems of the frames to give:

$[\ \mathbb{S}Account;\ \mathbb{S}Current;\ CurrentInit_I \mid \mathbb{O}_x CurrentCl \setminus sCurrent \neq \varnothing\ ]$

It is not required to prove the precondition consistency conjecture because it is *always true* (meta-theorem scl-ext-nop-epre).

The intermediate update promotion frame for Current is *fully generated* from template T30:

$\Phi\mathbb{S}CurrentUI_0$

$\Phi\mathbb{S}AccountUI_0[oCurrent?/oAccount?]$
$\Delta\mathbb{S}Current$
$\Delta Current$
$oCurrent? : \mathbb{O}CurrentCl$

$sCurrent' = sCurrent$
$stCurrent' = stCurrent \oplus \{oCurrent? \mapsto \theta Current\ '\}$

The final subclass promotion frame for Current is *fully generated* from template T31:

$\Phi\mathbb{S}CurrentUI$

$\Phi\mathbb{S}CurrentUI_0$

$oCurrent? \in sCurrent \cap \mathbb{O}_x CurrentCl$

$\theta Current = stCurrent\ oCurrent?$

The update subclass operations withdraw and deposit of Account are *fully generated* from template T32:

$$\mathbb{S}_\Delta CurrentWithdraw == \exists\ \Delta Current \bullet \Phi\mathbb{S}CurrentUI \land Current_\Delta Withdraw$$
$$\mathbb{S}_\Delta CurrentDeposit == \exists\ \Delta Current \bullet \Phi\mathbb{S}CurrentUI \land Current_\Delta Deposit$$

The preconditions of these operations are obtained with meta-theorem scl-ext-uop-pre of template T32 and precondition meta-theorems of the subclass update frames. The precondition of withdraw reduces to:

$$[\ \mathbb{S}Account;\ \mathbb{S}Current;\ oCurrent? : \mathbb{O}CurrentCl;\ amount? : \mathbb{N} \mid$$
$$oCurrent? \in sCurrent \cap \mathbb{O}_x CurrentCl \land (sCurrent\ oCurrent?).st = active$$
$$\land\ (sCurrent\ oCurrent?).balance - amount? + (sCurrent\ oCurrent?).overdraft \geq 0\ ]$$

The precondition of deposit reduces to:

$$[\ \mathbb{S}Current;\ oCurrent? : \mathbb{O}CurrentCl;\ amount? : \mathbb{N} \mid oCurrent? \in sCurrent \cap \mathbb{O}_x CurrentCl\ ]$$

The precondition consistency conjectures for these operations are always true by meta-theorem scl-ext-uop-epre of template T32.

The intermediate delete promotion frame for Current is fully generated from template T38:

$\Phi\mathbb{S}CurrentDI_0$

$\Phi\mathbb{S}AccountDI_0[oCurrent?/oAccount?]$

$\Delta\mathbb{S}Current$

$Current$

$oCurrent? : \mathbb{O}CurrentCl$

$sCurrent' = sCurrent \setminus \{oCurrent?\}$

$stCurrent' = \{oCurrent?\} \lhd stCurrent$

The final delete promotion frame is *fully generated* from template T39:

$\Phi\mathbb{S}CurrentDI$

$\Phi\mathbb{S}CurrentDI_0$

$oCurrent? \in sCurrent \cap \mathbb{O}_x CurrentCl$

$\theta Current = stCurrent\ oCurrent?$

The delete operation on Current is *fully generated* from template T40 to give:

$$\mathbb{S}_{\Delta}\, CurrentDelete == \exists\ Current \bullet \Phi\mathbb{S}CurrentDI \land CurrentFin$$

The well-formedness conjecture is *trivially true* in Z/Eves. The precondition of the operation is obtained by instantiating meta-theorem scl-ext-dop-pre of template T40 and precondition meta-theorems in the delete frames to give:

$$[\ \mathbb{S}Account;\ \mathbb{S}Current;\ oCurrent? : \mathbb{O}CurrentCl \mid oCurrent? \in sCurrent \cap \mathbb{O}_x\, CurrentCl \land (stCurrent\ oCurrent?).balance = 0\ ]$$

The precondition consistency conjecture is not required because it is always true (meta-theorem scl-ext-dop-epre).

Savings, intensional view

The operation withdraw on Savings extends the one of Account; it is *partially generated* from template T15:

Savings$_{\Delta}$ *Withdraw*

$\Delta Savings$

Account$_{\Delta}$ *Withdraw*

The well-formedness conjecture is *trivially true*. The precondition is simplified with meta-theorem scl-uopx-pre of template T15 to give:

$$[\ Savings;\ amount? : \mathbb{N} \mid st = active \land balance - amount? \geq 0\ ]$$

After this simplification, the precondition consistency conjecture is easily *provable* in Z/Eves. The remaining conjectures are as in the intension of Current (see above).

A function that gives the date when an interest credit is due is also required. This is *partially generated* from template T4; it takes the date of the last interest credit and the term of the account and returns the next date an interest credit is due:

$$nxtCredDt : (DATE \times TERM) \rightarrow DATE$$

The actual operation to deposit interest uses this function; it is *partially generated* from template T17:

Savings$_{\Delta}$ *PayInterest*

$\Delta Savings$

$AccountH$

$today? : DATE$

$nxtCredDt(interestDt, term) = today?$

$accountNo' = accountNo$

$balance' = balance + (balance * interestR') \operatorname{div} 100$

$interestDt' = today?$

$term' = term$

The input *today* gives the current date. The after state of $interestR'$ is left unspecified it is to be specialised in the subclasses.

The well-formedness conjecture is *trivially true* in Z/Eves. The precondition is simplified with meta-theorem scl-uopn-pre and Z/Eves to give:

$$[\ Savings;\ today? : DATE \mid nxtCredDt(interestDt, term) = today?\ ]$$

The precondition consistency conjecture is *provable* in Z/Eves after this simplification.

Class Savings, extensional view

In this view, the state space, initialisation and subclassing schema for Savings are as in Current. Savings does not have promoted operations that change the state, because it is an abstract class. However, like in Account, we need to define the inheritance promotion frames (see appendix F.2 for details).

Class WBased, intensional view

The operation withdraw extends the operation with the same name on Savings; this is *partially generated* from template T15:

$WBased_{\Delta}\, Withdraw$

$\Delta\, WBased$

$Savings_{\Delta}\, Withdraw$

$noWithds' = noWithds + 1$

$interestR' = interestR$

$interestDt' = interestDt$

$term' = term$

The well-formedness conjecture is *trivially true*. The precondition, calculated from meta-theorem scl-uopx-pre of template T15 and precondition meta-theorems of update frames, gives the precondition of the superclass in the predicate. The precondition consistency conjecture is *true by construction* (meta-theorem scl-uopx-epre-np) and, as Savings (superclass) is abstract, the applicability refinement conjecture is also *true by construction* (meta-theorem scl-uopx-refa-pa). The correctness refinement conjectures are always true (meta-theorems scl-uopx-refc-nbl and scl-uopx-refc-bl).

In Wbased, the interest rate may change when money is withdrawn from the account. So, there is a function to calculate the interest to be paid based on the number of account withdrawals performed over the last term. This function takes the number of withdrawals and the current interest rate, and returns the new interest. This is *partially generated* from template T4:

$$\mid\ calcInterestW : (\mathbb{N} \times PERCENTAGE) \rightarrow PERCENTAGE$$

WBased extends operation payInterest of Savings, by specifying the way interest rate is calculated on WBased accounts. The operation schema for payInterest is *partially generated* from template T15:

$WBased_{\Delta} PayInterest$

$\Delta WBased$
$Savings_{\Delta} PayInterest$

$interestR' = calcInterestW(noWithds, interestR)$
$noWithds' = 0$

The well-formedness conjecture is *trivially true*. The precondition is as in Savings (simplified with meta-theorem scl-uopx-pre). The precondition consistency conjecture is *true by construction* (meta-theorem scl-uopx-epre-np). As Savings is abstract, the applicability refinement conjecture is also *true by construction* by appeal to meta-theorem scl-uopx-refa-pa.

Class WBased, extensional view

The definitions of this view are as in Current (see appendix F.2.7 for further details).

Class BalBased, intensional view

Like Wbased, BalBased also requires a function to calculate new values of the interest rate, which in BalBased depends on the current balance. So, there is a function that takes the current balance and interest rate and returns a new interest rate; this is *partially generated* from template T4:

$$calcInterestBal : (\mathbb{Z} \times PERCENTAGE) \rightarrow PERCENTAGE$$

This function is used in the operation PayInterest, which is *partially generated* from template T15 to give:

$BalBased_{\Delta} PayInterest$

$\Delta BalBased$
$Savings_{\Delta} PayInterest$

$interestR' = calcInterestBal(balance, interestR)$

The proof of the conjectures and the precondition is as the operation with the same name in WBased (see above). See appendix F.2.7 for the definition of the remaining operations of class BalBased.

Class BalBased, extensional view

The definitions of this view are as in Current (see appendix F.2.7 for further details).

Polymorphic Savings

Polymorphic operations are built in a bottom-up fashion. The class Savings is abstract so its update operations offer a choice of the appropriate operations in its subclasses. The operation withdraw, *fully generated* from template T44, is:

$$\begin{aligned}\mathbb{S}_{\Delta} SavingsWithdraw == {} & \mathbb{S}_{\Delta} WBasedWithdraw[oSavings?/oWBased?] \land \Xi\mathbb{S}BalBased \\ & \lor \mathbb{S}_{\Delta} BalBasedWithdraw[oSavings?/oBalBased?] \land \Xi\mathbb{S}WBased\end{aligned}$$

The well-formedness conjecture for this operation, $\vdash ?\ SavingsCl \in abstractCl$, is *trivially true* in Z/Eves. The precondition of this operation is simplified with meta-theorem acl-ext-uop-poly-pre of template T44 to give:

$$
\begin{array}{l}
[\ \mathbb{S}Savings;\ \mathbb{S}WBased;\ \mathbb{S}BalBased;\ oSavings? : \mathbb{O}SavingsCl;\ amount? : \mathbb{N} \mid \\
\quad oSavings? \in sWBased \cap \mathbb{O}_x\, WBasedCl \\
\qquad \land\ (stWBased\ oSavings?).st = active \\
\qquad \land\ (stWBased\ oSavings?).balance - amount? > 0 \\
\quad \lor\ oSavings? \in sBalBased \cap \mathbb{O}_x BalBasedCl \\
\qquad \land\ (stBalBased\ oSavings?).st = active \\
\qquad \land\ (stBalBased\ oSavings?).balance - amount? > 0\ ]
\end{array}
$$

The precondition consistency conjecture is always true by meta-theorem acl-ext-uop-poly-epre of template T44.

It is not possible to build polymorphic operations of type new. To create some object its class must be explicitly mentioned. In fact, ZOO's new operations return an object to the environment, rather than receiving some object as input, and so their disjunction yields a non-deterministic operation. However, it is possible to create polymorphic-like operations of type new. This requires an input indicating the class of the object to create; the type $ACCTY$ (defined above) serves this purpose. It is also required to enforce a disjunction with an explicit precondition based on this type. A polymorphic-like operation to create a new savings account is *partially generated* from template T43:

$$
\begin{array}{l}
CondNWBased == [\ accTy? : ACCTY \mid accTy? = wbased\ ] \\
CondNBalBased == [\ accTy? : ACCTY \mid accTy? = balbased\ ] \\
\mathbb{S}_\Delta SavingsNew == CondNWBased \land \mathbb{S}_\Delta WBasedNew \land \Xi\mathbb{S}BalBased \\
\qquad\qquad\qquad\qquad \lor\ CondNBalBased \land \mathbb{S}_\Delta BalBasedNew \land \Xi\mathbb{S}WBased
\end{array}
$$

The precondition is simplified with meta-theorem acl-ext-nop-poly-pre to give:

$$
\begin{array}{l}
[\ \mathbb{S}BalBased;\ \mathbb{S}WBased;\ BalBasedInit_I;\ WBasedInit_I;\ accTy? : ACCTY \mid \\
\quad \mathbb{O}_x BalBasedCl \setminus sBalBased \neq \varnothing \land accTy? = balbased \\
\quad \lor\ \mathbb{O}_x\, WBasedCl \setminus sWBased \neq \varnothing \land accTy? = wbased\ ]
\end{array}
$$

It is not required to prove the precondition consistency conjecture (meta-theorem acl-ext-nop-poly-epre).

Delete operations can be made polymorphic by forming the disjunction of the subclass operations. The operation delete on Savings is *fully generated* from template T45:

$$
\begin{array}{l}
\mathbb{S}_\Delta SavingsDelete == \mathbb{S}_\Delta WBasedDelete \land \Xi\mathbb{S}BalBased \\
\qquad\qquad\qquad\qquad \lor\ \mathbb{S}_\Delta BalBasedDelete \land \Xi\mathbb{S}WBased
\end{array}
$$

The precondition is the disjunction of the operation's preconditions (meta-theorem acl-ext-dop-poly-pre). It is not required to prove the precondition consistency conjecture (meta-theorem acl-ext-dop-poly-epre).

Polymorphic Account

The polymorphic operation withdraw on Account is *fully generated* from template T44:

$$\mathbb{S}_\Delta AccountWithdraw == \mathbb{S}_\Delta CurrentWithdraw[oAccount?/oCurrent?] \land \Xi\mathbb{S}Savings \\ \lor \mathbb{S}_\Delta SavingsWithdraw[oAccount?/oSavings?] \land \Xi\mathbb{S}Current$$

The precondition is simplified with meta-theorem acl-ext-uop-poly-pre of template T44 to give:

$$[\ \mathbb{S}Current;\ \mathbb{S}WBased;\ \mathbb{S}BalBased;\ oAccount? : \mathbb{O}AccountCl;\ amount? : \mathbb{N} \mid \\ oAccount? \in sCurrent \cap \mathbb{O}_x CurrentCl \\ \land\ (sCurrent\ oAccount?).st = active \\ \land\ (stCurrent\ oAccount?).balance - amount? + (stCurrent\ oAccount?).overdraft > 0 \\ \lor\ oAccount? \in sWBased \cap \mathbb{O}_x WBasedCl \\ \land\ (sWBased\ oAccount?).st = active \\ \land\ (stWBased\ oAccount?).balance - amount? > 0 \\ \lor\ oAccount? \in sBalBased \cap \mathbb{O}_x BalBasedCl \\ \land\ (sBalBased\ oAccount?).st = active \\ \land\ (stBalBased\ oAccount?).balance - amount? > 0\ ]$$

The precondition consistency conjecture is always true by meta-theorem acl-ext-uop-poly-pre of template T44.

The polymorphic operation to create a new account is *fully generated* from template T43:

$$CondAccCurrent == [\ accTy? : ACCTY \mid accTy? = current\ ] \\ \mathbb{S}_\Delta AccountNew == CondAccCurrent \land \mathbb{S}_\Delta CurrentNew \land \Xi\mathbb{S}Savings \\ \lor \mathbb{S}_\Delta SavingsNew \land \Xi\mathbb{S}Current$$

The precondition obtained from meta-theorem acl-ext-nop-poly-pre of template T43 gives:

$$[\ \mathbb{S}Current;\ CurrentInit_I;\ accTy? : ACCTY \mid \\ \mathbb{O}_x CurrentCl \setminus sCurrent \neq \varnothing \land accTy? = current\ ] \\ \lor \text{pre}\ \mathbb{S}_\Delta SavingsNew$$

Again, it is not required to prove the precondition consistency conjecture (meta-theorem acl-ext-nop-poly-epre).

The operation delete on Account is similarly defined. See appendix F.2.9 for further details.

5.7.3 Global View

Global operations are specified in the usual way. Operation frames (Ψ frames) are conjoined with local operations (defined in extensional and relational views) to make a system operation. Most global operations are as defined in the previous chapter. Those that require some change or that are new are presented here. The preconditions of the global operations for the Bank system are documented in table 5.3.

The system operation OP2 (*Open a new account*) needs to be modified because the operation condition expressing system constraint C6 is affected by the new structure; the new formulation of this constraint is *partially generated* from template T55:

OP1	$\mathbb{O}_x\, CustomerCl \setminus sCustomer \neq \varnothing$
OP2	$oCustomer? \in sCustomer$ $accTy? = current \Rightarrow \mathbb{O}_x\, CurrentCl \setminus sCurrent \neq \varnothing$ $accTy? = wbased \Rightarrow \mathbb{O}_x\, WBasedCl \setminus sWBased \neq \varnothing$ $\quad opAmount? \geq reqOpAmount\ wbased$ $accTy? = balbased \Rightarrow \mathbb{O}_x\, BalBasedCl \setminus sBalBased \neq \varnothing$ $\quad opAmount? \geq reqOpAmount\ balbased$ $accTy? = wbased \lor accTy? = balbased \Rightarrow$ $\quad (stCustomer\ oCustomer?).ctype \neq company$ $\quad oCustomer? \in rHolds^{\sim}(\!|\ sCurrent\ |\!)$
OP3	$oAccount? \in sAccount?$
OP4	$oAccount? \in sAccount \land (stAccount\ oAccount?).st = active$ $\Sigma\{\ a : sAccount \bullet a \mapsto (stAccount\ a).balance\ \} \geq amount?$ $\quad oAccount? \in sCurrent$ $\quad\quad \land\ (stCurrent\ oAccount?).balance + (stCurrent\ oAccount?).overdraft \geq amount?$ $\quad \lor\ oAccount? \in sSavings \land (stSavings\ oAccount?).balance \geq amount?$
OP5	$oAccount? \in sAccount$
OP6	$true$
OP7	$true$
OP8	$oAccount? \in sAcconut \land (stAccount\ oAccount?).balance = 0$
OP9	$oAccount? \in sAccount \land (stAccount\ oAccount?).st = active$
OP10	$oAccount? \in sAccount \land (stAccount\ oAccount?).st = suspended$
OP11	$oCurrent? \in sCurrent \land (stCurrent\ oCurrent?).balance + overdraft? \geq 0$
OP12	$oSavings? \in sSavings$ $\land\ nxtCredDt((stSavings\ oSavings?).interestDt, (stSavings\ oSavings?).term) = today?$
OP13	$oSavings? \in sSavings$

Table 5.3: Preconditions of system operations (calculated with meta-theorems and Z/Eves).

OpConstIfSavingsHasCurrent

$\mathbb{S}Customer$; $\mathbb{S}Current$
$\mathbb{A}Holds$
$accTy? : ACCTY$
$oCustomer? : \mathbb{O}CustomerCl$

$\neg\ accTy? = current \Rightarrow oCustomer? \in rHolds^{\sim}(\!|\ sCurrent\ |\!)$

The actual system operation OP2 is *partially generated* from template T59 to give:

$$\Psi OpenAccount == \Delta System \land \Xi\mathbb{S}Customer$$
$$SysOpenAccount == \Psi OpenAccount \land OpConstIfSavingsHasCurrent \land \mathbb{S}_{\Delta} AccountNew \land \mathbb{A}_{\Delta} HoldsAdd[oAccount!/oAccount?]$$

The conjectures are proved as described in the previous chapter. The precondition is given in table 5.3.

The system operation OP11, to set the overdraft of a current account is *partially generated* from template T60:

$$\Psi CurrentOps == \Psi AccountOps \land \Xi\mathbb{S}Savings$$
$$SysSetOverdraft == \Psi CurrentOps \land \mathbb{S}_{\Delta} CurrentSetOverdraft$$

Here, the well-formedness conjecture (see template T60) is trivially true in Z/Eves. The consistency conjecture is provable in Z/Eves after the simplification given by meta-theorem sys-op-uo-pre of template T60.

System operations OP12 (deposit interest) and OP13 (change credit term) are also generated from template T60 (see appendix F.2.11 for details). Their consistency conjectures are provable in Z/Eves after the simplification given by the meta-theorem sys-op-uo-pre of template T60. The preconditions of these operations are given in table 5.3.

This completes the ZOO model of operations, which captures all the information given by the UML class and state diagrams and more (the full specification of operations). It captures all required system operations and remaining system constraints. System constraint constraint $C5$ is expressed locally as the finalisation of Account; constraint $C6$ is expressed globally as a condition of system operation $OP2$ (open account).

5.7.4 Discussion

The Bank inheritance hierarchy was made behavioural conformant by making Account and Savings abstract based on the abstract class relaxation. Without this relaxation, the applicability refinement conjectures for withdraw in Current and Savings would not be provable.

Again, the abstract class relaxation was essential to design a flexible inheritance hierarchy. Without it, the overdraft attribute and the associated invariant of Current would have to be moved up in the hierarchy, to *Account*.

With meta-proof, the effort in proving inheritance refinement conjectures becomes minimal. The initialisation and correctness refinement conjectures are all *true by construction*. In some situations the applicability conjecture is also *true by construction*.

It is worth analysing the effect of the global constraints on the local states of objects, to see if those constraints create divergence:

- Constraint C2 (p. 91), constrains the local state of Account objects, so the promotion is no longer free. But this does not create divergence, because that constraint is applicable to all objects of Account, including those that are instances of its subclasses.
- Constraint C3 (p. 138), constraints the internal states of Savings objects. There could be the risk of divergence because Savings is a subclass. In this case, however, we are safe, because the superclass, Account, is abstract, it has no objects of its own: so Savings objects cannot diverge from direct objects of its superclass because they do not exist.

The previous chapter has described the behaviour of Account with statechart (figure 4.7). This chapter introduces class inheritance, and subclasses of Account, which may add extra operations. There is the problem that these extra operations could change the behaviour of the statechart in undesired ways. For example, suppose that the extra operation of Current, setOverdraft, would set the state of Account to active. Then, we could imagine a customer who has his account suspended for some reason, to apply for an overdraft just to have his account active again in order to enable account withdrawals. These situations are avoided with the history invariant, which disables changes of state with respect to what is described in the statechart. This history invariant schema must be extended by each extra operation of the subclasses of Account.

5.8 Discussion

This chapter shows how ZOO can be used to express inheritance and to check behavioural conformance. The Queues case study illustrates ZOO's supports for multiple inheritance. The Bank case study illustrated ZOO's support for inheritance in the context of the $UML + Z$ framework, where the ZOO model is generated from templates of the $UML + Z$ catalogue.

In the intensional view, subclass operations are defined in a simple and modular way: subclass definitions extend their superclass counterparts using schema conjunction. This extension scheme is characterised by a simple retrieve relation (a total function), which allows the derivation of a simpler set of refinement rules (or conjectures) for inheritance refinement. This also helps to simplify inheritance refinement conjectures further by using meta-proof, to the point that the initialisation and correctness refinement conjectures became *true by construction*.

In the extensional view, however, there are some difficulties. Each class needs to keep its own repository of objects, and this impacts on the way promotion frames are built. These are also built in a modular way based on schema extension, but they are more cumbersome and rather heavy, because whenever the extension of the subclass needs to be updated, then all the extensions of its superclasses must also be updated. This is actually a consequence of a problem coming from the intensional view and to the lack of schema subtyping in Z.

Although schemas can be extended nicely with schema conjunction, each schema has its own type and it is not possible to establish subtyping relationships between schemas: Z does not support schema subtyping. This problem can be observed in figure 5.4 (p. 114). The set of object atoms is divided into sets, while the set of objects of the superclass includes the objects of its subclasses. The sets of states (instances of schemas), on the other hand, are disjoint and not related to each other. Schema subtyping (as in the approach for records of Cardelli [Car88]) would allow the set of states of the superclass to include those of its subclasses, which would remove the need for a mapping function in each class extension.

For example, with schema subtyping, the bank example would require just one mapping function for the whole Account hierarchy:

$$stAccount : \mathbb{O} AccountCl \pfun Account$$

This would allow the inclusion of any state that is a subclass of Account, because the subclass intensions would be a subtype of *Account*. This would relieve the burden on promotion frames, because only one mapping needs to be maintained.

Despite this problem, promotion frames are built in a modular fashion and their underlying structure is easily captured with templates. As the Bank example shows, the promotion frame templates are automatically instantiated from templates (they are fully generated). The definitions may seem unnatural, but one does not have to stare too much at them: the complexity is concealed, the specifier does not have to directly deal with it.

In ZOO, the proof burden of inheritance refinement is negligible. The initialisation and correctness conjectures are always *true by construction*. So, it all reduces to the applicability conjecture, but its proof is required only when the superclass is not abstract, and even then, in certain conditions it is *true by construction*, in the others what remains is easily provable in Z/Eves. So, in many cases, a subclassing relation is behavioural conformant by construction.

Again, it is important to emphasise ZOO's relaxations. Without them, it would be difficult to reconcile behavioural inheritance and incremental definition. As we have seen, even with relaxation the addition of new subclass may requires the inheritance structure to be refactored. The relaxations help to keep this to a minimum and all that is required, in most cases, is to make the superclass abstract.

5.9 Related Work

The idea of adding stuttering steps (or skip operations) to relax the restriction of refinement came from Lamport's work on TLA [Lam94]. In [ACM05], Abrial discusses the restrictions of this approach and proposes keep operations (or actions) to overcome them. A keep operation is a non-deterministic operation that is guaranteed to preserve the invariant. Abrial argues that it is safe to add keep operations to abstract types. This is similar to ZOO's *virtual* operations, which are safe because they are not visible to the outside world.

Lupton investigated the compositionality of promotion with respect to refinement [Lup90]. He gives interesting examples that show that, at the promoted level, the concrete type may diverge from the abstract, even though internally they are a refinement. This insight was used in ZOO to anticipate the conditions under which the behaviour of the subclass may diverge from that of its superclass. This may happen whenever there is a global constraint that affects the subclass and not its superclasses, and the superclasses are not abstract.

Liskov and Wing authored the seminal paper in the area of behavioural inheritance [LW94]. Their model of objects is similar to ours: there is a mapping from objects (atoms) to their state. Their approach, however, is based on a earlier method of data refinement [Hoa72b]; ZOO uses data refinement based on simulations [HHS86], which is still the basis of the theory. Liskov and Wing check behavioural conformance of each operation and allow extra operations to be added to the subtype. They do not consider, however, object creation (initialisation) and deletion (finalisation), which is very important because if these are not checked behavioural conformance is not guaranted; subclass objects may diverge from their superclass counterparts. Their rules correspond to ZOO's rule for blocking refinement. Liskov and Wing also

propose rules for history invariants that extra subclass operations must preserve. Inspired by this, ZOO uses history schemas to capture history invariants; these are defined in the superclass and extended in the extra operations of the subclass. The preservation of these history invariants is ensured in ZOO by proving precondition consistency proofs. Moreover, ZOO's simulation relation is fixed, Liskov and Wing allow more flexibility, allowing the relation to be a function. Liskov and Wing mention that "the requirement we impose is very strong and raises the concern that it might rule out many useful subtype relations". Relaxations are not considered.

In the field of Z, Hall's approach [Hal90b, Hal94] has been the major influence. From Hall, ZOO borrows the approach to extend subclass intensional definitions using schema conjunction; Hall also proposed the refinement relation that is used in ZOO, but he did not explore it to simplify the proof rules of general Z refinement. Hall considers neither initialisation nor finalisation of objects in his refinement rules, hence, not guaranteeing behavioural conformance (the behaviour of subclass and superclass objects may diverge). Relaxations are not considered.

Dhara et al [DL96] tried to relax the restrictions imposed by the rules of [LW94]. They propose a relaxation that is equivalent to non-blocking refinement: the subclass operation is allowed to weaken the precondition and provide more behaviours outside the superclass's precondition. ZOO also allows the more relaxed non-blocking refinement.

Wehrheim and Fischer [FW00, Weh00] investigate behavioural subtyping in the context of concurrency and the CSP process algebra [Ros98, Hoa85]. They studied how extra subclass operations may interfere with the behaviour of the superclass as observed from the environment, and under which conditions are safety and liveness properties preserved by the subclasses. They propose several inheritance refinement relations; the more liberal they are, the higher the risk of interference. The one that is closer to ZOO's relaxation on extra operations is *weak subtyping*, which says that the subclass should have the same behaviour as its superclass as long as no extra operations are called; the extra operations are not considered in the comparison. The authors also proposed a more restricted relation, *optimal subtyping*, which does not allow altering the behaviour of the superclass at all; it is the same as the skip behaviour. The authors argue that weak subtyping is appropriate for exclusive access to an object. This is the case with sequential systems, the intended application domain of the $UML + Z$ framework developed in this thesis.

Also in the context of concurrency, Harel and Kupferman [HK02] take a rather different perspective to behavioural inheritance from that of Wehrheim and Fischer. They are concerned with inheritance in the context of a class whose behaviour is described by a statechart. They argue that unlike traditional refinement, where every behaviour of an implementation must also be a behaviour of the specification, in inheritance, on the other hand, the inclusion is in the other direction: the replacement of A by B (B inherits from A) must not involve loss of behaviours. Inheritance, the authors argue, is about "adding stuff". The fact that B may be less abstract than A is captured in the refinement mapping. ZOO uses data refinement to check behavioural conformance and here the appropriate refinement direction is the one of traditional refinement: every behaviour of the subclass should be a behaviour of the superclass. This is what was proposed by Liskov and Wing. But from a process point of view, things may be different. As we have also seen in ZOO, sometimes different refinement relations are required; for polymorphism, the refinement direction is inverted: the superclass must refine the subclass and not the other way around.

Object-Z [Smi00, DR00, DB01] supports inheritance, but a discussion of behavioural con-

formance is often missing in its books. In [DB01], behavioural inheritance and its relation to refinement is discussed, but no proof obligations are proposed to check behavioural conformance. In [Smi00], the queues example is built in Object-Z, but there is no comment on the fact that BQueue is not behaviour conformant with that of Queue.

The Eiffel programming language [Mey92, Mey97] follows the general approach to behavioural inheritance based on data refinement: (a) the precondition may weakened and (b) the postcondition may be strengthened. Meyer also observes that the rule that requires the weakening of the precondition may be too restrictive.

5.10 Conclusions

This chapter presented the ZOO model of inheritance, its approach to check behavioural conformance, and two relaxations to the rules of behavioural conformance. This was illustrated with two cases studies. The first was a case study of queues that emphasised multiple-inheritance and the restrictions imposed by behavioural inheritance. The second case study was an extension to the Bank case study of the previous chapter with an inheritance hierarchy; it illustrated ZOO's support for inheritance in the context of $UML + Z$, where it was shown that a ZOO model with inheritance can be built by instantiating templates of the ZOO catalogue. By using the catalogue's meta-theorems, proofs of behavioural inheritance become almost negligible.

This chapter presented several contributions. ZOO's inheritance style is a substantial improvement from Hall's approach, especially in the specification of operations. ZOO's approach to behavioural inheritance and the relaxations are new. As discussed above, the issue of behavioural inheritance is often overlooked even in languages that are designed to be OO (like Object-Z); the author is not aware of another work where the issues of behavioural inheritance are considered in such depth . The catalogue of templates to generate ZOO models with inheritance is also an important contribution, illustrating the usefulness of FTL and its meta-proof approach.

The next chapter develops an approach to analyse $UML + Z$ models with snapshot diagrams. The $UML + Z$ model of the Bank case study developed here, which has been proved consistent, is going to be analysed: we known that our model is consistent, but is it the right model? The next chapter tries to answer this question by analysing the Bank model against its requirements.

[...] our recent progress in the development of a sound programming methodology should not lead us to ignore the more difficult aspects of engineering; and that in future we should pay more attention to the quality of our designs, and not just the accuracy of their implementation.

Anthony Hoare [Hoa78]

This paper has not addressed the ever-present problem of translating the vaguely felt desires of a set of prospective users into precise specifications and verifying the correctness of that translation.

David Parnas [Par77]

6 Snapshot analysis

The previous two chapters were concerned with the construction of consistent models. They developed the ZOO style and the $UML + Z$ catalogue of FTL templates and meta-theorems, which generate ZOO models and simplify their consistency proofs. ZOO constitutes the semantic domain for UML diagrams in the $UML+Z$ framework; UML diagrams are formalised in ZOO by instantiating templates of the $UML + Z$ catalogue.

$UML + Z$ models are formal (represented in Z) and consistent (proved in Z), and yet they may not be what the customer of the system wants. This chapter is concerned with analysing $UML + Z$ models to know whether they satisfy their requirements. The good thing of having a formal model is that the analysis can be conducted mechanically with the aid of the tool support for the Z language.

This chapter introduces a formal technique to analyse models constructed with the $UML+Z$ framework. This technique is based on Catalysis snapshots [DW98] and formal proof. It is called *snapshot analysis* and constitutes the analysis strategy of the $UML + Z$ framework. The following explains snapshot analysis and illustrates it with the model of the Bank case study developed in the previous chapter.

6.1 The analysis technique

A class diagram denotes a set of instances (figure 6.1). Some of those instances are *valid* (inside the oval): they satisfy the constraints of the modelled system. Others, although also instances of a class diagram, are *invalid* (inside the rectangle, but outside the oval): they do not satisfy the constraints of the model.

The same occurs in the world of Z models. There is the set of all possible instances of a Z model, but only a subset of those are valid. All these possible model instances are type-correct, but only through proof can we check whether they are valid or not. Usually, this involves existence proofs: we prove that there is some object with the required properties corresponding to the given instance. These checks need to resort to proof, because, in general, the problem of checking whether an instance of some Z definition satisfies its constraints is

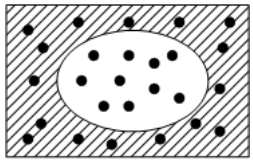


Figure 6.1: The sets of all possible instances of a $UML + Z$ model (rectangle), and all valid instances (oval).

undecidable (that is, it is impossible to build a tool that automatically does this). This is why Z users are required to prove consistency theorems, the *type-checker* cannot possibly check this for them, and they may use a proof tool, but the tool cannot prove everything automatically for them.

A snapshot is an object diagram, an instance of a class diagram [DW98]. In $UML + Z$, snapshots are represented as an instance of a ZOO model. Each snapshot of some class diagram represents a point (an instance) in the diagram of figure 6.1. Some are valid (in the oval), others are invalid (in the rectangle, but outside the oval). Those that are valid represent actual states of the modelled system.

Figure 6.1 gives a basis for model analysis: the instance space can be used to explore (or test) the model. The idea is to draw snapshots that describe instances of the model, and then check whether those instances are valid or not. This gives useful insight into the model. The advantage of having a formal model is that snapshot-validity can be checked mechanically through proof (but not automatically) with the aid of Z provers.

The analysis can either be *positive* or *negative*. It is positive when the valid instance space (inside the oval) is explored; this confirms that the model works as expected, otherwise there is something wrong. It is negative when the invalid instance space (within the rectangle, but outside the oval) is explored; if a negative snapshot turns out to be valid, then there is something wrong.

The validity of snapshots is checked through existence proofs. For positive snapshots, it is proved that there exists a system state as described by the snapshot. This is negated for negative snapshots, thus proving that such a state cannot possibly exist.

The obvious application of the technique is to use single snapshots to explore the state space of the modelled system. But there is more to snapshots than just analysis of state space.

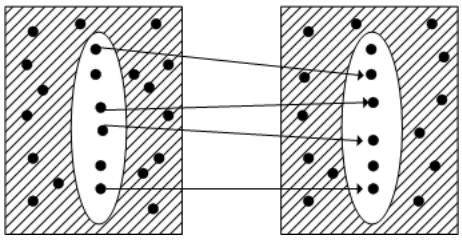


Figure 6.2: An operation is a relation between sets of valid model instances.

Operations perform state transitions. They take a system from one state into another. At this abstract level, an operation is a relation between sets of states (figure 6.2). This is how operations are represented in Z, where the relation may be partial: only a subset of the state space enables a state transition (those states that are in the domain of the relation, the operation's pre-condition). As figure 6.2 suggests, snapshots can also be used to analyse operations. This is done with *snapshot-pairs*: one snapshot represents the state before the execution of the operation (*before-state*), and the other the state af-

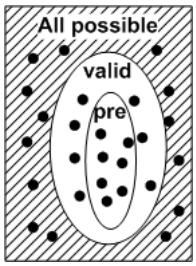


Figure 6.3: All possible model instances, valid instances, and valid instances that satisfy the precondition of some operation.

terwards (*after-state*). A snapshot-pair is valid if the *before-snapshot* is a valid system instance that satisfies the precondition of the operation, and if the after state describes a correct transition from the before state with respect to the operation (figure 6.3). This is also checked through existence proofs.

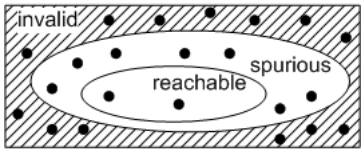


Figure 6.4: Invalid, spurious and reachable instances of a model.

In snapshot analysis, there is, however, an issue with *reachability*. Figure 6.1 shows us that there are valid and invalid instances of a model of a system, but there also instances, which, although valid, are not accepted by the model because they cannot possibly be reached by using the operations of the system. For example, if a system would just have the operation of figure 6.2, then it is clear that there are states that are not reachable. In terms of figure 6.1, there is another set inside the oval, representing those states that are reachable by using the operations of the system; the ones outside are valid, but not reachable — they are not accepted by the model and are called *spurious* (figure 6.4). The problem with spurious instances is that the results of the analysis may be misleading: a negative snapshot may turn out to be valid and we may wrongly conclude that there is something wrong with the model, when, in the end, it may well be that the snapshot is not accepted because it is spurious. This may occur especially when doing single snapshot analysis, but it may also happen with snapshot-pairs, by using a spurious instance as a before state.

This chapter proposes snapshot-sequences to investigate the effect of a combination of operation executions. This may be used to investigate reachability. A snapshot-sequence is a sequence of snapshot pairs, where the after state of one pair is the before-state of the next. We may study reachability by departing from the initial state of the system (or a state that we know for sure that is reachable) and describing a sequence of operation executions. A snapshot-sequence is valid if every pair is valid, and invalid otherwise.

The technique is now illustrated with the model of the *Bank* case study.

6.2 Snapshot analysis of the Bank case study

Snapshot analysis is illustrated with the Bank case study developed in the previous chapter. The analysis draws snapshots that are instances of the *Bank* class diagram of figure 5.6 (p.131). First, the state space is analysed with single snapshots. Then, individual system

operations are analysed with snapshot pairs. And, finally, combinations of executions of system operations are analysed with snapshot sequences. Validity proofs are conducted in the Z/Eves theorem prover [Saa97]. The analysis modifies the model, but this affects only the *global view*. The modified model is given in appendix F.3.

6.2.1 Analysis of State space with single snapshots

Single snapshots are used to explore the state. Analysis then checks whether the snapshot is a valid instance of the system model.

Figure 6.5 presents the first snapshot of our analysis. Boxes represent objects of some class, the lines connecting boxes represent links (or tuples) of some association. This snapshot describes a personal customer (oC1) who holds a current account (oAcC1) with the Bank. This snapshot is positive, but its validity needs to be checked formally.

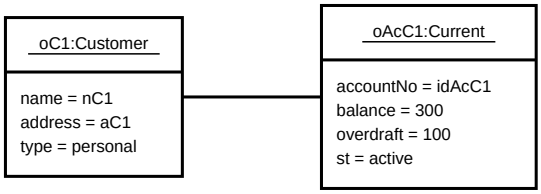


Figure 6.5: Snapshot of state. A personal customer with a current account.

Templates of the ZOO catalogue capture the ZOO representation of snapshots and required conjectures. This is *fully generated* for this snapshot from template T65:

$$
\begin{array}{|l}
idAcC1 : ACCID \\
nC1 : NAME \\
aC1 : ADDRESS \\
oC1 : \mathbb{O}_x\, CustomerCl \\
oC1St : Customer \\
oAcC1 : \mathbb{O}_x\, CurrentCl \\
AcC1St : Current \\
\hline
\mathbb{O}_x\, CustomerCl = \{oC1\} \land \#\mathbb{O}_x\, CustomerCl = \#\langle oC1\rangle \\
\mathbb{O}_x\, CurrentCl = \{oAcC1\} \land \#\mathbb{O}_x\, CurrentCl = \#\langle oAcC1\rangle \\
oC1St = \langle\!|\ name == nC1, address == aC1, type == personal\ |\!\rangle \\
oAcC1St = \langle\!|\ accNo == idAcC1, balance == 300, overdraft == 100, st == active\ |\!\rangle
\end{array}
$$

*StSnap*1

System

$$sCustomer = \{oC1\} \land stCustomer = \{oC1 \mapsto oC1St\}$$
$$sCurrent = \{oAcC1\} \land stCurrent = \{oAcC1 \mapsto oAcC1St\}$$
$$sAccount = \{\} \cup sCurrent \land stAccount = \{\} \cup stCurrent \, ⨾ \, (\lambda \, Current \bullet \theta Account)\}$$
$$sSavings = \varnothing \land stSavings = \varnothing$$
$$sWBased = \varnothing \land stWBased = \varnothing$$
$$sBalBased = \varnothing \land stBalBased = \varnothing$$
$$rHolds = \{oC1 \mapsto oAcC1\}$$

$\vdash ?\ \exists \, StSnap1 \bullet \text{true}$

The ZOO representation is divided into three parts (one for each Z paragraph): object definitions, instance and conjecture. The first defines names of objects and their states. The second defines the instance of the ZOO model described by the snapshot. Finally, the conjecture says what is required to prove to check whether the snapshot is valid.

This snapshot is positive, it should represent a valid state of the system. So, the generated conjecture requires a proof that such a state does exist. Note that the conjecture is similar to the initialisation conjectures used in the previous chapters, where it was proved that the initial state was a valid instance of some state space definition. Here, it is proved that the state described by the snapshot is a valid instance of the system state space. The conjecture is easily provable in Z/Eves — the snapshot is a valid state of the system.

Next, the analysis focuses on testing the constraints of the *Bank* system.

6.2.2 Analysing system constraints

Constraint C2, *total balances must not be negative.* Figure 6.6 is a negative snapshot. It describes a system where there is one sole Bank customer who holds a current account with a negative balance. This violates constraints C2 because the total balances are negative. Note that this snapshot does not violate constraint C7 because the negative balance is below the allowed overdraft. As the snapshot should be invalid, the conjecture is negated:

$\vdash ?\ \neg \ (\exists \, StSnap2 \bullet \text{true})$

As expected, this is provable — the snapshot is invalid, it is not accepted by the model.

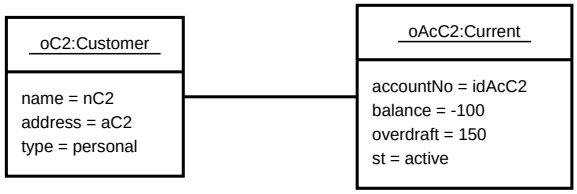


Figure 6.6: An invalid snapshot: the total balances are negative (constraint C2).

Constraint C1, *savings accounts must not have negative balances.* The snapshot of figure 6.7 violates constraint C1 (negative). It describes one customer with one Current and one WBased (savings) account, where the WBased account has a negative balance. Note that the snapshot does not violate constraint C2, because the total balances are greater than 0. As expected, the negative conjecture is easily provable — the model does not accept the snapshot.

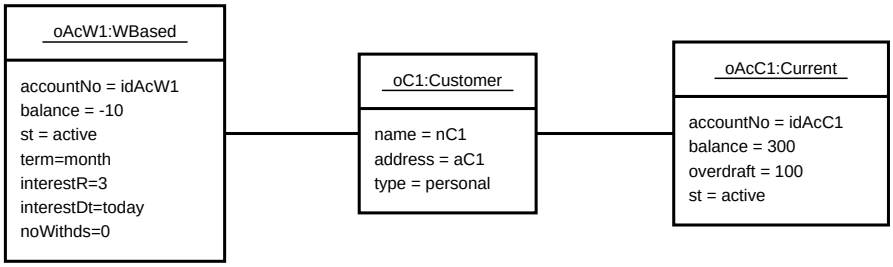


Figure 6.7: A savings account with a negative balance (constraint C1).

Constraint C7, *Negative balance of current accounts must not go above allowed overdraft.* This is tested with the negative snapshot of figure 6.8, which describes two customers, each with its own Current account, where one of these accounts is overdrawn beyond the allowed overdraft. Again, constraint C2 is not breached. As expected, the negative conjecture is provable — the snapshot is not accepted by the model.

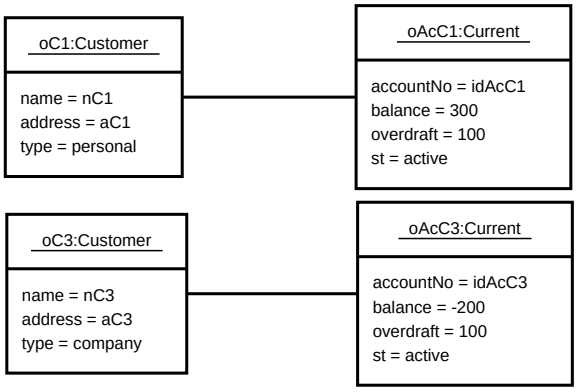


Figure 6.8: A current account with a negative balance above allowed overdraft (constraint C2).

Constraint C3, *Companies cannot have savings accounts.* The snapshot of figure 6.9 violates constraint C3, because it describes a company Customer with one Current and one BalBased account. As expected, the negative conjecture is provable — the snapshot is not accepted by the model.

The analysis now investigates some interesting questions posed by the snapshots above.

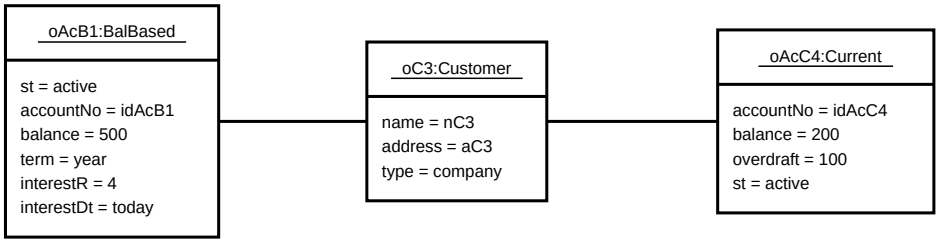


Figure 6.9: Snapshot of an invalid state: a company with a savings account.

The power of visualisation

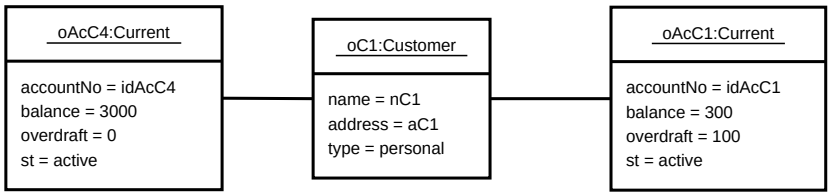


Figure 6.10: Snapshot of a customer with two current accounts.

The snapshot of figure 6.10 is prompted by the previous three snapshots. It describes a customer holding two current accounts. Analysis shows that this snapshot is valid (positive conjecture is provable) — but it cannot be accepted because customers may have at most one current account. This could be clarified with the customer of the system.

It turns out that a constraint is missing from the requirements of the Bank system:

C12	A customer may have at most one current account.

Consequently, the constraint was also missing in the model. It is now formalised, in the global view of the ZOO model, as:

ConstOneCurrent

$\mathbb{S}\,Customer;\ \mathbb{S}\,Current;\ \mathbb{A}\,Holds$

$\mathsf{mult}(rHolds, sCustomer, sCurrent, ozo, \{\}, \{\})$

This says that the multiplicity of the association Holds for the set of Current accounts must be *one to zero or one*: a customer may have at most one account. This is formalised using the mult generic of the ZOO toolkit that is used to express multiplicity constraints of associations — the Current account specialises the association that is inherited from Account.

The modified model is still consistent (all required consistency proofs of the previous chapter are provable). This new constraint changes, however, the precondition of operation OP2, *Open Account*, to (see table 5.3, p. 149 for old precondition):

$$
\begin{array}{l}
oCustomer? \in sCustomer \\
accTy? = current \Rightarrow \\
\quad \mathbb{O}_x\, CurrentCl \setminus sCurrent \neq \varnothing \wedge oCustomer \notin rHolds^{\sim}(\!|\, sCurrent \,|\!) \\
accTy? = wbased \Rightarrow \\
\quad \mathbb{O}_x\, WBasedCl \setminus sWBased \neq \varnothing \\
\quad opAmount? \geq reqOpAmount\ wbased \\
accTy? = balbased \Rightarrow \mathbb{O}_x\, BalBasedCl \setminus sBalBased \neq \varnothing \\
\quad opAmount? \geq reqOpAmount\ balbased \\
accTy? = wbased \vee accTy? = balbased \Rightarrow \\
\quad (stCustomer\ oCustomer?).ctype \neq company \\
\quad oCustomer? \in rHolds^{\sim}(\!|\, sCurrent \,|\!)
\end{array}
$$

All the required conjectures for the snapshots above are still provable in the modified model. For the snapshot of figure 6.10, the positive conjecture is false and the negative conjecture is provable — the snapshot is now invalid, the modified model does not accept it.

This ends the analysis of the state space. System constraints C1, C2, C3 and C7 have been analysed, and they work as expected. Analysis found that there was a constraint missing from the requirements, which was added as constraint C12.

6.2.3 Analysing of operations with snapshot pairs

Operations are analysed using snapshot-pairs. The first snapshot describes the state of the system before the execution of the operation (*before-state*), and the second the state afterwards (*after-state*). Analysis checks whether the snapshot-pair describes a valid system transition with respect to some system operation. This involves two conjectures: the first checks that the before-snapshot is a valid model instance that satisfies the operation's precondition; the second checks that whole snapshot-pair describes a valid transition with respect to the operation. A snapshot is valid if both positive conjectures are provable, and invalid otherwise (or if the negative conjectures are provable). One conjecture would suffice to check validity; the first conjecture (precondition) is redundant, but we add it because this makes errors easy to trace: if the precondition conjecture fails then there is an error related to the before-snapshot, and if it is the transition conjecture that fails then the problem is related to the after-snapshot.

Analysis now tests some of the operations of the Bank system.

Operation OP1, *New Customer***.** Figure 6.11 is the first snapshot-pair of the analysis. The before-snapshot is above the separating line, and the after below (changes are highlighted); the ellipsis indicates the operation being analysed and the inputs to the operation. This pair describes a positive transition. In the before snapshot, there is one Bank customer (oC1); the operation requests the creation of a new customer, and in the after state there is a new customer object (oC2).

The ZOO representation of this snapshot is *fully generated* from template T66:

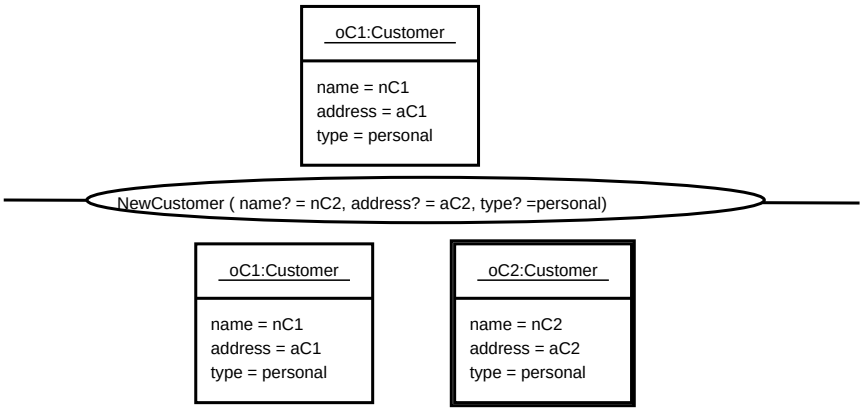


Figure 6.11: Snapshot-pair: a customer is added to the system.

$$
\begin{array}{|l}
nC1 : NAME \\
aC1 : ADDRESS \\
oC1, oC2 : \mathbb{O}_x\, CustomerCl \\
oC1St, oC2St : Customer \\
\hline
\mathbb{O}_x\, CustomerCl = \{oC1, oC2\} \land \#\mathbb{O}_x\, CustomerCl = \#\langle oC1, oC2\rangle \\
oC1St = \langle\!|\ name == nC1, address == aC1, type == personal\ |\!\rangle \\
oC2St = \langle\!|\ name == nC2, address == aC2, type == personal\ |\!\rangle
\end{array}
$$

*BOpSnap*1

$$
\begin{array}{|l}
System \\
\hline
sCustomer = \{oC1\} \\
stCustomer = \{oC1 \mapsto oC1St\} \\
sCurrent = \varnothing \land stCurrent = \varnothing \\
sAccount = \varnothing \land stAccount = \varnothing \\
sSavings = \varnothing \land stSavings = \varnothing \\
sWBased = \varnothing \land stWBased = \varnothing \\
sBalBased = \varnothing \land stBalBased = \varnothing \\
rHolds = \varnothing
\end{array}
$$

*IOpSnap*1

$$
\begin{array}{|l}
name? : NAME \\
address? : ADDRESS \\
type? : CUSTTYPE \\
\hline
name? = nC1 \\
address? = aC1 \\
type? = personal
\end{array}
$$

$$
\begin{array}{|l}
\hline
AOpSnap1 \\
System \\
\hline
sCustomer = \{oC1, oC2\} \land stCustomer = \{oC1 \mapsto oC1St, oC2 \mapsto oC2St\} \\
sCurrent = \varnothing \land stCurrent = \varnothing \\
sAccount = \varnothing \land stAccount = \varnothing \\
sSavings = \varnothing \land stSavings = \varnothing \\
sWBased = \varnothing \land stWBased = \varnothing \\
sBalBased = \varnothing \land stBalBased = \varnothing \land rHolds = \varnothing \\
\hline
\end{array}
$$

$\vdash ?\ \exists \operatorname{pre} SysNewCustomer \bullet BOpSnap1 \land IOpSnap1$

$\vdash ?\ \exists SysNewCustomer \bullet BOpSnap1 \land IOpSnap1 \land AOpSnap1\,'$

Above, the first Z paragraph (axiomatic definition) defines objects and their states. The second (schema) defines the *before* instance. The third defines the inputs to the operation. The fourth defines the *after* instance. Finally, there are two conjectures; the first checks that the before state is a valid system instance that satisfies the operation's precondition; the second checks that the snapshot-pair describes a valid system instance with respect to the operation.

Both conjectures are provable in Z/Eves — the snapshot pair describes a valid system transition for the *New customer* system operation.

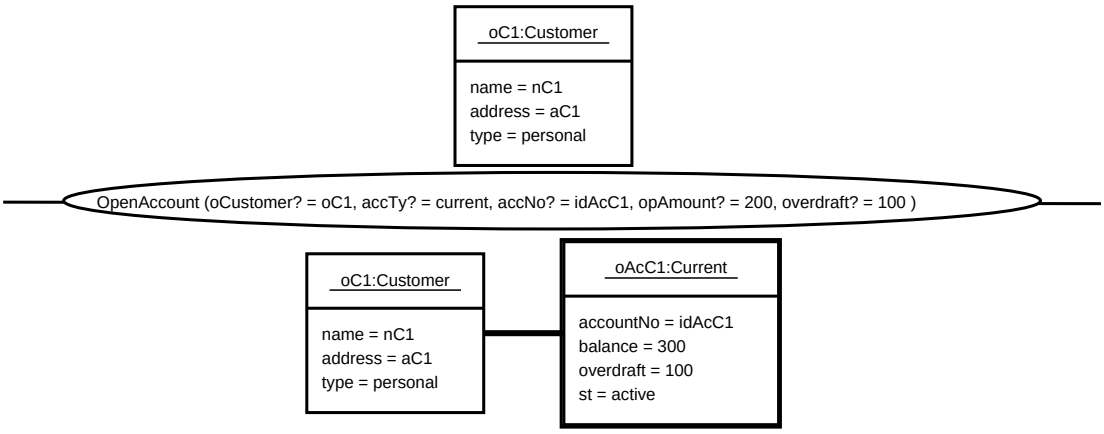


Figure 6.12: Snapshot-pair: a current account is created for an existing bank customer.

Operation OP2, *Open Account***.** The analysis for this operation starts with a positive snapshot-pair (figure 6.12). In the before state, there is a single Customer, oC1, which is associated with newly created Current account (oAcC1) in the after state. Both conjectures are provable — the model accepts the transition described by the snapshot.

The next snapshot-pair (figure 6.13) breaches system constraint C6. It describes a savings account being created for a customer that does not hold a current account with the Bank. Analysis shows that the snapshot is invalid (negative precondition conjecture is provable): the before-snapshot does not satisfy the operation's precondition (see p. 161). The precondition

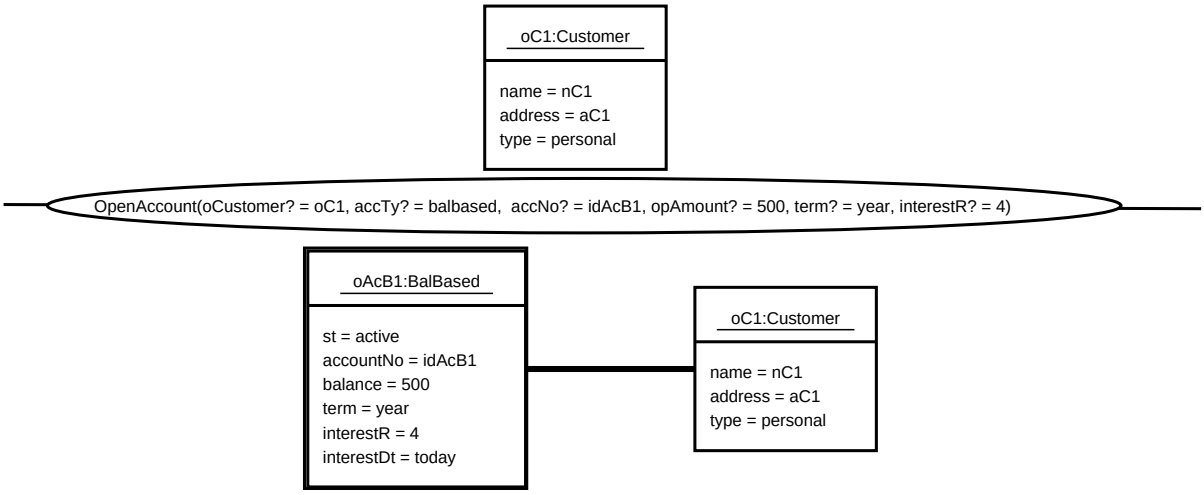


Figure 6.13: A savings account is created for an existing bank customer that does not hold a current account with bank.

allows savings account to be opened only for customers holding a current account — the transition described by the snapshot is not accepted by the model, as expected.

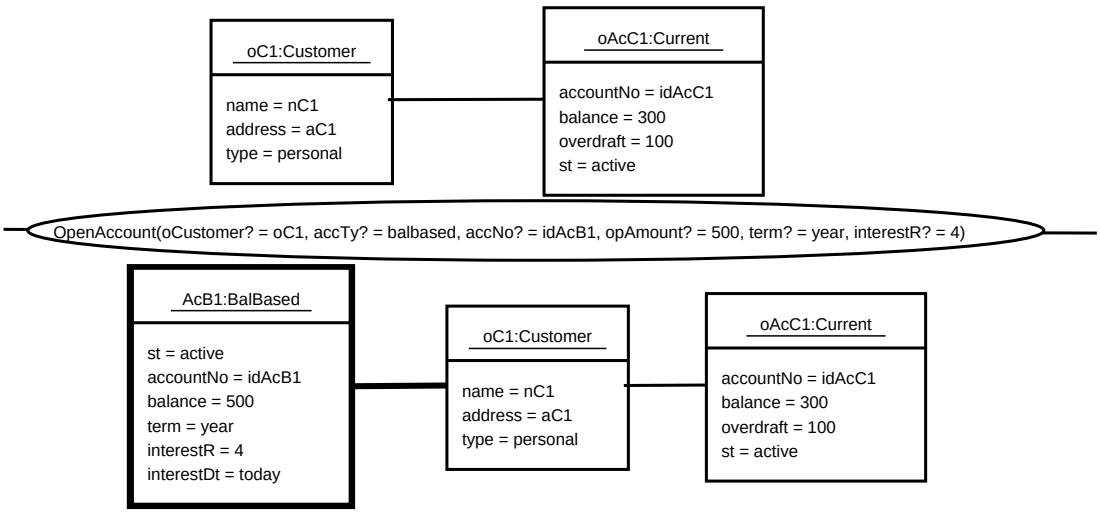


Figure 6.14: A savings account is created for an existing bank customer that already holds a current account with bank.

The next snapshot (figure 6.14) describes how to create a savings account properly (positive). In the before state, there is a customer with a current account, and in the after state a BalBased savings account is created for that customer. The positive precondition conjecture is provable, but, surprisingly, the transition conjecture is not — there is something wrong with the model.

Z/Eves says that the predicate corresponding to constraint C12,

$$\mathsf{mult}(rHolds', sCustomer', sCurrent', ozo, \varnothing, \varnothing)$$

is false for this snapshot. In the after state (see snapshot), *rHolds* is $\{(oC1, oAcC1), (oC1, oAcB1)\}$, *sCustomer* is $\{oC1\}$ and *sCurrent* is $\{oAcC1\}$. The mult constraint requires,

$$rholds^{\sim} \in sCurrent \rightarrowtail sCustomer$$

which does not holds for this instance: $oAcB1 \in \operatorname{dom} rHolds$, but $oAcB1 \notin sCurrent$. This because the formulation of the most recent constraint, C12, is not quite right. It needs to be re-formulated to:

ConstOneCurrent

$\mathbb{S}Customer$; $\mathbb{S}Current$; $\mathbb{A}Holds$

$\mathsf{mult}(rHolds \rhd sCurrent, sCustomer, sCurrent, ozo, \varnothing, \varnothing)$

The association Holds should be *one* to *zero*, *one* only for those tuples that include current accounts. With this modification: the model is consistent, the conjectures for all previous snapshots used in the analysis (above) are still provable, and now both positive conjectures for this snapshot are also provable — the snapshot-pair is valid.

The previous snapshot is valid because the customer is of type personal (constraint C3). The next snapshot (figure 6.15) tries to do the same with a company Customer. As expected, the snapshot is invalid (negative precondition conjecture is provable).

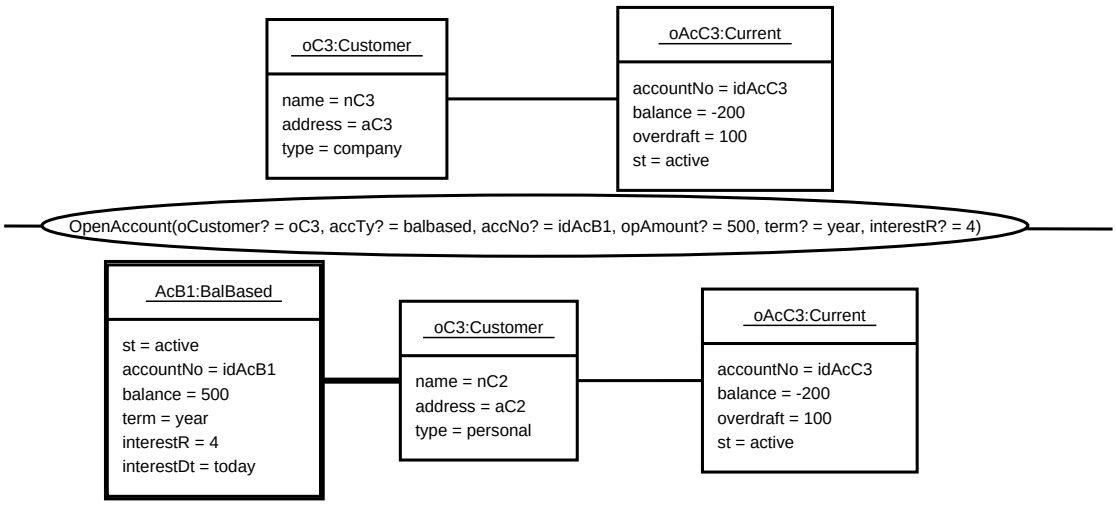


Figure 6.15: Snapshot-pair: a savings account being created for a *company* customer.

Above, all negative snapshots fail on the precondition conjecture. This means that there is either a problem with the before-state or the inputs. Sometimes, however, the positive precondition conjecture is provable, but the transition conjecture is not. This occurs whenever the after snapshot describes a state that is not reachable by running the operation from the before-state with the given inputs. The next snapshot (figure 6.16) illustrates this. In the before state there are two Customers. The operation requests the creation of a current account for oC1, but, in the after state, the account is created for another customer (oC2). So, the

positive precondition conjecture is true, there is nothing wrong with the before state and inputs, but the positive transition conjecture is false (the negative conjecture is provable) — the model does not accept the snapshot-pair.

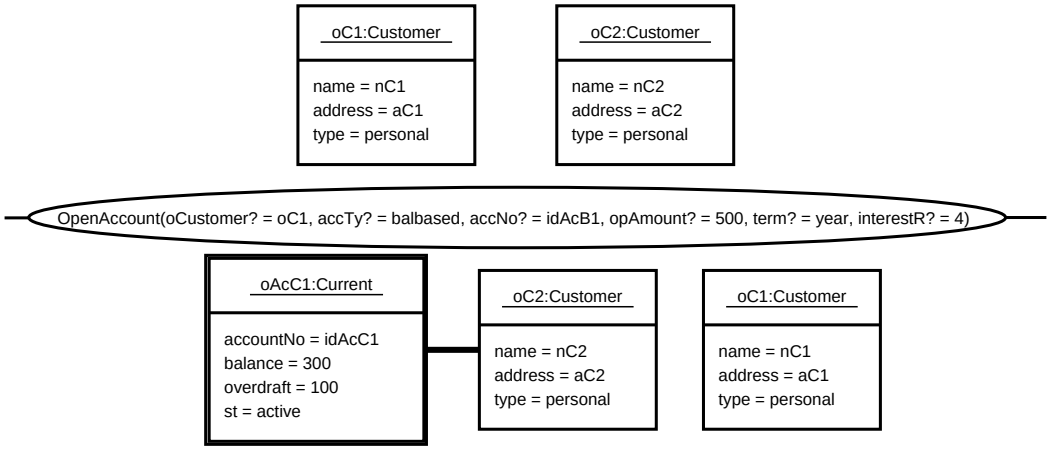


Figure 6.16: A current account being created for the wrong customer.

This ends the analysis with snapshot pairs. The analysis tested the operations *New Customer* (OP1) and *OpenAccount* (OP2), constraints C3 and C6. Analysis found an error in the model with with system constraint C12.

6.2.4 Analysis of operations with snapshot sequences

So far, operations have been investigated in isolation with snapshot-pairs. Sometimes, however, we need to study the effect of certain combinations of operation executions. This may be used to investigate *reachability*. The snapshot analysis does this with *snapshot-sequences*, which are sequences of snapshot pairs, where the after state of one pair is the before state of the next. They are valid if each pair is valid and invalid otherwise.

Figure 6.17 is the first snapshot sequence of the analysis. It tests system constraint C4, which says that withdrawals are not allowed when the account is suspended. So, the snapshot describes a run of the system operation *Suspend* followed by a *Withdraw*. Formal analysis should confirm that the first pair should be valid and the second invalid.

The representation of this snapshot sequence is *fully generated* from template T67 to give:

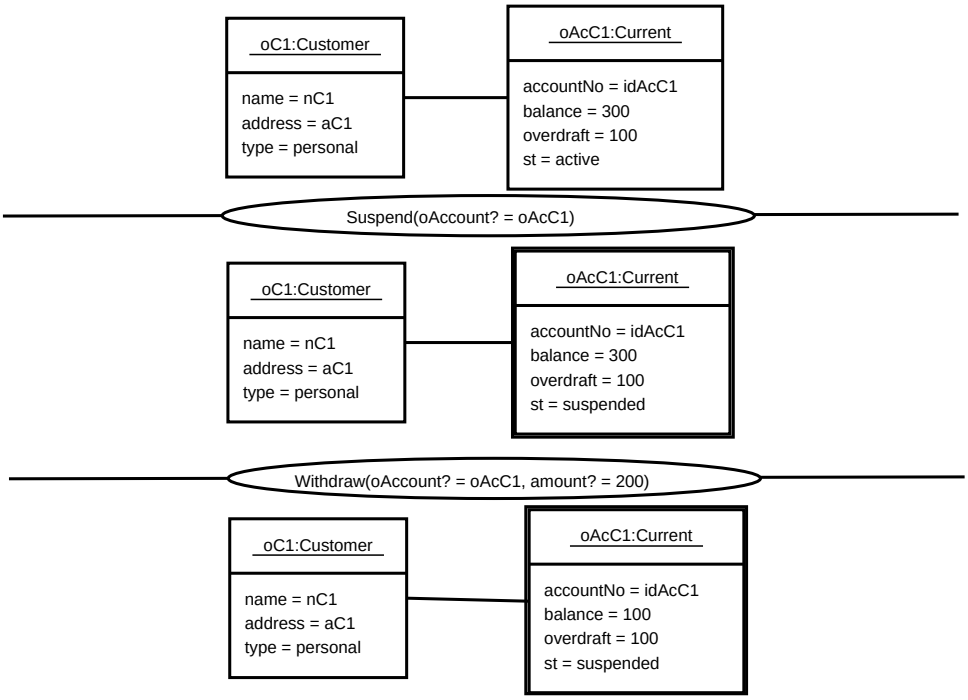


Figure 6.17: An attempt to withdraw money from an account that is *suspended*.

$$
\begin{array}{|l}
nC1 : NAME \\
aC1 : ADDRESS \\
idAcC1 : ACCID \\
oC1 : \mathbb{O}_x\, CustomerCl \\
oAcC1 : \mathbb{O}_x\, CurrentCl \\
oStCustomer : \mathbb{O}_x\, CustomerCl \pfun \text{seq}\ Customer \\
oStCurrent : \mathbb{O}_x\, CurrentCl \pfun \text{seq}\ Current \\
\hline
\mathbb{O}_x\, CustomerCl = \{oC1\} \land \#\mathbb{O}_x\, CustomerCl = \#\langle oC1 \rangle \\
\mathbb{O}_x\, CurrentCl = \{oAcC1\} \land \#\mathbb{O}_x\, CurrentCl = \#\langle oAcC1 \rangle \\
oStCustomer = \{oC1 \mapsto \langle \\
\quad \langle\!| \ name == nC1, address == aC1, type == personal \ |\!\rangle, \\
\quad \langle\!| \ name == nC1, address == aC1, type == personal \ |\!\rangle, \\
\quad \langle\!| \ name == nC1, address == aC1, type == personal \ |\!\rangle\rangle\} \\
oStCurrent = \{oAcC1 \mapsto \langle \\
\quad \langle\!| \ accountNo == idAcC1, balance == 300, st = active, overdraft == 100 \ |\!\rangle \\
\quad \langle\!| \ accountNo == idAcC1, balance == 300, st = suspended, overdraft == 100 \ |\!\rangle, \\
\quad \langle\!| \ accountNo == idAcC1, balance == 100, st = suspended, overdraft == 100 \ |\!\rangle\rangle\}
\end{array}
$$

*OpSqSnap*1

System

$sCustomer = \{oC1\} \land stCustomer = \{oC1 \mapsto (oStCustomer\ oC1)\ 1\}$

$sCurrent = \{oAcC1\} \land stCurrent = \{oAcC1 \mapsto (oStCurrent\ oAcC1)\ 1\}$

$sAccount = \{\} \cup sCurrent$
$\quad \land stAccount = \{\} \cup stCurrent \fatsemi (\lambda\ Current \bullet Account)$

$sSavings = \varnothing \land stSavings = \varnothing \land sWBased = \varnothing \land stWBased = \varnothing$

$sBalBased = \varnothing \land stBalBased = \varnothing \land rHolds = \{(oC1, oAcC1)\}$

$IOpSnap1 == [\ oAccount? : \mathbb{O} AccountCl \mid oAccount? = oAcC1\ ]$

*OpSqSnap*2

System

$sCustomer = \{oC1\} \land stCustomer = \{oC1 \mapsto (oStCustomer\ oC1)2\}$

$sCurrent = \{oAcC1\} \land stCurrent = \{oAcC1 \mapsto (oStCurrent\ oAcC1)\ 2\}$

$sAccount = \{\} \cup sCurrent$
$\quad \land stAccount = \{\} \cup stCurrent \fatsemi (\lambda\ Current \bullet Account)$

$sSavings = \varnothing \land stSavings = \varnothing \land sWBased = \varnothing \land stWBased = \varnothing$

$sBalBased = \varnothing \land stBalBased = \varnothing \land rHolds = \{(oC1, oAcC1)\}$

$\vdash?\ \exists \mathrm{pre}\ SysSuspend \bullet OpSqSnap1 \land IOpSqSnap1$

$\vdash?\ \exists\ SysSuspend \bullet OpSqSnap1 \land IOpSnap1 \land OpSnap2\ '$

$IOpSnap2 == [\ oAccount? : \mathbb{O} AccountCl;\ amount : \mathbb{N} \mid$
$\quad oAccount? = oAcC1 \land amount? = 200\ ]$

*OpSqSnap*3

System

$sCustomer = \{oC1\} \land stCustomer = \{oC1 \mapsto (oStCustomer\ oC1)\ 3\}$

$sCurrent = \{oAcC1\} \land stCurrent = \{oAcC1 \mapsto (oStCurrent\ oAcC1)\ 3\}$

$sAccount = \{\} \cup sCurrent$
$\quad \land stAccount = \{\} \cup stCurrent \fatsemi (\lambda\ Current \bullet Account)$

$sSavings = \varnothing \land stSavings = \varnothing \land sWBased = \varnothing \land stWBased = \varnothing$

$sBalBased = \varnothing \land stBalBased = \varnothing \land rHolds = \{(oC1, oAcC1)\}$

$\vdash?\ \neg\ (\exists \mathrm{pre}\ SysWithdraw \bullet OpSqSnap2 \land IOpSqSnap2)$

$\vdash?\ \neg\ (\exists\ SysWithdraw \bullet OpSqSnap2 \land IOpSnap2 \land OpSnap3\ ')$

Above, there are definitions of objects and their states (axiomatic definition), definitions of snapshot instances and inputs to the operations of the sequence.

The first pair of this sequence is valid, both positive conjectures are provable. The second is invalid, the negative precondition conjecture is provable — the model does not accept the sequence, as required.

Now, the analysis faces a reachability dilemma. Figure 6.18 presents a single snapshot that should not be accepted by the model because a customer may hold a savings account only

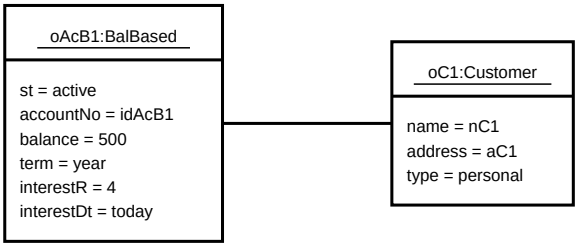


Figure 6.18: A personal customer with only a savings account.

if he also holds a current account. Formal analysis, however, says that the snapshot is valid. The snapshot may be *spurious*, because the analysis of the snapshot-pair of figure 6.13 (see above) tells us that the snapshot is not directly reachable from the operation Open Account (the before-snapshot of that pair is surely reachable). The question now is: could it be reached from another operation or some other combination of operation executions? There are only two operations that change the state of the association Holds: *Open Account* (OP8) and *Close Account* (OP9). We have seen that, directly from the first, it is not reachable, but it could be reached from the latter.

The snapshot-sequence of figure 6.19 investigates this reachability problem. It tries to arrive at the snapshot of figure 6.18 by opening a savings account for the customer, then by withdrawing all available funds from the current account (this to allow the account to be closed, constraint C5), and then by closing the current account. We know that the first snapshot is reachable by the analysis of the snapshot-pair of figure 6.12. Analysis confirms that the snapshot-sequence is valid, which means that the single snapshot is reachable — there is something wrong with the model.

It turns out that constraint C6 is not correctly formulated in the requirements, which is reflected in the ZOO model. So, first we change the natural language constraint to:

C6	A customer may holds a savings account provided it holds also a Current account.

In the ZOO model, this was formulated in terms of a constraint on the operation *Open Account*. This is changed to a system constraint, and the operation constraint is removed. The new system constraint becomes:

ConstHasCurrent

$$\mathbb{S}\mathit{Customer};\ \mathbb{S}\mathit{Current};\ \mathbb{A}\mathit{Holds}$$

$$\forall\, oC : sCustomer \mid oC \in \operatorname{dom} rHolds \bullet oC \in rHolds^{\sim}(\!|\, sCurrent \,|\!)$$

The modified model is consistent, all previous snapshot proofs are provable, and the snapshot of figure 6.18 and the snapshot-sequence are no longer valid.

This ends the analysis with snapshot sequences. Analysis tested operations *Open Account* (OP2), *Withdraw* (OP4), *Close Account* (OP8), and *Suspend* (OP9) and constraints C4, C5 and C6. Analysis found an error with the formulation of constraint C6 in the requirements, and (consequently) in the ZOO model.

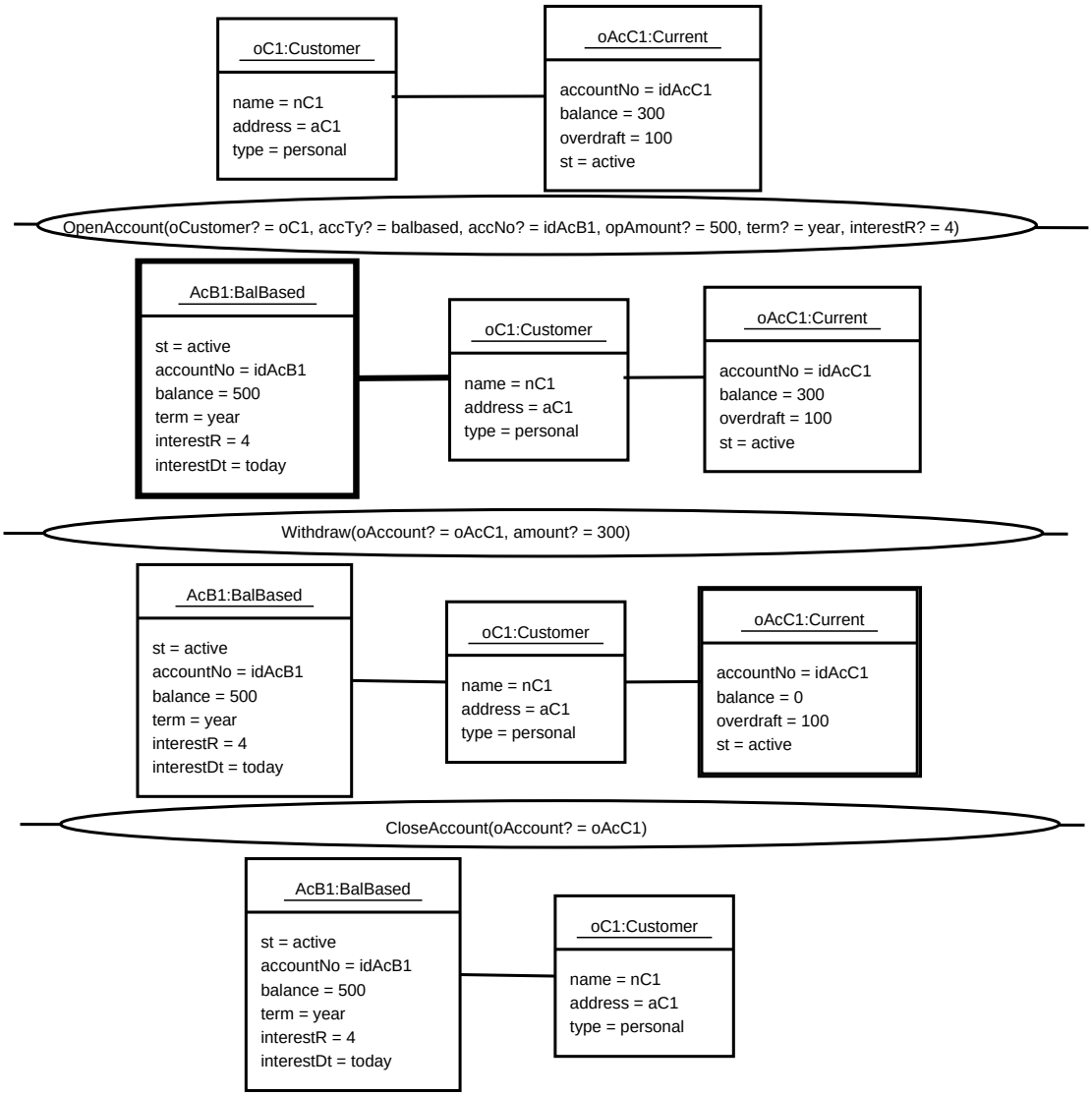


Figure 6.19: A savings account is opened for a customer, followed by the closure of a current account. The customer remains with just one savings account.

6.3 Discussion

Snapshot analysis helps to get the model and the requirements right. In the case study, snapshot analysis highlighted the need for a new system constraint (C12), found an error with the formulation of this constraint in the model, and found a problem with an existing system constraint (C6). Other applications of the technique [ASP04, APS06] yielded similar results. In [APS06], snapshot analysis played a significant role in getting the requirements of the system right. Snapshots give life to a model by animating a specification, helping in understanding the requirements, the intricacies of the system and its model.

Although not demonstrated here, snapshot analysis is capable of dealing with non-deterministic operations. This is illustrated in [APS06]. Essentially, as explained in section 6.1, snap-

shot conjectures check for membership in a set. The conjecture associated with a single snapshot checks whether the state represented by the snapshot belongs to the set of valid states described by the model (see figure 6.1). The conjectures associated with a snapshot-pair check whether the pair of before and after states represented by the snapshot belong to the set of all valid pairs of the operation (see figure 6.2). Finally, the snapshot sequence checks whether each individual snapshot-pair that makes a sequence belongs to the set of all valid state pairs of each transition (an operation call). So, from an analysis point of view there is no difference between a relation that is functional (a deterministic operation) and one that is not (a non-deterministic operation); in the end the pair of before and after states must belong to the set of all pairs.

Templates help the analysis to be focused on the most interesting questions. The ZOO model is generated from the templates of the increasingly mature ZOO style, which eliminates the possibility of those tiny errors that have nothing to do with requirements or interesting properties of the system and that just add noise to the analysis activity. When such errors are found (more were found when the style was fresher) the templates are corrected and the same error does not creep again into the generated models. As this case study and experience shows, most errors found with analysis are interesting: they are either related to the requirements or formulation of requirements in the model. This makes errors easy to trace. Usually, they can be found in the extra constraints that are added by the user.

The technique blends formality together with a visual notation, and it is in this blend that lies its power. Formality gives the means to capture errors mechanically, no matter how subtle they are. Formal analysis always brings some surprises: what, at first, seemed right either from the informal or even formal point of view turns out not to be so after analysis. In the bank case study, this can be observed in the way the error with the formulation of constraint C12 is captured (see above). The simple and visual notation, on the other hand, gives the right dose of abstraction from the details of the underlying formal model, making it easier to assimilate the results of the analysis. This enables modellers to think clearly and not to be confused with details, prompting them to make questions about the system and try things. In the bank case study, the analysis that lead to constraint C12 and the error with constraint C6 are examples of this.

The use of a graphical notation makes formal analysis reachable to a wider audience. It allows developers that are not experts in Z to participate in the formal analysis of the system, by drawing and reading snapshots, and understanding why they are satisfied or not. It also allows the customer to be engaged, constituting a good medium to receive feedback about the model, because snapshots are an intuitive notation. At this level of abstraction, snapshots describe objects from the application domain, which the customer knows about.

An effective analysis should make a good combination of positive and negative snapshots. Positive snapshots find errors when things do not work as expected, and give a feeling of confidence about the model. But it is negative snapshots that allow users to jump into the most interesting questions. At the beginning of the analysis, when the model is still fresh or when the requirements are not very clear, negative analysis can be a rewarding activity, exposing errors that would take a lot more time to be discovered with positive snapshots. In the case study, this can be observed in the analysis that lead to constraint C12, where a negative snapshot uncovered an error. In other applications of the technique [ASP04, APS06], negative analysis proved equally rewarding in the early stages.

As discussed above, there is the danger of using spurious snapshots in the analysis. That is, of using snapshots that although valid should not be accepted by the model because they

are not reachable. In such cases, the analysis may be misleading. Snapshot sequences help to investigate this. In the case study, the error with constraint C6 was found by making a reachability investigation with snapshot sequences.

It is interesting to analyse how spurious instances can creep into a model. As the example shows, whenever there are extra constraints on global operations, there is the possibility of spurious instances, because those constraints are not taken into account in the constraints of the state space. That is why the snapshot of figure 6.18 was initially valid. But when the operation constraint was transformed into a static constraint (a system constraint), that snapshot was ruled as invalid by the analysis. The advantage of operation constraints, however, is that the model becomes lighter: the same constraint does not have to be taken into account in the system initialisation and every system operation.

Snapshot proofs are easier than other kinds of specifications proofs (e.g. precondition proofs at the global system level are much harder). This is because they deal with instances of models (it is much easier to prove that one instance satisfies some property, than proving it for all instances). In most cases and with human intervention, Z/Eves can deduce whether a snapshot conjecture is true or false. But it cannot deduce false in every situation, that is why we prove negative conjectures. Moreover, although snapshot proofs are simple, some intervention from the user is required. In Z/Eves, to prove a big theorem we need to prove many smaller ones, and so some preliminary proving is involved in the proof of snapshots. So the Z expert is required for snapshot analysis. However, there is room for further automation because these preliminary proofs are always the same and they are trivial. There could be a tool that generates the representation of snapshots from templates, and also generates and proves preliminary theorems for Z/Eves so that snapshot proofs are automatically discharged.

Snapshot analysis is a form of model testing, which may be regarded with suspicion. After all in a space of a very large number of states, snapshot analysis considers but only a small fraction of these. The technique is also bound by Dijkstra's famous words [Dij72]: "Program testing can be used to show the presence of bugs, but never to show their absence." Yet, snapshot analysis has proved to be useful and effective at finding errors. It gives means to observe what goes on inside the system, sometimes the error that is captured is a direct result of the observation, and not related at all with the property that the snapshot intended to test. In the bank case study, the error with the formulation of constraint C12 was found in this way. Moreover, snapshots may prompt more exhaustive verifications, in the form of proofs of general theorems. Snapshot analysis still requires the user to design "good" test cases, but provides a visual tool that can be a valuable aid in doing so, and a formal tool to check those test cases.

The snapshots used in the analysis can be the basis of system acceptance tests. Snapshot analysis is used to check that the model reflects the requirements, making it an activity that is of interest to the customer of the system. The snapshots that are valid or invalid in the model should also be so in the final system. This brings the opportunity for using the snapshots used in the analysis to test the actual system.

6.4 Related Work

Catalysis [DW98] introduced single snapshots to illustrates states and snapshot-pairs to illustrate actions so that users can better understand the effects of modelling decisions. This thesis gives formal support to snapshots, so that validity-checks can be performed mechani-

cally, and introduces the snapshot sequence to analyse combinations of operation executions and to investigate reachability.

Snapshot analysis is related to model animation (or simulation) [JR00, HJ89] in formal modelling languages. Animator tools, such as the B-Toolkit, allow the user to execute operations of the specification with user supplied parameters, which is not possible when operations are non-deterministic. Other animator tools, such as the JaZA Z animator [Utt], allow a model to be tested by checking the validity of model instances. Our model analysis does animation through testing, like JaZA does, but with the Z/Eves prover. This does not impose restriction in terms of non-determinism, and Z/Eves, unlike JaZA, supports all constructs of the Z language (it would not be possible to perform the analysis done here with the JaZA animator).

Like the *UML + Z* framework, Alloy [Jac06] is a language designed for lightweight formal modelling. Alloy's analysis approach, *instance finding*, is related to the approach presented here. Alloy's tool, the Alloy analyser, looks for counter examples (snapshots) of some formula, and generates sample states (snapshots) of a model automatically [Jac06]. Alloy does this without using theorem proving, to avoid the need for expertise. These instances are found with constraint solvers. The idea is to run some formula against a very large set of test cases; if an instance (sample or counter-example) is found then it is reported, otherwise there is a high probability that such an instance does not exist, but this is not certain: it may be that the instance is not in the set of test cases. The Snapshot analysis technique proposed here relies on users to draw snapshots, and on the expert to prove them with the theorem prover. In our approach, if proved then it is certain that the snapshot is either valid or invalid, but this relies on the expert. However, as discussed above, there is room for automation, maybe not for all cases, but very close to that. There is also the possibility of generating instances automatically from templates.

The work of Richters and Gogolla [RG98, Ric01, GBR03] on OCL supports snapshot-based validation with the USE tool based on model animation and constraint resolution. It supports single snapshots and snapshot pairs. The approach is based on an interactive command language to animate the model. The approach to constraint resolution is similar to the one used by Alloy, so it is not 100% certain, but the checks are done automatically. In addition, the approach includes a generator of test cases [GBR03]. Our approach relies on the certainty of proof to check that a model is consistent and that a snapshot is valid (USE does not support model consistency, and the question of what constitutes a consistent UML+OCL model is still open [Ric01]).

6.5 Conclusions

This chapter presented snapshot-analysis, a technique to analyse *UML + Z* models by using snapshots (object diagrams) and formal proof. This allows the analysis of state space with single snapshots, and operations with snapshot-pairs and sequences. An approach to study reachability with snapshot sequences was also proposed. The technique was illustrated to analyse the model of the Bank case study developed in the previous section. The analysis found missing requirements, problems with the formulation of existing requirements, and problems with the formulation of requirements in the model.

The main contribution of this chapter is the snapshot analysis technique, an approach to formally analyse models with diagrams. This work was inspired by Catalysis snapshots and it

emerged from the observation that the validity of those snapshots could be formally checked through proof by representing them in ZOO. Later, snapshot sequences (not proposed by Catalysis) were also explored. There has been work in validating snapshots in the context of OCL, but the author is not aware of a similar work that explores snapshots and formal proof.

This is the last chapter in the development of the $UML + Z$ framework in this thesis. The next chapter evaluates the work reported in this thesis, draws the final conclusions and suggests future work.

7

Evaluation, Conclusions and Future Work

The previous chapters have developed FTL, which allows the construction of template catalogues for GeFoRME frameworks, ZOO, the semantic domain of the $UML + Z$ framework, and snapshot analysis, the analysis approach of $UML + Z$. Each chapter has already reflected on the reported work and evaluated it against related work.

The following evaluates the hypothesis in the light of the thesis' results and both GeFoRME and $UML + Z$ against related work. Finally, some future avenues of research are suggested.

7.1 The Hypothesis

The research reported in the previous chapters provides evidence in favour of the thesis' hypothesis:

> **It is possible to integrate formal methods into mainstream software modelling practice. This can be done in a flexible and practical way, so that the integration has an engineering value and is actually worth doing.**

As mentioned in the introduction (chapter 1), the hypothesis is subdivided into three sub-hypothesis: (a) the possible, (b) the flexible and practical, and (c) the worth doing. These are evaluated against the results of this thesis.

7.1.1 "Possible"

Other approaches are also used, but, essentially, it can be said that mainstream modelling is based on the OO paradigm, in most cases, using the UML. This is evidenced by the practice in industry, and by the fact that almost every undergraduate computer science course in the universities around the world has a course on OO analysis and design with the UML. By contrast, not all universities have a course on formal modelling. The ones that have such a course use various formal methods, sometimes with very diverse underlying paradigms. So,

the modelling idiom that seems to be *lingua franca* among software engineers is OO modelling using UML.

ZOO, developed in chapters 4 and 5 of this thesis, is the key to the integration of the Z formal method with OO modelling. ZOO makes the integration possible by providing a formal foundation for the representation of OO concepts, such as objects, classes, associations, systems and class inheritance, in an elegant way. It allows the checking of behavioural conformance in inheritance hierarchies based on the theory of refinement for Z. Preliminary research indicates that it is also possible to represent part-of relationships between classes (not shown in this thesis, see [ASP03]). As discussed in chapters 4 and 5, ZOO goes further than other formal representations of OO concepts. Unlike formalisations into B, ZOO's structuring is both flexible and modular, more flexible than Object-Z, and deals with the tricky issue of behavioural inheritance, which is not dealt with in most OO formalisations.

It is worth emphasising, once again, ZOO's flexibility, which is inherited from Z and its schema calculus. ZOO makes it possible to have alternative representations of certain concepts (for example associations may also be represented as class attributes), and to have concepts that are related but have slightly different meanings (using what in UML is known as *stereotypes*).

The thesis then demonstrated how ZOO can be used together with UML diagrams for the purpose of OO modelling. This was illustrated in the context of $UML + Z$ framework with two versions of the Bank case study, the second including inheritance. The $UML + Z$ framework has also been used to model a simple library system [ASP04], and a simple invoice system [APS06].

So, ZOO demonstrates that it is possible to integrate Z within a development based on OO modelling for general purpose sequential systems. Next, we see whether this is also flexible and practical, and worth doing.

7.1.2 "Flexible and Practical"

ZOO is key in making the integration of Z (a formal method) with OO modelling (a mainstream method) possible. ZOO already contributes to make the overall approach practical and flexible. However, the key to flexibility and practicality comes from GeFoRME together with FTL and its meta-proof approach.

The FTL templates of a GeFoRME catalogue represent recurring structures of formal models. FTL allows the factoring of experience for the purpose of reuse in the form of templates. Users of some GeFoRME framework, experts or not, do not need to start the development from a blank sheet of paper, they can experiment with the structures that the catalogue offers. If those structures are fit for the intended purpose then they can be reused. Otherwise the catalogue can be extended, possibly by specialising existing templates. A catalogue of templates is a living repository of knowledge for modelling within some domain; it grows and evolves with experience. Users can add templates to the catalogue, update existing templates, and delete templates that are redundant or that are no longer used.

This almost organic feature of GeFoRME's template catalogues was confirmed by the author's experience with $UML + Z$. The catalogue of FTL templates grew and evolved as $UML + Z$ was being applied to more problems. Once the solution to some problem was found it was captured with templates, and, most likely, it was changed many times until reaching maturity. In developing both ZOO and $UML + Z$'s templates catalogue, it was necessary to go back and change what had previously been done. These changes would not be substantial,

	Total	Meta-proof			ThP			Eff. %
		TbC	SiEP	SiP	TT	EP	P	
Inh Bank without user constraints	76	69	1	0	7	0	0	92 %
Inh Bank with user constraints	202	139	16	14	34	15	14	69 %

Table 7.1: Effectiveness of meta-proof in proving the consistency of a ZOO model for the Bank case study with inheritance (chapter 5). Two cases are considered: a ZOO model generated from the information contained in the diagrams only, and the same model, but with user-generated constraints and operations. The table gives the total number of proofs (Total) and data related to meta-proof, theorem proving (ThP) and a measure of the effectiveness of meta-proof. In meta-proof, there is the total number of proofs that are *true by construction* (TbC), and simplifications to *easily provable* (SiEP) and *provable* (SiP). In theorem proving (ThP), there is the total number of proofs that are *trivially true* in Z/Eves (TT), the ones that are easily provable (EP) and provable (P). The last column gives a measure of the effectiveness of meta-proof, indicating the percentage of those proofs that are true by construction.

but minor adaptations to something that was being explored, or the result of new insights. Once such a solution was captured and mature, it was of great practical value because it could be reused many times, not automatically because FTL tool support is not yet available, but routinely, allowing the mind to focus on new structures not yet captured with templates and the problem being modelled. The thinking for reuse also leads to better solutions. One is not just happy with a solution to the problem at hand, that solution must also be modular and reusable, preferably, by the widest possible variety of contexts. This helps to foster a pattern-based thinking, by identifying patterns of problems and their solutions.

So, reuse of templates makes GeFoRME frameworks practical, and templates themselves and extensibility of the catalogue makes it flexible. This is further empirical evidence for the value of patterns in software engineering. It is also evidence of something less known: that patterns can enhance the practicality of formal development through reuse of modelling structures. In addition, this thesis has focused on something that is equally important for practical formal development: reuse at the level of formal proof. But this raises the question: how effective is meta-proof?

Table 7.1 provides data regarding proof of consistency in the model of the Bank case study with inheritance (chapter 5). Two cases are considered: proof in the ZOO model that is generated from the information contained in the diagrams only, and the final ZOO model with constraints and operations provided by the user. The column (TT) indicates those proofs that are *trivially true*, which all involve well-formedness conjectures (see chapters 4 and 5). The column effectiveness (Eff. %) tries to assess the effectiveness of meta-proof: it gives the percentage of proofs that are *true by construction* (TbC). As the results show, when user constraints are added to the model the effectiveness of meta-proof decreases. The first model does not have user constraints, but that does not mean that it is unconstrained; as it can be seen in figure 5.2 (page 110), there is a constraint upon the association and several constraints related with inheritance. As these constraints are known in advance, they are dealt with by proof at the template level; meta-theorems discharge them so that user proofs

are simpler and do not have to deal with those constraints again.

The effectiveness figures of meta-proof look good, but there are some shortcomings. First, most of the results that are proved *true by construction* are also *trivially true* in the Z/Eves prover. So, although the results of true by construction are high, they are not giving us much more than what the machine already gives automatically. Moreover, *true by construction* is slightly inflated due to the high number of proofs related with behavioural inheritance, which are all true by construction in that model. A model where behavioural inheritance has less impact, would probably give a smaller number of proofs that are *true by construction*.

Despite these shortcomings, the results are promising. The high number of *simplification to provable* may seem, at first, disappointing, however, many of those simplifications make proof with the theorem prover easier. Theorem provers are efficient if the number of clauses in a sequent remains small, but their performance degrades as the number of clauses increases. At the system level, where theorems have a very high number of clauses, every simplification that discharges global constraints is useful. In the Bank example, in almost all cases, meta-proof would eliminate constraints involving associations and inheritance (those that are known in advance) from the user proof.

In the author's experience, the power of meta-proof lies in this a-priori knowledge. The author believes that if frameworks are made even more domain-specific, then there is a lot more to be gained from meta-proof. The $UML + Z$ framework targets general purpose OO sequential systems. This is still too general. The catalogue of templates basically restricts Z to express models with an OO structuring, and despite all that generality the table above showed the gains that can be made. If ZOO and $UML + Z$ are specialised even further, the meta-proof should yield even better results.

The biggest obstacle to practicality in $UML+Z$ seems to be the dependence on the expert. Many developers may participate in the development, but Z/ZOO experts are always required: to build templates that capture structures, to express constraints, and to make proofs in the theorem prover. But, the benefits of formal modelling are available for a wider audience, rather than just a few. The gap between mainstream modelling and formal modelling is diminished with a framework.

7.1.3 "Worth doing"

At a time where software is increasingly becoming part of our environment and in control of infrastructure, software reliability is paramount. And reliability is one of the reasons why an approach such as $UML + Z$ is *worth doing* and using.

However, given today's priorities of the IT industry, which are, essentially, driven by *time-to-market*, using reliability as argument for "doing it" may sound almost moralistic: engineering artifacts not only need to be reliable, their production must also be cost-effective. Although they may provide the best possible software, the practical limitations of formal methods usually mean that the costs of achieving reliability are just too high. But there may be economic benefits (which are perhaps more compelling in today's priorities of IT industry) in using approaches based on formal methods, and the more practical the approach the more likely it is that the approach is beneficial from an economic point of view.

First, only formal analysis can give a timely detection of those errors that cost much more if not found until later. However, the cost of software errors as well as the benefits of timely detection are difficult to assess. After all, how much do such undetected errors cost? An action in court? A lump-sum in compensation? Or just some dissatisfied users with their

faulty software.

Second, given today's increasing awareness of software errors, reliability may actually give a competitive advantage. Again, this is difficult to assess. What do we gain with more reliable software? A happy customer that will make business with us again? Or an unhappy one, because software, although reliable, is late?

It is difficult to evaluate all this. What will be decisive in the adoption of approaches to development based on formal methods is cost-effectiveness. This thesis tried to make practical those methods that detect early and costly software errors and yield reliable software. This may contribute to the widespread adoption of formal-based approaches (such as as $UML + Z$ developed here), simply because their use may be found cost-effective.

Overall, if it is ZOO that makes an integration of formal methods with mainstream practice possible, and the $UML + Z$ catalogue of templates that makes the approach flexible and practical, it is the possibility of snapshot analysis that makes it worth doing. This is because snapshot analysis is based on formal analysis, which helps detection of costly errors, and diagrams, which can be used by anyone. Of course, the expert needs to be around, but this is the small price to pay for reliability. This thesis demonstrated that $UML + Z$ is more practical than raw Z, which makes Z and formal methods more desirable for industrial software development.

7.2 Related Work: GeFoRME and $UML + Z$

Catalysis [DW98], a modelling method based on the UML, is related to the work presented here. Catalysis develops the idea of model frameworks and templates to make models reusable assets, the ideas of model refinement with UML diagrams, and the idea of defining semantics of UML constructs adapted to the context in which they are used. In Catalysis, this is all developed semi-formally. By contrast, GeFoRME and $UML + Z$ do it formally.

Also related, is the work on meta-modelling frameworks, such as BOOM [Ö00] and MMF (meta-modelling framework) [CEK02, CEK01]. These approaches are designed to define variants of graphical modelling notations (such as the UML), using a meta-modelling approach, but with a formally defined language. These works aim at the sole use of graphical notations with precise semantics. GeFoRME and $UML+Z$, on the other hand, aim at a combined use of graphical notations and formal modelling languages. Unlike BOOM and MMF, which resort to newly defined formalisms, GeFoRME uses a well-established formal modelling language, with a proof system, a refinement calculus, and that embodies years of research and best practice in formal development.

7.3 Future Work

This thesis has proposed the GeFoRME approach and developed the $UML + Z$ framework. The proposed work constitutes a valuable contribution, but it is far from over both on the general side (GeFoRME) and the particular ($UML + Z$ and other frameworks).

Further development of GeFoRME is linked with the development of FTL developed in chapter 3. FTL exhibits promising results, but would benefit from further features. In particular, naming of templates and a conditional instantiation construct (an if-then-else for templates). The naming should allow better combinations of templates than is possible with current FTL. FTL would also benefit from tool support to automatically generate models

from templates and to simplify proofs by instantiating meta-theorems. This would allow a better validation of templates and their meta-proofs, and it would automate what is currently being done manually, making template-based model generation truly practical.

Chapter 3 also developed a draft logic for template-Z. This logic should be completed, by proving the inference rules that are now laid as axioms of the logic. This should be done together with the encoding of the template-Z logic in ProofPower [Art]. This would bring support for meta-proof with Z.

Chapters 4 and 5 developed ZOO, which constitutes a rich semantic for the representation of UML-based models. However, the formal mapping from diagrams into FTL templates of ZOO has not been developed here. This should be considered together with tool support for $UML + Z$, which requires the existence of this mapping.

Chapter 6 developed snapshot analysis, an approach to formal proof with diagrams. An interesting development here it would be to try to generate more general theorems from user-drawn snapshots, so that proof does not only demonstrate the validity of the drawn snapshot, but also the validity of a whole class of related snapshots. This would provide a much more general analysis. It would also be interesting to automatically generate snapshots from class diagrams (or their ZOO representations) and then to automatically prove them, which would bring automated analysis to the level of snapshots.

$UML+Z$'s refinement component, missing from this thesis, should be considered by future work. The aim would be to develop an approach to refinement of $UML + Z$ models based on the refinement theory for Z. Then the aim would be to capture refactorings with templates and try to reduce the overhead of proof (that is associated with proving the correctness of refinements) by using meta-proof.

Future work should consider specialising $UML + Z$ to make more domain-specific frameworks. $UML + Z$ targets general-purpose OO sequential systems. It is a general framework, that could be specialised for more specific application domains. The author believes that the more-domain specific the framework, the more there is to be gained from meta-proof. This would also constitute a useful opportunity to explore inheritance to capture commonality at the level of frameworks, as is done in OO frameworks. OO inheritance together with templates could be an even more effective reuse mechanism.

The GeFoRME approach and FTL would benefit greatly from an application to another framework, preferably involving another formal modelling language. This would enable a validation of FTL to see what features of its design are too tied to Z, and it would probably highlight the need for further features in FTL.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Abr03] Jean-Raymond Abrial. Event based sequential program development: Application to costructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Int. Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 51–74. Springer, 2003.

[Abr05] Jean-Raymond Abrial. Using design patterns in formal developments. In *RefineNet, Workshop of ICFEM 2005*, 2005.

[AC96] Martín Abadi and Luca Cardelli. *A theory of objects.* Monographs in Computer Science. Springer, 1996.

[ACES01] José Alvarez, Anthony Clark, Andy Evans, and Paul Sammut. An action semantics for MML. In Martin Gogolla and Cris Kobryn, editors, *UML 2001: The Unified Modeling Language. Toronto, Canada.*, volume 2185 of *LNCS*, pages 2–18. Springer, 2001.

[ACM02] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of the IEEE 1394 tree identify protocol. *Formal Aspects of Computing*, 14(3), 2002.

[ACM05] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in Event_B. In Treharne et al. [TKHS05], pages 222–241.

[ACWG94] M. Ainsworth, A. Cruickshank, P. Wallis, and L. Groves. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, 1994.

[AIS77] Cristopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction.* Oxford University Press, 1977.

[Ale79] Christopher Alexander. *The timeless way of building.* Oxford University Press, 1979.

[Ale99] Christopher Alexander. The origins of pattern theory: the future of the theory, and the generation of a living world. *IEEE Softw.*, 16(5):71–82, 1999.

[AP03] Nuno Amálio and Fiona Polack. Comparison of formalisation approaches of UML class constructs in Z and Object-Z. In Bert et al. [BBKW03], pages 339–358.

[APS05] Nuno Amálio, Fiona Polack, and Susan Stepney. An object-oriented structuring for Z based on views. In Treharne et al. [TKHS05], pages 262–278.

[APS06] Nuno Amálio, Fiona Polack, and Susan Stepney. *Software Specification Methods: an overview using a case study*, chapter UML+Z: UML augmented with Z. Hermes Science, 2006.

[Ara96] João Araújo. *Metamorphosis: An Integrated Object-Oriented Requirements Analysis and Specification Method*. PhD thesis, Dept. of Computing, University of Lancaster, 1996.

[Art] R.D. Arthan. ProofPower. `http://www.lemma-one.com/ProofPower/index/index.html`. Last accessed on 24/01/2007.

[AS97] Klaus Achatz and Wolfram Schulte. A formal OO method inspired by Fusion and Object-Z. In Bowen and Hinchey [BH97], pages 91–111.

[ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. A specification language. In R. McNaughten and R.C. McKeag, editors, *On the construction of programs*. Cambridge University Press, 1980.

[ASP03] Nuno Amálio, Susan Stepney, and Fiona Polack. Modular UML semantics: Interpretations in Z based on templates and generics. In Hung Dang Van and Zhiming Liu, editors, *Int. Workshop on Formal Aspects of Component Software*, number 284, pages 81–100. UNU/IIST Technical Report, 2003.

[ASP04] Nuno Amálio, Susan Stepney, and Fiona Polack. Formal proof from UML models. In Jim Davies et al., editors, *ICFEM 2004*, volume 3308 of *LNCS*, pages 418–433. Springer, 2004.

[Bac60] John W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *IFIP Congress, 1959*, pages 125–132, 1960.

[Bal88] S. Balin. *Remarks on Object-Oriented requirements specification*. Computer Technology Associates, 1988.

[Bar98] John Barnes. *Ada 95*. Addison-Wesley, 1998.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–29, 1987.

[BBHR02] Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors. *ZB 2002: Formal Specification and Development in Z and B, Grenoble*, volume 2272 of *LNCS*. Springer, 2002.

[BBKW03] Didier Bert, Jonathan P. Bowen, Steve King, and Marina Walden, editors. *ZB 2003, Turku, Finland*, volume 2651 of *LNCS*. Springer, 2003.

[BDBS99] E.A. Boiten, J. Derrick, H. Bowman, and M.W.A. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75, 1999.

[BDG05] Fabrice Bouquet, Frèdèric Dadeau, and Julien Groslambert. Checking JML specifications with B machines. In Treharne et al. [TKHS05], pages 434–453.

[BDGK00] Jonathan Bowen, Steve Dunne, Andy Galloway, and Steve King, editors. *ZB 2000: Formal Specification and Development in Z and B, York, UK*, volume 1878 of *LNCS*. Springer, 2000.

[BDW99] Christie Bolton, Jim Davies, and Jim Woodcock. On the refinement and simulation of data types and processes. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *IFM'99, 1st Int. Conf. of Integrated Formal methods, York, UK*, pages 273–292. Springer, 1999.

[BDW06] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. Semantic issues of ocl: Past, present, and future. In B. Demuth, D. Chiorean, M. Gogolla, and J. Warmer, editors, *OCL for (Meta-)Models in Multiple Application Domains*, pages 213–228, 2006. Available as Technical Report, University Dresden, number TUD-FI06-04-Sept. 2006.

[BF98] Jean-Michel Bruel and Robert B. France. Transforming UML models to formal specifications. In Bézivin and Muller [BM98].

[BG80] Rod M. Burstall and Joseph A. Goguen. The semantics of Clear, a specification language. In *Proceedings of Advanced Course on Abstract Software Specifications*, volume 86 of *LNCS*. Springer, 1980.

[BGL03] Sandrine Blazy, Frédéric Gervais, and Régine Laleau. Reuse of specification patterns with the B method. In Bert et al. [BBKW03], pages 40–57.

[BH94] J.P. Bowen and A. Hall, editors. *Z User Workshop, Cambridge*, Workshops in Computing. Springer, 1994.

[BH97] J.P. Bowen and M.G. Hinchey, editors. *ZUM'97: The Z Formal Specification Notation, Int. Conf.*, volume 1212 of *LNCS*. Springer, 1997.

[Big94] Ted Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *International Conference on Software Reuse, November, 1994*. IEEE Computing Society, 1994.

[BKdV03] Mark Bezem, Jan Willem Klop, and Roel de Vrijer. *Term Rewriting Systems*. Cambridge University Press, 2003.

[BM98] Jean Bézivin and Pierre-Alain Muller, editors. *UML'98: Beyond the Notation, Mulhouse, France*, volume 1618 of *LNCS*. Springer, 1998.

[BMA97] Davide Brugali, Giuseppe Menga, and Amund Aarsten. The framework life span. *Comm. ACM*, 40(10):65–68, 1997.

[BMR95] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Trans. on Softw. Eng.*, 21(10):785–798, 1995.

[Boe76] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, pages 1266–1241, 1976.

[Bol02] Christie Bolton. *On the refinement of state-based and event-based models*. PhD thesis, University of Oxford, 2002.

[Boo81] Grady Booch. Describing software design in Ada. *ACM Sigplan Notices*, 16(9), 1981.

[Boo86] Grady Booch. Object-oriented development. *IEEE Trans. on Softw. Eng.*, 12(2):211–221, 1986.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2nd edition, 1994.

[Bos00] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley, 2000.

[BP89] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software reusability: concepts and models*, volume 1. ACM Press, 1989.

[BPJS05] Richard Banach, Michael Poppleton, Czeslaw Jeske, and Susan Stepney. Retrenching the purse: Finite sequence numbers and the tower pattern. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *FM 2005, Newcastle, UK, July 2005*, volume 3582 of *LNCS*, pages 382–398. Springer, 2005.

[Bri96] Sjaak Brinkkemper. Method engineering: engineering of information systems development methods and tools. *Information and Software Technology*, 38:275–280, 1996.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[Bro75] Frederick P. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley, 1975.

[BS96] Anthony Bryant and Lesley Semmens, editors. *Method Integration Workshop, Leeds, UK*, Electronic Workshops in Computing (eWIC). Springer, 1996.

[BSC94] Rosalind Barden, Susan Stepney, and David Cooper. *Z in practice*. BCS Practitioner Series. Prentice Hall, 1994.

[BSW] Bad software web site. `http://www.badsoftware.com`. Last accessed on 24/01/2007.

[CAB+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[CD94] Steve Cook and John Daniels. *Designing Object Systems: object-oriented modelling with Syntropy*. The Object Oriented Series. Prentice Hall, 1994.

[CEK01] Tony Clark, Andy Evans, and Stuart Kent. The metamodelling language calculus: foundation semantics for UML. In Hussmann [Hus01], pages 17–31.

[CEK02] Tony Clark, Andy Evans, and Stuart Kent. Engineering modelling languages: A precise meta-modelling approach. In R.-D.Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *LNCS*, pages 159–173. Springer, 2002.

[CGR93] Dan Craigen, Susan Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods. Technical report, Atomic Energy Control Board of Canada, US National Institute of Standards and Technology, and US Naval Research Laboratories, 1993.

[CGR95] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Trans. on Softw. Eng.*, 21(2):90–98, 1995.

[Cha05] Robert N. Charette. Why software fails. *IEEE Spectrum*, September 2005.

[Che76] Peter Chen. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CM95] G. Cleland and D. MacKenzie. Inhibiting factors, market structure and the industrial uptake of formal methods. In *WIFT '95: Proceedings of the 1st Workshop on Industrial-Strength Formal Specification Techniques*, page 46. IEEE Computer Society, 1995.

[Coa92] Peter Coad. Object-oriented patterns. *Comm. ACM*, 35(9):152–159, 1992.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proc. of 3rd annual ACM symposium on Theory of Computing*, pages 151–158, 1971.

[Coo00] Steve Cook. The UML family: Profiles, prefaces and packages. In Evans et al. [EKS00], pages 255–264.

[CSW02] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. Refinement of actions in Circus. In *Electronic Notes in Theoretical Computer Science* [DBWvW02].

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[CW01] A. Clark and J. Warner, editors. *Object Modelling with the OCL*, volume 2263 of *LNCS*. Springer, 2001.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis*. Prentice Hall, 2nd edition, 1991.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99*, pages 411–420. IEEE, 1999.

[DB01] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: foundations and advanced applications*. FACIT. Springer, 2001.

[DBWvW02] John Derrick, Eerke Boiten, Jim Woodcock, and Joakim von Wright, editors. *Refine 2002: the BCS FACS Refinement Workshop*, volume 70 (3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002.

[DC02] Jim Davies and Charles Crichton. Concurrency and refinement in the UML. In Derrick et al. [DBWvW02].

[DDH72] O.-J. Dahl, E. W. Dijkstra, and Anthony Hoare. *Structured Programming*. Academic Press, 1972.

[DE96] D. Duke and A. S. Evans, editors. *BCS-FACS 2nd Northern Formal methods workshop*, Electronic Workshops in Computing. British Computer Society, 1996.

[Dem79] Tom Demarco. *Structured Analysis and System Specification*. Prentice Hall, 1979.

[Deu89] Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software reusability: applications and experience*, volume 2, pages 57–71. ACM Press, 1989.

[DH72] Ole-Johan Dahl and Anthony Hoare. Hierarchical program structures. In *Structured Programming* [DDH72], pages 175–220.

[DHR03] Moshe Deutsch, Martin Henson, and Steve Reeves. Operation refinement and monotonicity in the schema calculus. In Bert et al. [BBKW03], pages 103–126.

[Dij72] Edsger W. Dijkstra. Notes on structured programming. In *Structured Programming* [DDH72], pages 1–82.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[DL91] J. Dick and J. Loubersac. Integrating structured and formal methods: A visual approach to VDM. In Axel van Lamsweerde and Alfonso Fugetta, editors, *ESEC '91: European Softw. Eng. Conf.*, volume 550 of *LNCS*, pages 37–59. Springer, 1991.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioural subtyping through specification inheritance. In IEEE Computer Society, editor, *ICSE-18: 18th Int. Conf. on Software Engineering*, pages 258–267, 1996. Also as TR #95-20c Dept. Comp Science, Iowa State University.

[DLCP97] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Integrating OMT and Object-Z. In A. Evans and K. Lano, editors, *Proceedings of BCS FACS/EROS ROOM Workshop*, 1997.

[DR00] Roger Duke and Gordon Rose. *Formal Object-Oriented specification using Object-Z*. Cornerstones of Computing. MacMillan Press Limited, 2000.

[DSB99] Desmond D'Souza, Aamod Sane, and Alan Birchenough. First class extensibility for UML — packaging of profiles, stereotypes, patterns. In France and Rumpe [FR99], pages 265–291.

[Dup00] Sophie Dupuy. *Couplage de notations semi-formelles et formelles pour la spécification des systèmes d'information*. PhD thesis, Université Joseph Fourier, Grenoble I, 2000.

[DvL96] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *SIGSOFT '96*, pages 179–190. ACM Press, 1996.

[DW98] Desmond D'Sousa and A. C. Wills. *Object Components and Frameworks with UML: the Catalysis approach*. Addison-Wesley, 1998.

[Ede00] Ammon Eden. *Precise specification of design patterns and tool support in their application*. PhD thesis, Dept. Comp Science, Tel Aviv University, 2000.

[Ede01] Amnon Eden. Formal specification of object oriented design. In *Int. Conf. on Multidisciplinary Design in Engineering CSME-MDE 2001*, 2001.

[EFH83] Hartmut Ehrig, Werner Fey, and Horst Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83–03, Technical University of Berlin, Fachbereich Informatik, 1983.

[EFLR98] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. In Bézivin and Muller [BM98], pages 336–348.

[EKH01] Gregor Engels, Jochen M. Küster, and Reiko Heckel. A methodology for specifying and analysing consistency of object-oriented behavioral models. In *9th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 186–195, 2001.

[EKS00] Andy Evans, Stuart Kent, and Bran Selic, editors. *UML 2000: the Unified Modelling language, York, UK*, volume 1939 of *LNCS*. Springer, 2000.

[EL02] Lars-Henrik Eriksson and Peter Alexander Lindsay, editors. *FME 2002: Formal Methods – Getting it Right*, volume 2391 of *LNCS*. Springer, 2002.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[FBLPG97] Robert B. France, Jean-Michel Bruel, Maria M. Larrondo-Petrie, and Emmanuel Grant. Rigorous object-oriented modeling: Integrating formal and informal notations. In Michael Johnson, editor, *Algebraic Methodology and Software Technology (AMAST), Berlin*, volume 1349 of *LNCS*. Springer, 1997.

[FBR96] Robert B. France, Jean-Michel Bruel, and Gopal Raghavan. Towards rigorous analysis of fusion models: The MIRG experience. In Duke and Evans [DE96].

[FGB00] Robert B. France, Emanuel Grant, and Jean-Michel Bruel. UMLtranZ: An UML-based rigorous requirements modeling technique. Technical report, Colorado State University, 2000.

[FH01] Marc Frappier and Henri Habrias, editors. *Software Specification Methods: an overview using a case study*. FACIT. Springer, 2001.

[FKV91] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and formal requirements specification languages: Bridging the gap. *IEEE Trans. on Softw. Eng.*, 17(5):454–465, 1991.

[FL95] Philipe Facon and Régine Laleau. Des spécifications informelles aux spécifications formelles: compilation ou interprétation? In *Inforsid'95 – 13th conference INFORSID, Grenoble, France, Proceedings*, 1995.

[FLN96] Philippe Facon, Régine Laleau, and Hong Phuong Nguyen. Mapping object diagrams into B specifications. In Bryant and Semmens [BS96].

[FLN01] Philippe Facon, Régine Laleau, and Hong Phuong Nguyen. From OMT diagrams to B specifications. In Frappier and Habrias [FH01], pages 57–77.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Mathematical aspects of computer science: proceedings of symposia in applied mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[FLP97] Robert B. France and Maria M. Larrondo-Petrie. An integrated object-oriented and formal model environment. *Journal of Object-Oriented Programming*, 10(7):25–34, 1997.

[FMR01] Andres Flores, Richard Moore, and Luis Reynoso. A formal model of object-oriented design and GoF design patterns. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Int. Symposium of Formal Methods Europe*, volume 2021 of *LNCS*, pages 223–241. Springer, 2001.

[Fow97] Martin Fowler. *Analysis Patterns*. Addison-Wesley, 1997.

[FOW01] Clemens Fischer, Ernst-Rüdiger Olderog, and Heike Wehrheim. A CSP view on UML-RT structure diagrams. In Hussmann [Hus01], pages 91–108.

[Fow03] Martin Fowler. Patterns. *IEEE Softw.*, 20(2):56–57, 2003.

[FR99] Robert France and Bernhard Rumpe, editors. *UML'99: The Unified Modeling Language. 2nd Int. Conf.*, volume 1723 of *LNCS*. Springer, 1999.

[Fra92] Robert B. France. Semantically extended data flow diagrams: A formal specification tool. *IEEE Trans. on Softw. Eng.*, 18(4):329–346, 1992.

[FS00] Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley, 2000.

[FW00] Clemens Fischer and Heike Wehrheim. Behavioural subtyping relations for object-oriented formalisms. In T. Rus, editor, *AMAST 2000: Algebraic methodology and software technology*, volume 1816 of *LNCS*, pages 469–483. Springer, 2000.

[Gab96] Richard P. Gabriel. *Patterns of software: tales from the software community*. Oxford University Press, 1996.

[Gal96] Andy Galloway. *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-Perspective Systems*. PhD thesis, University of Teesside, School of Computing and Mathematics, 1996.

[GBR03] Martin Gogolla, Jörn Bohling, and Mark Richters. Validation of UML and OCL models by automatic snapshot generation. In Perdita Stevens and Jonathan Whittle, editors, *UML 2003*, volume 2683 of *LNCS*, pages 265–279. Springer, 2003.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and tools for formal specification*. Springer, 1993.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: abstraction and reuse of object-oriented design. In O. M. Nierstrsz, editor, *ECOOP'93: European Conf. on Object-Oriented Programming*, volume 707 of *LNCS*, pages 406–431. Springer, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Resusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing. Larch family of specification languages. *IEEE Softw.*, 2(5):24–36, 1985.

[GM02] C. George and H. Miao, editors. *ICFEM 2002: Int. Conf. of Formal Engineering Methods*, volume 2495 of *LNCS*. Springer, 2002.

[Gro92] Raise Language Group. *The Raise Specification Language*. BCS Practitioner series. Prentice Hall, 1992.

[GS93] David Gries and Fred B. Schneider. *A logical approach to discrete Math*. Springer, 1993.

[GS95] M. Goodland and C. Slater. *SSADM Version 4: a Practical Approach*. McGraw-Hill, 1995.

[GT79] J. A. Goguen and Joseph J. Tardo. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In *Proceedings Specifications of Reliable Software*, pages 170–189. IEEE Computer Society, 1979.

[Gua98] Nicola Guarino. Formal ontology and information systems. In *FOIS'98, Int. Conf. on Formal Ontology in Information Systems*, pages 3–15. IOS Press, 1998.

[GW96] Marie-Claude Gaudel and James Woodcock, editors. *FME'96: 3rd Int. Symposium of Formal Methods Europe*, volume 1051 of *LNCS*. Springer, 1996.

[Hal60] Paul Halmos. *Naive Set Theory*. Van Nostrand Reinhold, 1960.

[Hal90a] Anthony Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, 1990.

[Hal90b] Anthony Hall. Using Z as a specification calculus for object-oriented systems. In Anthony Hoare, Dines Bjørner, and Hans Langmaack, editors, *VDM '90*, volume 428 of *LNCS*, pages 290–318. Springer, 1990.

[Hal94] Anthony Hall. Specifying and interpreting class hierarchies in Z. In Bowen and Hall [BH94], pages 120–138.

[Hal96] Anthony Hall. Using formal methods to develop an ATC information system. *IEEE Softw.*, 13(2):66–76, 1996.

[Ham91] V. Hamilton. Experience of combining Yourdon and VDM. In *Proceedings of the Methods Integration Workshop*. Springer, 1991.

[Ham94] Jonathan Hammond. Producing Z specifications from object-oriented analysis. In Bowen and Hall [BH94], pages 316–336.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har92] David Harel. Biting the silver bullet: towards a brighter future for system development. *IEEE Computer*, 25(1):8–20, 1992.

[HH98] Anthony Hoare and Jifeng He. *Unifying Theories of Programming*. Series in Computer Science. Prentice Hall, 1998.

[HHB01] Rolf Hennicker, Heinrich Hussmann, and Michel Bidoit. On the precise meaning of OCL constraints. In Clark and Warner [CW01], pages 69–84.

[HHS86] Jifeng He, Anthony Hoare, and Jeff W. Sanders. Data refinement refined. In *ESOP'86*, volume 213 of *LNCS*, pages 187–196. Springer, 1986.

[HJ89] Ian Hayes and Cliff Jones. Specifications are not (necessarily) executable. *Softw. Eng. J.*, 4(6):330–338, 1989.

[HK02] David Harel and Orna Kupferman. On objects systems and behavioural inheritance. *IEEE Trans. on Softw. Eng.*, 28(9):889–903, 2002.

[HMP94] T. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Communication*, 112(2):273–337, 1994.

[Hoa69] Anthony Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[Hoa72a] Anthony Hoare. Notes on data structuring. In *Structured Programming* [DDH72], pages 83–174.

[Hoa72b] Anthony Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[Hoa78] Anthony Hoare. Software engineering: A keynote address. In *ICSE '78: Proceedings of the 3rd International Conference on Software Engineering*, pages 1–4, 1978.

[Hoa81] Anthony Hoare. The emperor's old clothes. *Comm. ACM*, 24(2):75–83, 1981. ACM Turing Award Lecture.

[Hoa85] Anthony Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoa96] Anthony Hoare. How did software get so reliable without proof? In Gaudel and Woodcock [GW96], pages 1–17.

[Hoa99] Anthony Hoare. Software: Barrier or frontier. available at `http://research.microsoft.com/~thoare/` (last accessed on 24/01/2007), 1999.

[HSB99] Brian Henderson-Sellers and Franck Barbier. Black and white diamonds. In France and Rumpe [FR99], pages 550–565.

[HTVW02] Ahmed Hammad, Bruno Tatibouët, Jean-Christophe Voisinet, and Wu Weiping. From B specification to UML statechart diagrams. In George and Miao [GM02], pages 511–522.

[Hus01] H. Hussmann, editor. *Fundamental Approaches to Software Engineering (FASE 2001)*, volume 2029 of *LNCS*. Springer, 2001.

[ISO96] ISO. Information technology — Vienna Development Method (VDM) — specification language, 1996. ISO/IEC 13817-1:1996, International Standard.

[ISO02a] ISO. Information processing systems—LOTOS — a formal description technique based on the temporal orderings of observational behaviour, 2002. ISO 8807:1989, International Standard.

[ISO02b] ISO. Information technology—Z formal specification notation—syntax, type system and semantics, 2002. ISO/IEC 13568:2002, Int. Standard.

[Jac95a] Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):365–389, 1995.

[Jac95b] Michael Jackson. *Software Requirements & Specifications*. Addison-Wesley, 1995.

[Jac98] Michael Jackson. Formal methods and traditional engineering. *The Journal of Systems and Software*, 40(3):191–194, 1998.

[Jac04] Daniel Jackson. Alloy 3.0 reference manual. available at http://alloy.mit.edu/, 2004.

[Jac06] Daniel Jackson. *Software Abstractions: logic, language and analysis*. MIT Press, 2006.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Object Technology series. Addison-Wesley, 1999.

[JCJÖ93] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1993.

[JD96] Daniel Jackson and Craig A. Damon. Elements of style: analyzing a software design feature with a counter-example detector. *IEEE Trans. on Softw. Eng.*, 22(7):484–495, 1996.

[JF88] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, 1988.

[JK04] Ho-Won Jung and Seung-Gweon Kim. Measuring software quality: a survey of ISO/IEC 9126. *IEEE Software*, 21(5):88–92, 2004.

[Joh92] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA'92*, pages 63–76, 1992.

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Comm. ACM*, 40(10):39–42, 1997.

[Jon91] Cliff Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1991.

[Jon99] Cliff Jones. Scientic decisions which characterize VDM. In Wing et al. [WWD99], pages 28–47.

[JR00] Daniel Jackson and Martin Rinard. Software analysis: a roadmap. In *Proc. of the Conf. on The future of Software engineering*, pages 133–145. ACM Press, 2000.

[JSS01] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *ACM SIGSOFT Foundation of Software Engineering/ Europoean Software Engineering Conf.*, 2001.

[KC99] Soon-Kyeong Kim and David Carrington. Formalizing the UML class diagram using Object-Z. In France and Rumpe [FR99], pages 83–98.

[KC00] Soon-Kyeong Kim and David Carrington. A formal mapping between UML models and Object-Z specifications. In Bowen et al. [BDGK00], pages 2–21.

[KC02a] Soon-Kyeong Kim and David Carrington. A formal metamodeling approach to a transformation between the UML state machine and Object-Z. In George and Miao [GM02], pages 548–560.

[KC02b] Soon-Kyeong Kim and David Carrington. A formal model of the UML metamodel: The UML state machine and its integrity constraints. In Bert et al. [BBHR02], pages 497–516.

[KC05] Soon-Kyeong Kim and David Carrington. A rigorous foundation for pattern-based design models. In Treharne et al. [TKHS05], pages 242–261.

[KFdB+04] Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing UML models and OCL constraints in PVS. In *2nd Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004)*, volume 115 of *Electronic Notes in Theoretical Computer Science*, pages 39–47. Elsevier, 2004.

[KHCP00] Steve King, Jonathan Hammond, Roderick Chapman, and Andy Pryor. Is proof more cost-effective than testing? *IEEE Trans. on Softw. Eng.*, 26(8):675–686, 2000.

[Kob99] Cris Kobryn. UML 2001: a standardization odyssey. *Comm. ACM*, 42(10):29–37, 1999.

[KP98] Cem Kaner and David Pels. *Bad Software: What to Do When Software Fails.* Wiley, 1998.

[Kru00] Philippe Kruchten. *The rational unified process :an introduction.* Object Technology series. Addison-Wesley, 2000.

[KWC98] Anneke Kleppe, Jos Warmer, and Steve Cook. Informal formality? The Object Constraint Language and its application in the UML metamodel. In Bézivin and Muller [BM98], pages 148–161.

[Kwo00] Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In Evans et al. [EKS00], pages 528–540.

[Lam89] Leslie Lamport. A simple approach to specify concurrent systems. *Comm. ACM*, 32(1):32–45, 1989.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[Lar98] Craig Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design.* Prentice Hall, 1998.

[LBG96] K. Lano, J.C. Bicarregui, and S. Goldsack. Formalising design patterns. In Duke and Evans [DE96].

[LH94] Kevin Lano and H. Haughton. Improving the process of system specification and development in B. In D. Till, editor, *6th Refinement Workshop*, Workshops in Computing. Springer, 1994.

[LHW96] Kevin Lano, H. Haughton, and P. Wheeler. Integrating formal and structured methods in object-oriented system development. In *Formal methods and Object Technology, chapter 7.* Springer, 1996.

[LLH02] Zhiming Liu, Xiaoshan Li, and Jifeng He. Using transition systems to unify UML models. In George and Miao [GM02], pages 535–547.

[LP01a] Régine Laleau and Fiona Polack. A rigorous metamodel for UML static conceptual modelling of information systems. In *CAiSE 2001: Advanced Information Systems Engineering*, volume 2068 of *LNCS*, pages 402–416. Springer, 2001.

[LP01b] Régine Laleau and Fiona Polack. Specification of integrity-preserving operations in information systems by using a formal UML-based language. *Information and Software Technology*, 43(12):693–704, 2001.

[LP02] Régine Laleau and Fiona Polack. Coming and going from UML to B: a proposal to support traceability in rigorous IS development. In Bert et al. [BBHR02], pages 517–534.

[Luc] Peter Lucas. Main approaches to formal specifications. In Dines Bjørner and Cliff Jones, editors, *Formal Specification and Software Development*, pages 3–23.

[Lup90] P. J. Lupton. Promoting forward simulation. In J. E. Nichols, editor, *Z User Workshop, Oxford*, pages 27–49. Springer, 1990.

[Lv85] David C. Luckham and Friedrich W. von Henke. Overview of Anna, a specification language for Ada. *IEEE Softw.*, 2(2):9–22, 1985.

[LvKP+91] Peter Gorm Larsen, Jan van Katwijk, Nico Plat, Kees Pronk, and Hans Toetenel. SVDM: An integrated combination of SA and VDM. In *Methods Integration Workshop*. Springer-Verlag, 1991.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):56–73, 1974.

[Mac96] Donald MacKenzie. Computer-related accidental death. In *Knowing machines: essays on technical change*, pages 185–213. MIT Press, 1996.

[Mac01] Donald MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, 2001.

[Mai00] T. Maibaum, editor. *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*. Springer, 2000.

[Mal98] A. Malioukov. An object-based approach to the B formal method. In Didier Bert, editor, *B'98 : Int. B Conf.*, volume 1393 of *LNCS*, pages 162–181. Springer, 1998.

[McC62] John McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, editor, *IFIP'62*, pages 21–28, 1962.

[McM92] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Dept. Comp Science, Carnegie Mellon University, 1992.

[Mey85] Bertrand Meyer. On formalism in specifications. *IEEE Softw.*, 2(1):6–26, 1985.

[Mey92] Bertrand Meyer. *Eiffel the language*. Prentice Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[Mik98] Tommi Mikkonen. Formalizing design patterns. In B. Werner, editor, *ICSE'98: Int. Conf. Softw. Eng.*, pages 115–124. IEEE Computer Society, 1998.

[Mil80] R. Milner. *A calculus of communicating systems*, volume 92 of *LNCS*. Springer, 1980.

[Mil89] Robin Milner. *Communication and concurrency*. International series in computer science. Prentice Hall, 1989.

[Mil93] R. Milne. The formal basis for the RAISE specification language. In D.J. Andrews, J. F. Groote, and C.A. Middleburg, editors, *International Workshop on Semantics of Specification Languages*, pages 23–50. Springer, 1993.

[Mil99] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[Mor94] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 2nd edition, 1994.

[Mos95] Simon Moser. Metamodels for object-oriented systems: a proposition of metamodels describing object-oriented systems at consecutive levels of abstraction. *Software concepts and tools*, 16:63–80, 1995.

[MP91] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer, 1991.

[MP95] Keith C. Mander and Fiona Polack. Rigorous specification using structured systems analysis and Z. *Information and Software Technology*, 37(5):285–291, 1995.

[MS99] Eric Meyer and Jeanine Souquières. Systematic approach to transform OMT diagrams to a B specification. In Wing et al. [WWD99], pages 875–895.

[Nau66] Peter Naur. Proof of algorithms by general snapshots. *BIT*, 6(4):310–316, 1966.

[Nei80] James M. Neighbors. *Software construction using components*. PhD thesis, Dept. Information and Computer science, University of California, 1980.

[Nei84] James M. Neighbors. The draco approach to constructing software from reusable components. *IEEE Trans. on Softw. Eng.*, 10(5):564–574, 1984.

[Nei89] James Neighbors. Draco: a method for engineering reusable software systems. In Biggerstaff and Perlis [BP89], pages 295–319.

[Ngu98] Hong Phuong Nguyen. *Dérivation De Spécifications Formelles B à partir de Spécifications Semi-Formelles*. PhD thesis, Laboratoire CEDRIC, Conservatoire National des Arts et Métiers, Evry, France, 1998.

[NIS02a] NIST. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology (NIST), 2002. `http://www.nist.gov/` (last accessed on 24/01/2007).

[NIS02b] NIST. NIST study: Software bugs take bite out of nation's economy. web, 2002. `http://www.nist.gov/public_affairs/update/upd20020624.htm` (last accessed on 24/01/2007).

[NR94] Naïma Nagui-Raïss. A formal software specification tool using the entity-relationship model. In P. Loucopoulos, editor, *ER'94: Entity-Relationship Approach*, volume 881 of *LNCS*, pages 316–332. Springer, 1994.

[Ö00] Gunnar Övergaard. Formal specification of object-oriented meta-modelling. In Maibaum [Mai00], pages 193–207.

[OMG03] OMG. *Unified Modeling Language Specification.* Object Management Group, 2003. Available at `http://www.uml.org/`.

[OMG06] OMG. *Object Constraint Language (OCL), version 2.0.* Object Management Group, 2006. Available at `http://www.uml.org/`.

[OSRSC99] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS langauge reference manual, version 2.3. Technical report, SRI International, 1999.

[Par72] David Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, 1972.

[Par76] David Parnas. On the design and development of program families. *IEEE Trans. on Softw. Eng.*, 2(1):1–9, 1976.

[Par77] David Parnas. The use of precise specifications in the development of software. In B Gilchrist, editor, *Information Processing (IFIP) 77*, pages 861–867. North-Holland Publishing Company, 1977.

[Par98] David Parnas. "Formal methods" technology transfer will fail. *The Journal of Systems and Software*, 40(3):195–198, 1998.

[Pet77] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3), 1977.

[PL90] Prezemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants.* Springer, 1990.

[Pnu81] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[PO01] Richard Paige and Jonathan Ostroff. Metamodelling and conformance checking with PVS. In Hussmann [Hus01], pages 2–16.

[Pol01] Fiona Polack. SAZ: SSADM version 4 and Z. In Frappier and Habrias [FH01], pages 21–38.

[PS99] Fiona Polack and Susan Stepney. Systems development using Z generics. In Wing et al. [WWD99], pages 1048–1067.

[PvKP91] Nico Plat, Jan van Karwijk, and Kees Pronk. A case for structured analysis/formal design. In Soren Prehn and Hans Toetenel, editors, *Proceedings of Formal Software Development Methods (VDM '91)*, volume 552 of *LNCS*, pages 81–105. Springer, 1991.

[PWM93] F. Polack, M. Whiston, and K. C. Mander. The SAZ project: Integrating SSADM and Z. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *LNCS*, pages 541–557. Springer, 1993.

[RA96] Mark Rawson and Pat Allen. Synthesis – an integrated, object-oriented method and tool for requirements specification. In Bryant and Semmens [BS96].

[RACH00] Gianna Reggio, Egidio Astesiano, C. Choppy, and H. Hussman. Analysing UML active classes and associated state machines – a lightweight formal approach. In Maibaum [Mai00], pages 127–146.

[RAE03] The challenges of complex IT projects. Technical report, Royal Academy of Engineering and the British Computer Society, 2003.

[RBP+91] James Rumbaugh, Michael Blaha, William Permerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RC01] Rajeev R. Raje and Sivakumar Chinnasamy. elelepus - a language for specification of software design patterns. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 600–604, 2001.

[Red05] Viviane Reding. Software and services: powering the European digital economy. `http://europa.eu.int/` (last accessed on 24/01/2007), September 2005. Official launch of the Technology Platform NESSI (Networked European Software and Services Initiative).

[RG98] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok-Wang Ling et al., editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.

[RG01] Mark Richters and Martin Gogolla. OCL: Syntax, semantics and tools. In Clark and Warner [CW01], pages 42–68.

[Ric01] Mark Richters. *A precise approach to validating UML models and OCL constraints*. PhD thesis, Universität Bremen, 2001.

[Rie00] Dirk Riehle. *Framework Design: a role modelling approach*. PhD thesis, ETH Zürich, 2000.

[RJ97] Don Roberts and Ralph E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[RM00] Luis Reynoso and Richard Moore. GoF behavioural patterns: a formal specification. Technical report, The United Nations University, UNU/IIST, 2000.

[Ros98] Andrew W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[RSM05] Rodrigo Ramos, Augusto Sampaio, and Alexandre Mota. A semantics for UML-RT active classes via mapping into Circus. In Martin Steffen and Gianluigi Zavattaro, editors, *FMOODS 2005*, volume 3535 of *LNCS*, pages 99–114. Springer, 2005.

[Rus95] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Comp. Science Lab., SRI Int., 1995.

[SA91] L. T. Semmens and P. Allen. Using Yourdon and Z : an approach to formal specification. In *Fifth Annual Z User Meeting, Oxford, 1990*. Springer, 1991.

[Saa97] M. Saaltink. The Z/EVES system. In Bowen and Hinchey [BH97], pages 72–85.

[SAJ+02] Eva Söderström, Birger Andersson, Paul Johannesson, Erik Perjons, and Benkt Wangler. Towards a framework for comparing process modelling languages. In Anne Banks Pidduck, John Mylopoulos, Carson C. Woo, and M. Tamer Özsu, editors, *CAiSE 2002*, volume 2348 of *LNCS*, pages 600–611. Springer, 2002.

[SB00] Colin Snook and Michael Butler. Tool-supported use of UML for constructing B specifications. `http://www.ecs.soton.ac.uk/~mjb/` (last accessed on 24/01/2007), 2000.

[SB06] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[SBC92] Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object orientation in Z*. Workshops in Computing. Springer, 1992.

[SC00] Susan Stepney and David Cooper. Formal methods for industrial products. In Bowen et al. [BDGK00], pages 374–393.

[Sch00] Steve Schneider. *Concurrent and real time systems: the CSP approach*. Wiley, 2000.

[Sch01] Steve Schneider. *The B-Method: an introduction*. Cornerstones of computing. Palgrave, 2001.

[Sch06] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[SCW00] Susan Stepney, David Cooper, and Jim Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, 2000.

[SFD92] L. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6), 1992.

[SFR05] Arnor Solberg, Robert France, and Raghu Reddy. Navigating the metamuddle. In *4th Workshop in Software Model Engineering (WiSME 2005)*, 2005.

[Sha84] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE Softw.*, 1(4), 1984.

[SM88] S. Shlaer and S. J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice Hall, 1988.

[SM92] S. Shlaer and S. J. Mellor. *Object Oriented Life Cycles: Modeling the World in States*. Prentice Hall, 1992.

[Smi95] Graeme Smith. A fully abstract semantics of class for Object-Z. *Formal Aspects of Computing*, 7(3):30–65, 1995.

[Smi00] Graeme Paul Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2001.

[SP00] Perdita Stevens and Rob Pooley. *Using UML: software engineering with objects and components*. Addison-Wesley, 2000.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[SPT03a] Susan Stepney, Fiona Polack, and Ian Toyn. Patterns to guide practical refactoring: examples targetting promotion in Z. In Bert et al. [BBKW03], pages 20–39.

[SPT03b] Susan Stepney, Fiona Polack, and Ian Toyn. A Z patterns catalogue I, specification and refactoring. Technical Report YCS-2003-349, Dept. of Comp Science, University of York, 2003.

[SR98] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime, 1998.

[SS71] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, 1971.

[SSA90] SSADM. *CCTA: SSADM Version 4 Reference Manual*. NCC Blackwell Ltd, 1990.

[ST02] Steve Schneider and Helen Treharne. Communicating B machines. In Bert et al. [BBHR02], pages 416–435.

[Ste02] Perdita Stevens. On the interpretation of binary associations in the unified modelling language. *Software and Systems Modeling*, 1(1):68–79, 2002.

[STE05] Steve Schneider, Helen Treharne, and Neil Evans. Chunks: Component verification in CSP‖ B. In Judi Romijn et al., editors, *IFM 2005: Integrated Formal Methods*, volume 3771 of *LNCS*, pages 89–108. Springer, 2005.

[Sto77] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey approach to programming language theory*. MIT Press, 1977.

[SW01] Barry Smith and Christopher Wely. Ontology: towards a new synthesis. In *FOIS'2001: Int. Conf. on Formal Ontology in Information Systems*, pages iii–ix. ACM Press, 2001.

[SWC02] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in Circus. In Eriksson and Lindsay [EL02], pages 451–470.

[Ten76] R. D. Tennent. The denotational semantics of programming languages. *Comm. ACM*, 19(8):437–453, 1976.

[TKHS05] Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors. *ZB 2005: Int. Conf. of B and Z users*, volume 3455 of *LNCS*. Springer, 2005.

[Tre02] Helen Treharne. Supplementing a UML development process with B. In Eriksson and Lindsay [EL02], pages 568–586.

[TS00] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Number 43 in Tracts in theoretical computer science. Cambridge University Press, 2000.

[TSB03] Helen Treharne, Steve Schneider, and Marchia Bramble. Composing specifications using communication. In Bert et al. [BBKW03], pages 58–78.

[Tur93] K. J. Turner, editor. *Using Formal Description Techniques: an introduction to Estelle, LOTOS, and SDL*. John Wiley & Sons, 1993.

[USE] USE. A UML-based specification environment. `http://www.db.informatik.uni-bremen.de/projects/USE/` (last accessed on 24/01/2007).

[Utt] Mark Utting. Data structures for Z testing tools. available at `http://www.cs.waikato.ac.nz/~marku/jaza/` (last accessed on 24/01/2007).

[UW03] Mark Utting and Shaochun Wang. Object orientation without extending Z. In Bert et al. [BBKW03], pages 319–338.

[VJ00] Mandana Vaziri and Daniel Jackson. Some shortcomings of OCL, the object constraint language of UML. In Q. Li et al., editors, *TOOLS: 34th Int. Conf.*, pages 555–560. IEEE Computer Society, 2000.

[VST04] Samuel H. Valentine, Susan Stepney, and Ian Toyn. A Z patterns catalogue II: definitions and laws, v0.1. Technical Report YCS-2004-383, Dept. of Comp Science, University of York, 2004.

[VTSK00] Samuel H. Valentine, Ian Toyn, Susan Stepney, and Steve King. Type-constrained generics for Z. In Bowen et al. [BDGK00], pages 250–263.

[Wan96] Yair Wand. Ontology as a foundation for meta-modelling and method engineering. *Information and Software Technology*, 38:281–287, 1996.

[Wat04] David A. Watt. *Programming language design concepts*. Wiley, 2004.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In Bert et al. [BBHR02], pages 184–203.

[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

[Web96] Matthias Weber. Combining statecharts and Z for the design of safety-critical control systems. In Gaudel and Woodcock [GW96], pages 307–326.

[Weh00] Heike Wehrheim. Behavioral subtyping and property preservation. In S. F. Smith and C. L. Talcott, editors, *FMOODS 2000: formal methods for object-based distributed systems*, pages 213–231. Kluwer, 2000.

[Wei71] Gerald M. Weinberg. *The Psychology of computer programming.* Van Nostrand Reinhold, 1971.

[Wie98] Roel Wieringa. A survey of structured and object-oriented software specification methods. *ACM Computing Surveys*, 30(4), 1998.

[Win90] Jeanette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Comm. ACM*, 14(4):221–227, 1971.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language: precise modeling with UML.* Addison-Wesley, 1999.

[WN95] K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Development.* Prentice Hall, 1995.

[Woo89] Jim Woodcock. Structuring specifications in Z. *Software Engineering Journal*, 4(1):51–66, 1989.

[WWD99] Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *FM'99, Toulouse, France*, volume 1708 and 1709 of *LNCS*. Springer, 1999.

[ZJ93] Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Trans. on Soft. Eng. and Methodology*, 2(4):379–4111, 1993.

A

Formal definition of FTL and Instantiation Calculus

This appendix complements the definitions of FTL given in chapter 3. That is, what is not defined in that chapter, using the BNF and equational style of presentation used in the main text, is defined here. In addition, this appendix includes the full Z definition of FTL and its instantiation calculus (IC), which is the definition that was type-checked and tested with proof using the Z/Eves [Saa97] prover. So, this chapter defines the same things twice using two different styles of presentation: one in BNF and the equational style (that is easier to read) and another in Z.

The following starts with the definition of the syntax of FTL, then it defines the semantics, and, finally, the instantiation calculus is also defined.

A.1 Syntax

The following defines the Syntax of FTL using BNF (Backus-Naur form) and Z.

A.1.1 BNF Definition

$[I, SYMB]$
$Str == \text{seq}\, SYMB$
$\Lambda \ : \ Str$

$$E ::= A \mid C \mid A\ E \mid C\ E$$
$$C ::= (\!|\ E\ |\!)^{?} \mid (\!|\ CL\ |\!)$$
$$CL ::= E_1\ [\!]\ E_2 \mid E\ [\!]\ CL$$
$$A ::= \lessdot I \gtrdot \mid T \mid L$$
$$L ::= [\![\ LT\ ]\!]_{(SEP,EI)} \mid [\![\ LT\ ]\!]$$
$$LT ::= A \mid A\ LT$$
$$[\![\ LT\ ]\!] = [\![\ LT\ ]\!]_{(\Lambda,\Lambda)}$$
$$SEP ::= Str$$
$$EI ::= Str$$
$$T ::= Str$$

A.1.2 Z Definition

$$[I, SYMB]$$
$$Str == \operatorname{seq} SYMB$$

$$\Lambda : Str \vdash \Lambda = \langle\rangle$$

$$A ::= param\langle\!\langle I \rangle\!\rangle \mid tx\langle\!\langle Str \rangle\!\rangle \mid ls\langle\!\langle L \rangle\!\rangle$$
$$L ::= list\langle\!\langle LT \times Str \times Str \rangle\!\rangle \mid listr\langle\!\langle LT \rangle\!\rangle$$
$$LT ::= at\langle\!\langle A \rangle\!\rangle \mid lat\langle\!\langle A \times LT \rangle\!\rangle$$
$$CL ::= chs\langle\!\langle E \times E \rangle\!\rangle \mid lchs\langle\!\langle E \times CL \rangle\!\rangle$$
$$C ::= och\langle\!\langle E \rangle\!\rangle \mid mch\langle\!\langle CL \rangle\!\rangle$$
$$E ::= eat\langle\!\langle A \rangle\!\rangle \mid ech\langle\!\langle C \rangle\!\rangle \mid eats\langle\!\langle A \times E \rangle\!\rangle \mid echs\langle\!\langle C \times E \rangle\!\rangle$$

$$\vdash \forall\, lt : LT \bullet listr\ lt = list(lt, \Lambda, \Lambda)$$

A.2 Semantics

A.2.1 Definitions in Equational Style

Auxiliary definitions

The string concatenation is introduced to facilitate string manipulation:

$$\_ \mathbin{+\!\!+} \_ : (Str \times Str) \to Str$$

This is defined as sequence concatenation.

A concatenation operator for expressions is also required. It takes a pair of expressions and returns another expression:

$$\_ \mathbin{+\!\!+}_E \_ : E \times E \to E$$

This is defined by the equations:

$$
\begin{aligned}
&A +\!\!+_E\ E = A\ E \\
&C +\!\!+_E\ E = C\ E \\
&(A\ E_1) +\!\!+_E\ E_2 = A\ (E_1 +\!\!+_E\ E_2) \\
&(C\ E_1) +\!\!+_E\ E_2 = C\ (E_1 +\!\!+_E\ E_2)
\end{aligned}
$$

These equations build an expression from the two expressions given as arguments. If the first argument expression is an atom or a choice, then the resulting expression is the atom or choice followed by the second expression. And if the first argument expression is an atom or choice followed by another expression, then the result is the atom or choice followed by the concatenation of the remaining first expression with the second.

To define the semantics, we need to know the set of variables of a template formula. For example, the expression $\ll x \gg : \ll t \gg$ has the variable set $\{x, t\}$. So, there are functions that give the set of variables of a template formula; one function for each syntactic construct of FTL, where each is defined by structural induction. The functions for the syntactic set of template atoms (A),

$$\mathcal{V}_{\mathcal{A}} : A \to \mathbb{P}\ I$$

is defined by the equations:

$$
\begin{aligned}
&\mathcal{V}_{\mathcal{A}}(\ll I \gg) = \{I\} \\
&\mathcal{V}_{\mathcal{A}}(T) = \varnothing \\
&\mathcal{V}_{\mathcal{A}}(L) = \varnothing
\end{aligned}
$$

If the atom is a placeholder then the variable set is the placeholder's variable. If it is text or a list then the variable set is the empty set.

The function that gives the variable set of template expressions,

$$\mathcal{V}_{\mathcal{E}} : E \to \mathbb{P}\ I$$

is defined by the equations:

$$
\begin{aligned}
&\mathcal{V}_{\mathcal{E}}(A) = \mathcal{V}_{\mathcal{A}}(A) \\
&\mathcal{V}_{\mathcal{E}}(C) = \mathcal{V}_{\mathcal{C}}(C) \\
&\mathcal{V}_{\mathcal{E}}(A\ E) = \mathcal{V}_{\mathcal{A}}(A) \cup \mathcal{V}_{\mathcal{E}}(E) \\
&\mathcal{V}_{\mathcal{E}}(C\ E) = \mathcal{V}_{\mathcal{C}}(C) \cup \mathcal{V}_{\mathcal{E}}(E)
\end{aligned}
$$

If the expression is an atom then its set of variables is the set of variables of the atom, and similarly for the choice. If it is an atom or choice followed by another expression then the variable set is the union of the variable set of the atom or choice with the variable set of the remaining expression.

The remaining functions are similarly defined. See below for their full Z definition.

A.2.2 Z Definition

Auxiliary definitions

$$\_ +\!\!+ \_ : (Str \times Str) \rightarrow Str$$
$$(\_ +\!\!+ \_) = (\_ \frown \_)$$

$$\_ +\!\!+_E \_ : E \times E \rightarrow E$$
$$\forall\, a : A;\ e : E \bullet (eat\ a) +\!\!+_E\ e = eats(a, e)$$
$$\forall\, c : C;\ e : E \bullet (ech\ c) +\!\!+_E\ e = echs(c, e)$$
$$\forall\, a : A;\ e_1, e_2 : E \bullet eats(a, e_1) +\!\!+_E\ e_2 = eats(a, e_1 +\!\!+_E\ e_2)$$
$$\forall\, c : C;\ e_1, e_2 : E \bullet echs(c, e_1) +\!\!+_E\ e_2 = echs(c, e_1 +\!\!+_E\ e_2)$$

$$\mathcal{V}_{\mathcal{A}} : A \rightarrow \mathbb{P}\, I$$
$$\forall\, i : I \bullet \mathcal{V}_{\mathcal{A}}(param\ i) = \{i\}$$
$$\forall\, s : Str \bullet \mathcal{V}_{\mathcal{A}}(tx\ s) = \varnothing$$
$$\forall\, l : L \bullet \mathcal{V}_{\mathcal{A}}(ls\ l) = \varnothing$$

$$\mathcal{V}_{\mathcal{LT}} : LT \rightarrow \mathbb{P}\, I$$
$$\forall\, a : A \bullet \mathcal{V}_{\mathcal{LT}}(at\ a) = \mathcal{V}_{\mathcal{A}}\ a$$
$$\forall\, a : A;\ le : LT \bullet \mathcal{V}_{\mathcal{LT}}(lat(a, le)) = \mathcal{V}_{\mathcal{A}}\ a \cup \mathcal{V}_{\mathcal{LT}}\ le$$

$$\mathcal{V}_{\mathcal{C}} : C \rightarrow \mathbb{P}\, I$$
$$\mathcal{V}_{\mathcal{CL}} : CL \rightarrow \mathbb{P}\, I$$
$$\mathcal{V}_{\mathcal{E}} : E \rightarrow \mathbb{P}\, I$$
$$\forall\, e : E \bullet \mathcal{V}_{\mathcal{C}}(och\ e) = \mathcal{V}_{\mathcal{E}}\ e$$
$$\forall\, cl : CL \bullet \mathcal{V}_{\mathcal{C}}(mch\ cl) = \mathcal{V}_{\mathcal{CL}}\ cl$$
$$\forall\, e_1, e_2 : E \bullet \mathcal{V}_{\mathcal{CL}}(chs(e_1, e_2)) = \mathcal{V}_{\mathcal{E}}\ e_1 \cup \mathcal{V}_{\mathcal{E}}\ e_2$$
$$\forall\, e_1 : E;\ cl : CL \bullet \mathcal{V}_{\mathcal{CL}}(lchs(e_1, cl)) = \mathcal{V}_{\mathcal{E}}\ e_1 \cup \mathcal{V}_{\mathcal{CL}}\ cl$$
$$\forall\, a : A \bullet \mathcal{V}_{\mathcal{E}}(eat\ a) = \mathcal{V}_{\mathcal{A}}\ a$$
$$\forall\, c : C \bullet \mathcal{V}_{\mathcal{E}}(ech\ c) = \mathcal{V}_{\mathcal{C}}\ c$$
$$\forall\, a : A;\ e : E \bullet \mathcal{V}_{\mathcal{E}}(eats(a, e)) = \mathcal{V}_{\mathcal{A}}\ a \cup \mathcal{V}_{\mathcal{E}}\ e$$
$$\forall\, c : C;\ e : E \bullet \mathcal{V}_{\mathcal{E}}(echs(c, e)) = \mathcal{V}_{\mathcal{C}}\ c \cup \mathcal{V}_{\mathcal{E}}\ e$$

Semantic functions

$$Env == I \nrightarrow Str$$
$$TreeEnv ::= \mathsf{tree}\langle\langle Env \times \mathrm{seq}\ TreeEnv \rangle\rangle$$
$$GEnv == \mathrm{seq}\,\mathbb{N} \times TreeEnv$$

$$\mathcal{M}_{\mathcal{A}} : A \to TreeEnv \nrightarrow Str$$
$$\mathcal{M}_{\mathcal{L}} : L \to \mathrm{seq}\ TreeEnv \nrightarrow Str$$
$$\mathcal{M}_{\mathcal{LT}} : LT \to TreeEnv \nrightarrow Str$$

$$\forall\, t : Str;\ env : Env;\ lte : \mathrm{seq}\ TreeEnv \bullet \mathcal{M}_{\mathcal{A}}(tx\ t)(\mathsf{tree}(env, lte)) = t$$
$$\forall\, i : I;\ env : Env;\ lte : \mathrm{seq}\ TreeEnv \bullet \mathcal{M}_{\mathcal{A}}(param\ i)(\mathsf{tree}(env, lte)) = env\ i$$
$$\forall\, l : L;\ env : Env;\ lte : \mathrm{seq}\ TreeEnv \bullet \mathcal{M}_{\mathcal{A}}(ls\ l)(\mathsf{tree}(env, lte)) = \mathcal{M}_{\mathcal{L}}\ l\ lte$$
$$\forall\, le : LT;\ ld, let : Str \bullet \mathcal{M}_{\mathcal{L}}(list(le, ld, let))\langle\rangle = let$$
$$\forall\, le : LT;\ ld, let : Str;\ env : Env;\ lte : \mathrm{seq}\ TreeEnv \bullet$$
$$\mathcal{M}_{\mathcal{L}}(list(le, ld, let))\langle(\mathsf{tree}(env, lte))\rangle =$$
$$\mathsf{if}\ \neg\, \mathcal{V}_{\mathcal{LT}}\ le \cap \mathrm{dom}\ env = \varnothing\ \ \mathsf{then}$$
$$\mathcal{M}_{\mathcal{LT}}(le)(\mathsf{tree}(env, lte))$$
$$\mathsf{else}\ \ let$$
$$\forall\, le : LT;\ ld, let : Str;\ env : Env;\ ltes, ltet : \mathrm{seq}\ TreeEnv \bullet$$
$$\mathcal{M}_{\mathcal{L}}(list(le, ld, let))(\langle\mathsf{tree}(env, ltes)\rangle \frown ltet) =$$
$$\mathsf{if}\ \ \neg\, \mathcal{V}_{\mathcal{LT}} le \cap \mathrm{dom}\ env = \varnothing\ \ \mathsf{then}$$
$$\mathcal{M}_{\mathcal{LT}}\ le\ (\mathsf{tree}(env, ltes))\ +\!\!+\ \mathcal{M}_{\mathcal{L}}\ (list(lat((tx\ ld), le), \Lambda, \Lambda))ltet$$
$$\mathsf{else}\ \ let$$
$$\forall\, a : A;\ te : TreeEnv \bullet \mathcal{M}_{\mathcal{LT}}(at\ a)te = \mathcal{M}_{\mathcal{A}}\ a\ te$$
$$\forall\, a : A;\ le : LT;\ te : TreeEnv \bullet \mathcal{M}_{\mathcal{LT}}(lat(a, le))te = \mathcal{M}_{\mathcal{A}}\ a\ te +\!\!+ \mathcal{M}_{\mathcal{LT}}\ le\ te$$

$$\mathcal{M}_{\mathcal{CL}} : CL \to \mathbb{N}_1 \nrightarrow E$$

$$\forall\, e_1, e_2 : E \bullet \mathcal{M}_{\mathcal{CL}}(chs(e_1, e_2))1 = e_1$$
$$\forall\, e_1, e_2 : E \bullet \mathcal{M}_{\mathcal{CL}}(chs(e_1, e_2))2 = e_2$$
$$\forall\, e : E;\ cl : CL;\ n : \mathbb{N}_1 \bullet \mathcal{M}_{\mathcal{CL}}(lchs(e, cl))n = \ \mathsf{if}\ \ n = 1\ \ \mathsf{then}\ \ e\ \ \mathsf{else}\ \ \mathcal{M}_{\mathcal{CL}}\ cl(n-1)$$

$$\mathcal{M}_{\mathcal{C}} : C \to \mathbb{N} \nrightarrow E$$

$$\forall\, e : E \bullet \mathcal{M}_{\mathcal{C}}(och\ e)\ 0 = eat(tx\ \Lambda)$$
$$\forall\, e : E;\ n : \mathbb{N}_1 \bullet \mathcal{M}_{\mathcal{C}}(och\ e)\ n = e$$
$$\forall\, cl : CL;\ n : \mathbb{N}_1 \bullet \mathcal{M}_{\mathcal{C}}(mch\ cl)\ n = \mathcal{M}_{\mathcal{CL}}\ cl\ n$$

$$\mathcal{M}_{\mathcal{E}} : E \to GEnv \to Str$$

$$\forall\, a : A;\ chs : \mathrm{seq}\,\mathbb{N};\ te : TreeEnv \bullet \mathcal{M}_{\mathcal{E}}(eat\ a)(chs, te) = \mathcal{M}_{\mathcal{A}}\ a\ te$$
$$\forall\, c : C;\ n : \mathbb{N};\ te : TreeEnv \bullet \mathcal{M}_{\mathcal{E}}(ech\ c)(\langle n\rangle, te) = \mathcal{M}_{\mathcal{E}}(\mathcal{M}_{\mathcal{C}}\ c\ n)(\langle\,\rangle, te)$$
$$\forall\, c : C;\ n : \mathbb{N};\ chs : \mathrm{seq}\,\mathbb{N};\ te : TreeEnv \bullet$$
$$\mathcal{M}_{\mathcal{E}}(ech\ c)((\langle n\rangle \frown chs), te) = \mathcal{M}_{\mathcal{E}}(\mathcal{M}_{\mathcal{C}}\ c\ n)(chs, te)$$
$$\forall\, a : A;\ e : E;\ chs : \mathrm{seq}\,\mathbb{N};\ te : TreeEnv \bullet$$
$$\mathcal{M}_{\mathcal{E}}(eats(a, e))(chs, te) = \mathcal{M}_{\mathcal{A}}\ a\ te +\!\!+ \mathcal{M}_{\mathcal{E}}\ e(chs, te)$$
$$\forall\, c : C;\ e : E;\ n : \mathbb{N};\ te : TreeEnv \bullet$$
$$\mathcal{M}_{\mathcal{E}}(echs(c, e))(\langle n\rangle, te) = \mathcal{M}_{\mathcal{E}}((\mathcal{M}_{\mathcal{C}}\ c\ n) +\!\!+_E\ e)(\langle\rangle, te)$$
$$\forall\, c : C;\ e : E;\ n : \mathbb{N};\ chs : \mathrm{seq}\,\mathbb{N};\ te : TreeEnv \bullet$$
$$\mathcal{M}_{\mathcal{E}}(echs(c, e))((\langle n\rangle \frown chs), te) = \mathcal{M}_{\mathcal{E}}((\mathcal{M}_{\mathcal{C}}\ c\ n) +\!\!+_E\ e)(chs, te)$$

A.3 The Instantiation Calculus

A.3.1 Definitions in Equational style

Variable Renaming

$MI, SI : \mathbb{P}\ I$
$\langle MI, SI \rangle$ partitions I

$subI : I \times \mathbb{N}_1 \rightarrowtail SI$

$\mathcal{R}\mathbf{n}_{\mathcal{A}} : \mathbf{A} \rightarrow \mathbb{N}_{\mathbf{1}} \rightarrow \mathbf{A}$

$\mathcal{R}n_{\mathcal{A}}\ \lessdot I \gtrdot\ n = \lessdot subI(I, n) \gtrdot$
$\mathcal{R}n_{\mathcal{A}}\ (T)\ n = T$
$\mathcal{R}n_{\mathcal{A}}\ (L)\ n = \mathcal{R}n_{\mathcal{L}}\ (L)\ n$

$\mathcal{R}\mathbf{n}_{\mathcal{L}} : \mathbf{L} \rightarrow \mathbb{N} \rightarrow \mathbf{L}$

$\mathcal{R}n_{\mathcal{L}}\ (\llbracket\ LT\ \rrbracket_{(SEP,EI)})\ n = \llbracket\ \mathcal{R}n_{\mathcal{LT}}\ LT\ n\ \rrbracket_{(SEP,EI)}$
$\mathcal{R}n_{\mathcal{L}}\ (\llbracket\ LT\ \rrbracket)\ n = \llbracket\ \mathcal{R}n_{\mathcal{LT}}\ LT\ n\ \rrbracket$

$\mathcal{R}\mathbf{n}_{\mathcal{LT}} : \mathbf{LT} \rightarrow \mathbb{N} \rightarrow \mathbf{LT}$

$\mathcal{R}n_{\mathcal{LT}}(A)\ n = (\mathcal{R}n_{\mathcal{A}}\ A\ n)$
$\mathcal{R}n_{\mathcal{LT}}(A\ LT)\ n = (\mathcal{R}n_{\mathcal{A}}\ A\ n)\ (\mathcal{R}n_{\mathcal{LT}}\ LT\ n)$

$\mathcal{R}\mathbf{n}_{\mathcal{CL}} : \mathbf{CL} \rightarrow \mathbb{N} \rightarrow \mathbf{CL}$

$\mathcal{R}n_{\mathcal{CL}}(E_1 \ [\!]\ E_2)\ n = \mathcal{R}n_{\mathcal{E}}(E_1, n)\ [\!]\ \mathcal{R}n_{\mathcal{E}}(E_2, n)$
$\mathcal{R}n_{\mathcal{CL}}(E\ [\!]\ CL)\ n = \mathcal{R}n_{\mathcal{E}}(E, n)\ [\!]\ \mathcal{R}n_{\mathcal{CL}}(CL, n)$

$\mathcal{R}\mathbf{n}_{\mathcal{C}} : \mathbf{C} \rightarrow \mathbb{N} \rightarrow \mathbf{C}$

$\mathcal{R}n_{\mathcal{C}}(\!|\ E\ |\!)^?\ n = (\!|\ \mathcal{R}n_{\mathcal{E}}(E, n)\ |\!)^?$
$\mathcal{R}n_{\mathcal{C}}(\!|\ CL\ |\!)\ n = (\!|\ \mathcal{R}n_{\mathcal{CL}}(CL, n)\ |\!)$

$\mathcal{R}\mathbf{n}_{\mathcal{E}} : \mathbf{E} \rightarrow \mathbb{N} \rightarrow \mathbf{E}$

$\mathcal{R}n_{\mathcal{E}}(A)\ n = \mathcal{R}n_{\mathcal{A}}\ (A)\ n$
$\mathcal{R}n_{\mathcal{E}}(C)\ n = \mathcal{R}n_{\mathcal{C}}\ (C)\ n$
$\mathcal{R}n_{\mathcal{E}}(A, E)\ n = (\mathcal{R}n_{\mathcal{A}}(A)\ n)\ (\mathcal{R}n_{\mathcal{E}}(E)\ n)$
$\mathcal{R}n_{\mathcal{E}}(C, E)\ n = (\mathcal{R}n_{\mathcal{C}}(C)\ n)\ (\mathcal{R}n_{\mathcal{E}}(E)\ n)$

List Dependencies

$$\mathcal{DV}_{\mathcal{L}} : \mathbf{L} \to \mathbb{P}\,\mathbf{I} \to \mathbb{P}\,\mathbf{I}$$

$$\mathcal{DV}_{\mathcal{L}}([\![\ LT\ ]\!]_{(SEP,EI)})is = \begin{cases} \mathcal{V}_{\mathcal{LT}}\ LT \cup is & \mathcal{V}_{\mathcal{LT}}\ LT \cap is \neq \varnothing \\ is & \text{otherwise} \end{cases}$$

$$\mathcal{DV}_{\mathcal{A}} : \mathbf{A} \to \mathbb{P}\,\mathbf{I} \to \mathbb{P}\,\mathbf{I}$$

$$\mathcal{DV}_{\mathcal{A}}(\ll I \gg)\ is = is$$
$$\mathcal{DV}_{\mathcal{A}}(T)\ is = is$$
$$\mathcal{DV}_{\mathcal{A}}(L)\ is = \mathcal{DV}_{\mathcal{L}}\ L\ is$$

$$\mathcal{DV}_{\mathcal{CL}} : \mathbf{CL} \to \mathbb{P}\,\mathbf{I} \to \mathbb{P}\,\mathbf{I}$$

$$\mathcal{DV}_{\mathcal{CL}}(E_1\ [\!]\ E_2)\ is = \mathcal{DV}_{\mathcal{E}}\ E_1\ is \cup \mathcal{DV}_{\mathcal{E}}\ E_2\ is$$
$$\mathcal{DV}_{\mathcal{CL}}(E\ [\!]\ CL)\ is = \mathcal{DV}_{\mathcal{E}}\ E\ is \cup \mathcal{DV}_{\mathcal{CL}}\ CL\ is$$

$$\mathcal{DV}_{\mathcal{C}} : \mathbf{C} \to \mathbb{P}\,\mathbf{I} \to \mathbb{P}\,\mathbf{I}$$

$$\mathcal{DV}_{\mathcal{C}}([\![\ E\ ]\!]^{?})\ is = \mathcal{DV}_{\mathcal{E}}\ E\ is$$
$$\mathcal{DV}_{\mathcal{C}}([\![\ CL\ ]\!])\ is = \mathcal{DV}_{\mathcal{CL}}\ CL\ is$$

$$\mathcal{DV}_{\mathcal{E}} : \mathbf{E} \to \mathbb{P}\,\mathbf{I} \to \mathbb{P}\,\mathbf{I}$$

$$\mathcal{DV}_{\mathcal{E}}(A)\ is = \mathcal{DV}_{\mathcal{A}}\ A\ is$$
$$\mathcal{DV}_{\mathcal{E}}(C)\ is = \mathcal{DV}_{\mathcal{C}}\ C\ is$$
$$\mathcal{DV}_{\mathcal{E}}(A, E)\ is = \mathcal{DV}_{\mathcal{E}}\ E(\mathcal{DV}_{\mathcal{A}}\ A\ is)$$
$$\mathcal{DV}_{\mathcal{E}}(C, E)\ is = \mathcal{DV}_{\mathcal{E}}\ E(\mathcal{DV}_{\mathcal{C}}\ C\ is)$$

Conversion from list term to expression

The instantiation functions need to convert the list term to an expression so that the list term can be instantiated. This conversion function has the obvious signature, a total function from list terms (LT) to template expressions (E):

$$\mathcal{LT}_2\mathcal{E} : LT \to E$$

This is defined by the equations:

$$\mathcal{LT}_2\mathcal{E}(A) = A$$
$$\mathcal{LT}_2\mathcal{E}(A\ LT) = A\ (\mathcal{LT}_2\mathcal{E}\ LT)$$

If the list term is an atom, then the atom is returned. If it is an atom followed by the remaining list term, then the atom followed by the conversion to expression of the remaining list term (recursive call to $\mathcal{LT}_2\mathcal{E}$) is returned.

Choice instantiation functions

$$\mathcal{IL}_{\mathcal{CL}} : CL \to LEnv \to CL$$

$$\mathcal{IL}_{\mathcal{CL}}(E_1 \,[\!]\, E_2)\ le = \mathcal{IL}_{\mathcal{E}0}\ E_1\ le \,[\!]\, \mathcal{IL}_{\mathcal{E}0}\ E_2\ le$$
$$\mathcal{IL}_{\mathcal{CL}}(E_1 \,[\!]\, CL)\ le = \mathcal{IL}_{\mathcal{E}0}\ E_1\ le \,[\!]\, \mathcal{IL}_{\mathcal{CL}}\ CL\ le$$

$$\mathcal{IL}_{\mathcal{C}} : C \to LEnv \to C$$

$$\mathcal{IL}_{\mathcal{C}}(\!|\ E\ |\!)^?\ le = (\!|\ \mathcal{IL}_{\mathcal{E}0}\ E\ le\ |\!)^?$$
$$\mathcal{IL}_{\mathcal{C}}(\!|\ CL\ |\!)\ le = (\!|\ \mathcal{IL}_{\mathcal{CL}}\ CL\ le\ |\!)$$

A.3.2 Z Definition

Parameters

$$\mathcal{IP}_{\mathcal{A}} : A \to Env \to A$$

$$\forall\, t : Str;\ env : Env \bullet \mathcal{IP}_{\mathcal{A}}(tx\ t)env = tx\ t$$
$$\forall\, i : I;\ env : Env \bullet \mathcal{IP}_{\mathcal{A}}(param\ i)env = \mathsf{if}\ i \in \mathrm{dom}\ env\ \mathsf{then}\ tx(env\ i)\ \mathsf{else}\ param\ i$$
$$\forall\, l : L;\ env : Env \bullet \mathcal{IP}_{\mathcal{A}}(ls\ l)env = ls\ l$$

$$\mathcal{IP}_{\mathcal{E}} : E \to Env \to E$$
$$\mathcal{IP}_{\mathcal{C}} : C \to Env \to C$$
$$\mathcal{IP}_{\mathcal{CL}} : CL \to Env \to CL$$

$$\forall\, e_1, e_2 : E;\ env : Env \bullet \mathcal{IP}_{\mathcal{CL}}(chs(e_1, e_2))env = chs((\mathcal{IP}_{\mathcal{E}}\ e_1\ env), (\mathcal{IP}_{\mathcal{E}}\ e_2\ env))$$
$$\forall\, e_1 : E;\ cl : CL;\ env : Env \bullet \mathcal{IP}_{\mathcal{CL}}(lchs(e_1, cl))env = lchs((\mathcal{IP}_{\mathcal{E}}\ e_1\ env), (\mathcal{IP}_{\mathcal{CL}}\ cl\ env))$$
$$\forall\, e : E;\ env : Env \bullet \mathcal{IP}_{\mathcal{C}}(och\ e)env = och((\mathcal{IP}_{\mathcal{E}} e)env)$$
$$\forall\, cl : CL;\ env : Env \bullet \mathcal{IP}_{\mathcal{C}}(mch\ cl)env = mch(\mathcal{IP}_{\mathcal{CL}}\ cl\ env)$$
$$\forall\, a : A;\ env : Env \bullet \mathcal{IP}_{\mathcal{E}}(eat\ a)env = eat(\mathcal{IP}_{\mathcal{A}}\ a\ env)$$
$$\forall\, c : C;\ env : Env \bullet \mathcal{IP}_{\mathcal{E}}(ech\ c)env = ech(\mathcal{IP}_{\mathcal{C}}\ c\ env)$$
$$\forall\, a : A;\ e : E;\ env : Env \bullet \mathcal{IP}_{\mathcal{E}}(eats(a, e))env = eats((\mathcal{IP}_{\mathcal{A}}\ a\ env), (\mathcal{IP}_{\mathcal{E}}\ e\ env))$$
$$\forall\, c : C;\ e : E;\ env : Env \bullet \mathcal{IP}_{\mathcal{E}}(echs(c, e))env = echs((\mathcal{IP}_{\mathcal{C}}\ c\ env), (\mathcal{IP}_{\mathcal{E}}\ e\ env))$$

Lists

Parameter Renaming.

$$MI, SI : \mathbb{P}\ I$$

$$\langle MI, SI\rangle \,\mathrm{partitions}\, I$$

$$subI : I \times \mathbb{N}_1 \rightarrowtail SI$$

$$
\begin{array}{l}
\mathcal{R}n_{\mathcal{A}} : A \to \mathbb{N} \to A \\
\mathcal{R}n_{\mathcal{L}} : L \to \mathbb{N} \to L \\
\mathcal{R}n_{\mathcal{LT}} : LT \to \mathbb{N} \to LT \\
\hline
\forall\, i : I;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{A}}(param\ i)\ n = param(subI(i, n)) \\
\forall\, t : Str;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{A}}(tx\ t)\ n = tx\ t \\
\forall\, l : L;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{A}}(ls\ l)\ n = ls(\mathcal{R}n_{\mathcal{L}}\ l\ n) \\
\forall\, le : LT;\ sep, ei : Str;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{L}}(list(le, sep, ei))\ n = list((\mathcal{R}n_{\mathcal{LT}}\ le\ n), sep, ei) \\
\forall\, le : LT;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{L}}(listr\ le)\ n = listr(\mathcal{R}n_{\mathcal{LT}}\ le\ n) \\
\forall\, a : A;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{LT}}(at\ a)\ n = at(\mathcal{R}n_{\mathcal{A}}\ a\ n) \\
\forall\, a : A;\ le : LT;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{LT}}(lat(a, le))\ n = lat((\mathcal{R}n_{\mathcal{A}}\ a\ n), (\mathcal{R}n_{\mathcal{LT}}\ le\ n))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{R}n_{\mathcal{CL}} : CL \to \mathbb{N} \to CL \\
\mathcal{R}n_{\mathcal{C}} : C \to \mathbb{N} \to C \\
\mathcal{R}n_{\mathcal{E}} : E \to \mathbb{N} \to E \\
\hline
\forall\, e_1, e_2 : E;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{CL}}(chs(e_1, e_2))\ n = chs(\mathcal{R}n_{\mathcal{E}}\ e_1\ n, \mathcal{R}n_{\mathcal{E}}\ e_2\ n) \\
\forall\, e : E;\ cl : CL;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{CL}}(lchs(e, cl))\ n = lchs(\mathcal{R}n_{\mathcal{E}}\ e\ n, \mathcal{R}n_{\mathcal{CL}}\ cl\ n) \\
\forall\, e : E;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{C}}(och\ e)\ n = och(\mathcal{R}n_{\mathcal{E}}\ e\ n) \\
\forall\, cl : CL;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{C}}(mch\ cl)\ n = mch(\mathcal{R}n_{\mathcal{CL}}\ cl\ n) \\
\forall\, a : A;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{E}}(eat\ a)\ n = eat(\mathcal{R}n_{\mathcal{A}}\ a\ n) \\
\forall\, c : C;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{E}}(ech\ c)\ n = ech(\mathcal{R}n_{\mathcal{C}}\ c\ n) \\
\forall\, a : A;\ e : E;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{E}}(eats(a, e))\ n = eats(\mathcal{R}n_{\mathcal{A}}\ a\ n, \mathcal{R}n_{\mathcal{E}}\ e\ n) \\
\forall\, c : C;\ e : E;\ n : \mathbb{N} \bullet \mathcal{R}n_{\mathcal{E}}(echs(c, e))n = echs(\mathcal{R}n_{\mathcal{C}}\ c\ n, \mathcal{R}n_{\mathcal{E}}\ e\ n)
\end{array}
$$

Variable Dependency.

$$
\begin{array}{l}
\mathcal{DV}_{\mathcal{L}} : L \to \mathbb{P}\, I \to \mathbb{P}\, I \\
\hline
\forall\, lt : LT;\ sep, ei : Str;\ is : \mathbb{P}\, I \bullet \\
\qquad \mathcal{DV}_{\mathcal{L}}(list(lt, sep, ei))is = \ \mathsf{if}\ \neg\ \mathcal{V}_{\mathcal{LT}}\ lt \cap is = \{\}\ \mathsf{then}\ \mathcal{V}_{\mathcal{LT}}\ lt \cup is\ \mathsf{else}\ is
\end{array}
$$

$$
\begin{array}{l}
\mathcal{DV}_{\mathcal{A}} : A \to \mathbb{P}\, I \to \mathbb{P}\, I \\
\hline
\forall\, i : I;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{A}}(param\ i)is = is \\
\forall\, t : Str;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{A}}(tx\ t)is = is \\
\forall\, l : L;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{A}}(ls\ l)is = \mathcal{DV}_{\mathcal{L}}\ l\ is
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{DV}_{\mathcal{E}} : E \to \mathbb{P}\, I \to \mathbb{P}\, I \\
\mathcal{DV}_{\mathcal{C}} : C \to \mathbb{P}\, I \to \mathbb{P}\, I \\
\mathcal{DV}_{\mathcal{CL}} : CL \to \mathbb{P}\, I \to \mathbb{P}\, I \\
\hline
\forall\, a : A;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{E}}(eat\ a)is = \mathcal{DV}_{\mathcal{A}}\ a\ is \\
\forall\, c : C;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{E}}(ech\ c)is = \mathcal{DV}_{\mathcal{C}}\ c\ is \\
\forall\, a : A;\ e : E;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{E}}(eats(a, e))is = \mathcal{DV}_{\mathcal{E}}\ e(\mathcal{DV}_{\mathcal{A}}\ a\ is) \\
\forall\, c : C;\ e : E;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{E}}(echs(c, e))is = \mathcal{DV}_{\mathcal{E}}\ e(\mathcal{DV}_{\mathcal{C}}\ c\ is) \\
\forall\, e : E;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{C}}(och\ e)is = \mathcal{DV}_{\mathcal{E}}\ e\ is \\
\forall\, cl : CL;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{C}}(mch\ cl)is = \mathcal{DV}_{\mathcal{CL}}\ cl\ is \\
\forall\, e_1, e_2 : E;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{CL}}(chs(e_1, e_2))is = \mathcal{DV}_{\mathcal{E}}\ e_1\ is \cup \mathcal{DV}_{\mathcal{E}}\ e_2\ is \\
\forall\, e : E;\ cl : CL;\ is : \mathbb{P}\, I \bullet \mathcal{DV}_{\mathcal{CL}}(lchs(e, cl))is = \mathcal{DV}_{\mathcal{E}}\ e\ is \cup \mathcal{DV}_{\mathcal{CL}}\ cl\ is
\end{array}
$$

Instantiation Functions

$$
\begin{array}{|l}
\mathcal{LT}_2\mathcal{E} : LT \to E \\
\hline
\forall\, a : A \bullet \mathcal{LT}_2\mathcal{E}(at\ a) = eat\ a \\
\forall\, a : A;\ le : LT \bullet \mathcal{LT}_2\mathcal{E}(lat(a, le)) = eats(a, (\mathcal{LT}_2\mathcal{E}\ le))
\end{array}
$$

$$LEnv == \mathbb{P}\, I \times \operatorname{seq} Env$$

$$
\begin{array}{|l}
LEnvC : \mathbb{P}\, LEnv \\
\hline
\forall\, is : \mathbb{P}\, I \bullet (is, \langle\rangle) \in LEnvC \\
\forall\, is : \mathbb{P}\, I;\ e : Env \bullet (is, \langle e\rangle) \in LEnv \Leftrightarrow \operatorname{dom} e = is \\
\forall\, is : \mathbb{P}\, I;\ e : Env;\ se : \operatorname{seq} Env \bullet \\
\qquad (is, \langle e\rangle \frown se) \in LEnvC \Leftrightarrow \operatorname{dom} e = is \wedge (is, se) \in LEnvC
\end{array}
$$

$$
\begin{array}{|l}
\mathcal{IL}_{\mathcal{L}} : L \to LEnv \to E \\
\mathcal{IL}_{\mathcal{LC}} : L \to LEnv \times \mathbb{N}_1 \to E \\
\hline
\forall\, l : L;\ lenv : LEnv \bullet \mathcal{IL}_{\mathcal{L}}\ l\ lenv = \mathcal{IL}_{\mathcal{LC}}\ l(lenv, 1) \\
\forall\, lt : LT;\ sep, ei : Str;\ is : \mathbb{P}\, I;\ n : \mathbb{N} \bullet \\
\qquad \mathcal{IL}_{\mathcal{LC}}(list(lt, sep, ei))((is, \langle\rangle), n) = \\
\qquad\qquad \textsf{if } \mathcal{V}_{\mathcal{LT}}\ lt \cap is \neq \varnothing \textsf{ then } eat(tx\ ei) \\
\qquad\qquad \textsf{else } eat(ls(list(lt, sep, ei))) \\
\forall\, lt : LT;\ sep, ei : Str;\ is : \mathbb{P}\, I;\ env : Env;\ rlenv : \operatorname{seq} Env;\ n : \mathbb{N} \bullet \\
\qquad \mathcal{IL}_{\mathcal{LC}}(list(lt, sep, ei))((is, \langle env\rangle \frown rlenv), n) = \\
\qquad\qquad \textsf{if } \mathcal{V}_{\mathcal{LT}} lt \cap is \neq \varnothing \textsf{ then} \\
\qquad\qquad\qquad \mathcal{R}n_{\mathcal{E}}(\mathcal{IP}_{\mathcal{E}}(\mathcal{LT}_2\mathcal{E}\ lt)env)\ n \\
\qquad\qquad\qquad\qquad \Cup_E\ \mathcal{IL}_{\mathcal{LC}}(list((lat((tx\ sep), lt)), \Lambda, \Lambda))((is, rlenv), n + 1) \\
\qquad\qquad \textsf{else } eat(ls(list(lt, sep, ei)))
\end{array}
$$

$$\mathcal{IL}_{\mathcal{A}} : A \to LEnv \to E$$

$$\forall\, i : I;\ lenv : LEnv \bullet \mathcal{IL}_{\mathcal{A}}(param\ i)lenv = eat(param\ i)$$
$$\forall\, t : Str;\ lenv : LEnv \bullet \mathcal{IL}_{\mathcal{A}}(tx\ t)lenv = eat(tx\ t)$$
$$\forall\, l : L;\ lenv : LEnv \bullet \mathcal{IL}_{\mathcal{A}}(ls\ l)lenv = \mathcal{IL}_{\mathcal{L}}\ l\ lenv$$

$$\mathcal{IL}_{\mathcal{E}0} : E \to LEnv \to E$$
$$\mathcal{IL}_{\mathcal{C}} : C \to LEnv \to C$$
$$\mathcal{IL}_{\mathcal{CL}} : CL \to LEnv \to CL$$

$$\forall\, a : A;\ lenv : LEnv \bullet \mathcal{IL}_{\mathcal{E}0}(eat\ a)lenv = \mathcal{IL}_{\mathcal{A}}\ a\ lenv$$
$$\forall\, c : C;\ lenv : LEnv \bullet \mathcal{IL}_{\mathcal{E}0}(ech\ c)lenv = ech(\mathcal{IL}_{\mathcal{C}}\ c\ lenv)$$
$$\forall\, a : A;\ e : E;\ lenv : LEnv \bullet$$
$$\quad \mathcal{IL}_{\mathcal{E}0}(eats(a, e))lenv = (\mathcal{IL}_{\mathcal{A}}\ a\ lenv) +\!\!+_E\ (\mathcal{IL}_{\mathcal{E}0}\ e\ lenv)$$
$$\forall\, c : C;\ e : E;\ lenv : LEnv \bullet$$
$$\quad \mathcal{IL}_{\mathcal{E}0}(echs(c, e))lenv = echs((\mathcal{IL}_{\mathcal{C}}\ c\ lenv), (\mathcal{IL}_{\mathcal{E}0}\ e\ lenv))$$
$$\forall\, e : E;\ lenv : LEnv \bullet$$
$$\quad \mathcal{IL}_{\mathcal{C}}(och\ e)lenv = och(\mathcal{IL}_{\mathcal{E}0}\ e\ lenv)$$
$$\forall\, cl : CL;\ lenv : LEnv \bullet \mathcal{IL}_{\mathcal{C}}(mch\ cl)lenv = mch(\mathcal{IL}_{\mathcal{CL}}\ cl\ lenv)$$
$$\forall\, e_1, e_2 : E;\ lenv : LEnv \bullet$$
$$\quad \mathcal{IL}_{\mathcal{CL}}(chs(e_1, e_2))lenv = chs((\mathcal{IL}_{\mathcal{E}0}\ e_1\ lenv), (\mathcal{IL}_{\mathcal{E}0}\ e_2\ lenv))$$
$$\forall\, e : E;\ cl : CL;\ lenv : LEnv \bullet$$
$$\quad \mathcal{IL}_{\mathcal{CL}}(lchs(e, cl))lenv = lchs((\mathcal{IL}_{\mathcal{E}0}\ e\ lenv), (\mathcal{IL}_{\mathcal{CL}}\ cl\ lenv))$$

$$\mathcal{IL}_{\mathcal{E}} : E \to LEnvC \to E$$

$$\forall\, e : E;\ is : \mathbb{P}\, I;\ se : \mathrm{seq}\, Env \bullet \mathcal{IL}_{\mathcal{E}}\ e\ (is, se) = \mathcal{IL}_{\mathcal{E}0}\ e\ ((\mathcal{DV}_{\mathcal{E}}\ e\ is), se)$$

Choice

$$\mathcal{IC}_{\mathcal{E}} : E \to \mathrm{seq}\, \mathbb{N} \nrightarrow E$$

$$\forall\, a : A \bullet \mathcal{IC}_{\mathcal{E}}(eat\ a)\langle\rangle = eat\ a$$
$$\forall\, c : C;\ n : \mathbb{N} \bullet \mathcal{IC}_{\mathcal{E}}(ech\ c)\langle n\rangle = \mathcal{M}_{\mathcal{C}}\ c\ n$$
$$\forall\, a : A;\ e : E;\ sn : \mathrm{seq}\, \mathbb{N} \bullet \mathcal{IC}_{\mathcal{E}}(eats(a, e))sn = eats(a, (\mathcal{IC}_{\mathcal{E}}\ e\ sn))$$
$$\forall\, c : C;\ e : E;\ n : \mathbb{N};\ sn : \mathrm{seq}\, \mathbb{N} \bullet \mathcal{IC}_{\mathcal{E}}(echs(c, e))(\langle n\rangle \frown sn) = (\mathcal{M}_{\mathcal{C}}\ c\ n) +\!\!+_E\ (\mathcal{IC}_{\mathcal{E}}\ e\ sn)$$

B
Inference rules

This appendix presents a collection of inference rules for proof with Z and template-Z. Only the rules of template-Z are new, and so they are proved in this document (appendix G, section G.2). The other rules are taken from [WD96, VST04, Abr96], where their proofs can be found.

This appendix starts by briefly explaining formal proof and inference rules. Then, it presents inference rules for sequent calculus, propositional calculus, predicate calculus, Z schemas and template-Z.

B.1 Formal Proof, sequents, conjectures and inference rules

In logic, a proof is an argument that demonstrates the truth of an assertion, called a conclusion, based on the truth of a set of assertions, called premises. A proof is made formal by rendering it within a formal system, which comprises a set of axioms that are used to logically derive theorems. A formal proof consists of transforming a formal structure, here *sequents* of Gentzen systems are used (natural deduction systems uses *derivations trees*), by appeal to *inference rules* [TS00, Abr96]. A sequent has the form:

$$\Gamma \vdash P$$

Γ stands for the hypothesis (also called premises or assumptions), which one may assume in the course of proving P, and P is the goal. The collection of hypothesis, Γ, and the goal, P, are predicates. (For example, the proof of section 3.1.1 is done by transforming a sequent.)

Inference rules have the form:

$$\frac{P_1,\ P_2,\ \ldots\ ,\ P_n}{C}\ \begin{array}{l}[\text{rulename}]\\ [\textit{proviso}]\end{array}$$

$P1, \ldots, Pn$ are the antecedents (or premises) of the rule, and C is the consequent (or conclusion). This says that C can be derived from the antecedents $P_1, \ldots, P_n$ provided that the condition stated in the proviso is true. Provisos may refer to the free variables of a formula

(briefly, a variable is free if it is not bound by a quantifier). In the following, the function FV, which gives the set of free variables of a predicate calculus formula, is assumed.

Z conjectures have the form $\vdash?\ \ P$, where P is a well-formed Z predicate, which is said to be proved under the statements of the specification. The truth of the conjecture $\vdash?\ \ P$ is demonstrated by proving the sequent $\Gamma\ \vdash\ P$, where Γ may include statements of the Z specification; if the sequent is proved then the conjecture establishes a theorem of the specification.

B.2 Sequent Calculus

$$\frac{\Gamma;\ P\ \vdash\ P}{true}\ [\text{hyp}] \qquad \frac{\Gamma;\ Q\ \vdash\ P}{\Gamma\ \vdash\ P}\ [\text{thin}]$$

$$\frac{\Gamma\ \vdash\ Q}{\Gamma\ \vdash\ P \quad \Gamma;\ P\ \vdash\ Q}\ [\text{add hyp}]$$

B.3 Propositional Calculus

$$\frac{\Gamma\ \vdash\ Q \Rightarrow P}{\Gamma;\ Q \vdash\ P}\ [\Rightarrow\text{-hyp}] \qquad \frac{\Gamma\ \vdash\ P \wedge \text{true}}{\Gamma \vdash\ P}\ [\wedge\text{-Id}]$$

$$\frac{\Gamma\ \vdash\ P \wedge Q}{\Gamma\ \vdash\ P \quad \Gamma\ \vdash\ Q}\ [\text{conj}]$$

B.4 Predicate Calculus

$$\frac{}{\Gamma \vdash\ e = e}\ [\text{eq-ref}] \qquad \frac{\Gamma \vdash\ e = f \quad \Gamma \vdash\ P[x := f]}{\Gamma \vdash\ P[x := e]}\ [\text{eq-sub}]$$

$$\frac{\Gamma\ \vdash\ \forall\ x\ :\ A\ \bullet P}{\Gamma;\ x\ \in\ A\ \vdash\ P}\ [\forall\text{-I}]\ [x\ \notin FV(\Gamma)] \qquad \frac{\Gamma\ \vdash\ \exists\ x : A \bullet P \wedge x = v}{\Gamma\ \vdash\ P[\ x\ :=\ v\ ] \wedge v \in A}\ [\text{one-point}]\ [x\ \notin\ FV(v)]$$

B.5 Z Schema Calculus

$$\frac{\Gamma\ \vdash \exists[\ ScD \mid ScP\ ] \bullet P}{\Gamma\ \vdash \exists\, ScD \bullet P \wedge ScP}\ [\exists\ Sc] \qquad \frac{\Gamma\ \vdash \exists\, ScA \bullet [\ ScA;\ D \mid ScP\ ]}{\Gamma\ \vdash [D \mid \exists\, ScA \bullet ScP\ ]}\ [\exists\ Sc\text{-2}]$$

$$\frac{\Gamma\ \vdash \exists\ S \bullet \theta S = b \wedge P}{\Gamma\ \vdash P[\theta S := b] \wedge b \in S}\ [\theta\text{-point}]\ [Ss\ \notin\ FV(P)]$$

B.6 Template-Z inference rules

B.6.1 Axiom rules

$$\frac{\Gamma \vdash E_1 \ll N \gg E_2}{(\mathcal{IP}_{\mathcal{E}}(\Gamma \vdash E_1 \ll N \gg E_2)\{N \mapsto \text{“}N\text{”}\})} \quad \begin{array}{l}[\text{T-I-PN}] \\ [N \notin FV(\Gamma)]\end{array}$$

$$\frac{\Gamma \vdash E_1 \ll P \gg E_2}{(\mathcal{IP}_{\mathcal{E}}(\Gamma \vdash E_1 \ll P \gg E_2)\{P \mapsto \text{“}P\text{”}\})} \quad [\text{T-I-PP}]$$

$$\frac{\Gamma \vdash E_1 \ll S \gg E_2}{S \in \mathbb{U};\ (\mathcal{IP}_{\mathcal{E}}(\Gamma \vdash E_1 \ll S \gg E_2)\{S \mapsto \text{“}S\text{”}\})} \quad \begin{array}{l}[\text{T-I-PS}] \\ [S \notin FV(\Gamma)]\end{array}$$

$$\frac{\Gamma \vdash E_1 [\![\ LE_1 \ll x \gg LE_2\ ]\!]\ E_2}{\begin{array}{l}\mathcal{IL}_{\mathcal{E}}(\Gamma \vdash E_1 [\![\ LE_1 \ll x \gg LE_2\ ]\!]\ E_2)(\{x\}, \langle\rangle) \\ \quad \land\ \mathcal{IL}_{\mathcal{E}}(\Gamma \vdash E_1 [\![\ LE_1 \ll x \gg LE_2\ ]\!]\ E_2)(\{x\}, se) \\ \qquad \Rightarrow \mathcal{IL}_{\mathcal{E}}(\Gamma \vdash E_1 [\![\ LE_1 \ll x \gg LE_2\ ]\!]\ E_2)(\{x\}, se \frown \langle\{x \mapsto \text{“}x\text{”}\}\rangle)\end{array}} \quad [\ \mathsf{T\text{-}I\text{-}L}\ ]$$

B.6.2 Derived Rules

$$\frac{\Gamma \vdash\ \exists \ll x \gg : \ll t \gg \bullet\ \ll P \gg\ \land\ \ll x \gg = \ll v \gg}{\Gamma \vdash\ \ll P \gg [\ \ll x \gg := \ll v \gg\ ]\ \land\ \ll v \gg \in \ll t \gg} \quad \begin{array}{l}[\text{T-one-point}] \\ [\ll x \gg\ \notin\ FV(\ll v \gg)]\end{array}$$

$$\frac{\Gamma \vdash\ \exists [\![\ \ll x \gg : \ll t \gg\ ]\!] \bullet\ \ll P \gg [\![\ \land\ \ll x \gg = \ll v \gg\ ]\!]}{\Gamma \vdash\ \ll P \gg [\ [\![\ \ll x \gg := \ll v \gg\ ]\!]\ ] [\![\ \land\ \ll v \gg \in \ll t \gg\ ]\!]} \quad \begin{array}{l}[\text{T-ls-one-point}] \\ [[\![\ \ll x \gg\ \notin\ FV(\ll v \gg)\ ]\!]]\end{array}$$

$$\frac{\Gamma \vdash \exists\ [\ \ll ScD \gg \mid \ll ScP \gg\ ] \bullet \ll P \gg}{\Gamma \vdash \exists\ \ll ScD \gg \bullet \ll P \gg \land \ll ScP \gg} \quad [\text{T}\ \exists\ Sc]$$

$$\frac{\Gamma \vdash \exists \ll ScA \gg \bullet [\ \ll ScA \gg;\ D \mid \ll ScP \gg\ ]}{\Gamma \vdash [D \mid \exists \ll ScA \gg \bullet \ll ScP \gg\ ]} \quad [\text{T}\ \exists\ Sc\text{-2}]$$

$$\frac{\Gamma \vdash \exists\ \ll S \gg \bullet\, \theta \ll S \gg = \ll b \gg \land \ll P \gg}{\Gamma \vdash \ll P \gg [\theta S := b] \land \ll b \gg \in \ll S \gg} \quad \begin{array}{l}[\text{T-}\theta\text{-point}] \\ [\ll S \gg s\ \notin\ FV(\ll P \gg)]\end{array}$$

C

ZOO Generics toolkit

This appendix presents the generics toolkit of the ZOO style. This toolkit is used by the ZOO models generated from the $UML + Z$ templates catalogue.

The following presents the ZOO domain toolkit and other generics used in the ZOO models of this thesis.

C.1 ZOO Domain Toolkit

First, a separate Z section is defined for the ZOO toolkit, which uses the standard Z toolkit (parent). Every ZOO model uses the ZOO toolkit (as a *parent*):

$$\mathsf{section}\ \mathit{ZOO\_toolkit}\ \mathsf{parents}\ \mathit{standard\_toolkit}$$

There is a single Z type to represent all object atoms, the given set OBJ. Its definition asserts that this set is nonempty:

$$[OBJ]$$
$$OBJ \neq \varnothing$$

Generic G1 (Graphs) The following generics are related to graphs, defining acyclic graphs, direct acyclic graphs (DAGs) and trees:

$$\mathsf{Acyclic}[X] == \{rel : X \leftrightarrow X \mid rel^+ \cap \mathrm{id}\, X = \varnothing\}$$
$$\mathsf{Dag}[X] == \{rel : X \leftrightarrow X \mid rel \in \mathsf{Acyclic}\}$$
$$\mathsf{Tree}[X] == \{rel : X \leftrightarrow X \mid rel \in \mathsf{Dag} \land rel \in X \nrightarrow X\}$$

Laws. (proved in Z/Eves)

$$\vdash \varnothing \in \mathsf{Dag}[X]$$
$$\vdash \varnothing \in \mathsf{Tree}[X]$$

□

Generic G2 (Class extension) The class extension generic defines a set of object atoms and a function mapping objects to their states; the set of object is equal to the domain of the function:

$$
\begin{array}{|l}
\hline
\mathbb{S}\mathrm{CL}\,[OS, OST] \\
os : \mathbb{P}\ OS \\
oSt : OS \pfun OST \\
\hline
\mathrm{dom}\ oSt = os \\
\hline
\end{array}
$$

□

Generic G3 (Multiplicity constraints) The following is used to express association multiplicity constraints. The *MultTy* free type defines all possible combinations of association multiplicity constraints, such as many to many (mm), one-to-many (om) and so forth:

$$
\begin{aligned}
MultTy ::= {} & mm \mid mo \mid om \mid mzo \mid zom \mid oo \mid zozo \mid zoo \mid ozo \mid ms \mid sm \mid ss \\
& \mid so \mid os \mid szo \mid zos
\end{aligned}
$$

The generic is defined as a set composed of a relation (the association), two sets of objects (the extensions of the participating classes), one value of the set *MultTy* (the multiplicity constraint) and two set sets of natural numbers (this is to define cardinalities other than one, one or zero, and many). This defines the appropriate constraints upon the relation depending on the multiplicity constraint. For example, the many to many (mm) association constrains the relation to be a relation between the two sets of objects, the one to many (om) association constrains the relation to be total function from one sets of objects to another, and so forth.

$$
\begin{array}{|l}
\hline
[X, Y] \\
\mathsf{mult}\_ : \mathbb{P}((X \rel Y) \times \mathbb{P}\ X \times \mathbb{P}\ Y \times MultTy \times \mathbb{F}\ \mathbb{N} \times \mathbb{F}\ \mathbb{N}) \\
\hline
\forall\, r : X \rel Y;\ sx : \mathbb{P}\ X;\ sy : \mathbb{P}\ Y;\ s_1, s_2 : \mathbb{F}\ \mathbb{N} \bullet \\
\quad (\mathsf{mult}(r, sx, sy, mm, s_1, s_2)) \Leftrightarrow r \in sx \rel sy \\
\forall\, r : X \rel Y;\ sx : \mathbb{P}\ X;\ sy : \mathbb{P}\ Y;\ s_1, s_2 : \mathbb{F}\ \mathbb{N} \bullet \\
\quad (\mathsf{mult}(r, sx, sy, mo, s_1, s_2)) \Leftrightarrow r \in sx \tfun sy \\
\forall\, r : X \rel Y;\ sx : \mathbb{P}\ X;\ sy : \mathbb{P}\ Y;\ s_1, s_2 : \mathbb{F}\ \mathbb{N} \bullet \\
\quad (\mathsf{mult}(r, sx, sy, om, s_1, s_2)) \Leftrightarrow r^{\sim} \in sy \tfun sx \\
\forall\, r : X \rel Y;\ sx : \mathbb{P}\ X;\ sy : \mathbb{P}\ Y;\ s_1, s_2 : \mathbb{F}\ \mathbb{N} \bullet \\
\quad (\mathsf{mult}(r, sx, sy, mzo, s_1, s_2)) \Leftrightarrow r \in sx \pfun sy \\
\forall\, r : X \rel Y;\ sx : \mathbb{P}\ X;\ sy : \mathbb{P}\ Y;\ s_1, s_2 : \mathbb{F}\ \mathbb{N} \bullet \\
\quad (\mathsf{mult}(r, sx, sy, zom, s_1, s_2)) \Leftrightarrow r^{\sim} \in sy \pfun sx \\
\hline
\end{array}
$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, oo, s_1, s_2)) \Leftrightarrow r \in sx \rightarrowtail\!\!\!\!\!\rightarrow sy$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, zozo, s_1, s_2)) \Leftrightarrow r \in sx \rightarrowtail\!\!\!\!\!\!\mapsto sy$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, zoo, s_1, s_2)) \Leftrightarrow r \in sx \rightarrowtail sy$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, ozo, s_1, s_2)) \Leftrightarrow r^{\sim} \in sy \rightarrowtail sx$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, ms, s_1, s_2)) \Leftrightarrow (\mathsf{mult}(r, sx, sy, mm, s_1, s_2)) \land (\forall x : \mathrm{dom}\, r \bullet \#(\{x\} \lhd r) \in s_1)$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, sm, s_1, s_2)) \Leftrightarrow (\mathsf{mult}(r, sx, sy, mm, s_1, s_2)) \land (\forall y : \mathrm{ran}\, r \bullet \#(r \rhd \{y\}) \in s_1)$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, ss, s_1, s_2)) \Leftrightarrow (\mathsf{mult}(r, sx, sy, ms, s_1, \{\})) \land (\mathsf{mult}(r, sx, sy, sm, s_2, \{\}))$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, so, s_1, s_2)) \Leftrightarrow (\mathsf{mult}(r, sx, sy, mo, s_1, s_2)) \land (\mathsf{mult}(r, sx, sy, sm, s_1, s_2))$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, os, s_1, s_2)) \Leftrightarrow (\mathsf{mult}(r, sx, sy, om, \{\}, \{\})) \land (\mathsf{mult}(r, sx, sy, ms, s_1, \{\}))$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, szo, s_1, s_2)) \Leftrightarrow (\mathsf{mult}(r, sx, sy, mzo, \{\}, \{\})) \land (\mathsf{mult}(r, sx, sy, sm, s_1, \{\}))$$

$$\forall r : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ s_1, s_2 : \mathbb{F}\, \mathbb{N} \bullet$$
$$(\mathsf{mult}(r, sx, sy, zos, s_1, s_2)) \Leftrightarrow (\mathsf{mult}(r, sx, sy, zom, \{\}, \{\})) \land (\mathsf{mult}(r, sx, sy, ms, s_1, \{\}))$$

Laws. (proved in Z/Eves) The first law says that an empty relation, and empty sets (the pair) satisfies any constraint. The second says that if a relation and a pair of sets satisfy the one-many constraint, then so does the same relation, with an object added to the first set, and the same second set.

$$\vdash \forall m : \mathit{MultTy};\ r_1, r_2 : \mathbb{F}\, \mathbb{N} \bullet \mathsf{mult}(\varnothing, \varnothing, \varnothing, m, r_1, r_2)$$

$$\vdash [X, Y]\, \forall \mathit{rel} : X \leftrightarrow Y;\ sx : \mathbb{P}\, X;\ sy : \mathbb{P}\, Y;\ x : X;\ r_1, r_2 : \mathbb{F}\, \mathbb{N} \mid \mathsf{mult}(\mathit{rel}, sx, sy, om, r_1, r_2) \bullet \mathsf{mult}(\mathit{rel}, \{x\} \cup sx, sy, om, r_1, r_2)$$

□

C.2 Other Generics

Generic G4 (Sum over a finite labelled set) The following generic defines a sum over a finite set of indexed integers. This is defined recursively.

$[L]$

$$\Sigma : (L \pfun \mathbb{Z}) \to \mathbb{Z}$$

$$\Sigma\ \varnothing = 0$$
$$\forall\, l : L;\ n : \mathbb{Z};\ S : L \pfun \mathbb{Z} \mid l \notin \operatorname{dom} S \bullet \Sigma\ (\{l \mapsto n\} \cup S) = n + \Sigma\ S$$

Laws. (proved in Z/Eves)

The following gives a law for a labelled set overridden with a new tuple.

$$\vdash \forall\, s_1 : (L \pfun \mathbb{Z});\ l : L;\ n : \mathbb{Z} \bullet \Sigma(s_1 \oplus \{l \mapsto n\}) = \Sigma(\{l\} \ndres s_1) + n$$

□

D

UML + Z templates catalogue

This appendix defines the catalogue of FTL templates and meta-theorems of the *UML + Z* framework. The template definitions use generics from the ZOO toolkit, which are defined in appendix C. The templates capture typical structures of ZOO models, so that every sentence of a ZOO model is generated by instantiating one of these templates. The meta-proofs of the template's meta-theorems are given in appendix G (section G.3).

The names of meta-theorems are formed with suffixes that refer to the template being used and to the meta-theorem itself. The following suffixes are used:

- cl: class (when alone refers to an intensional definition).
- scl: subclass (when alone refers to an intensional definition).
- ext: indicates the class or subclass definition is extensional.
- assoc: association.
- sys: system.
- init: initialisation.
- pre: pre-condition.
- uop: update operation.
- nop: new operation.
- oop: observe operation.
- dop: delete operation.
- d: distributes.

The following presents the templates of the catalogue for each view of the ZOO style.

D.1 Structural View

The structural view defines global names that are used in the remaining views. These names introduce the set of class atoms of a ZOO model and the set of all possible object atoms of each class.

Template T1 (Model section and set of all classes)

Every ZOO model needs to define a separate section, which has as parent the ZOO toolkit:

$$\mathsf{section}\ \lessdot mdl \gtrdot\_model\ \mathsf{parents}\ ZOO\_toolkit$$

The set $CLASS$ (defined as a Z free-type) defines the set of all classes of a model as atoms. This template is defined as an FTL list of terms composed of a placeholder (a name) appended with 'Cl' and separated by '|':

$$CLASS ::= [\![\lessdot Cl \gtrdot Cl\ ]\!]_{(|,"")}$$

The following defines the relation $subCl$, which captures all subclass relationships of a model, and the set $abstractCl$, which captures all abstract classes of a model. $subCl$ is a relation between class atoms ($CLASS$ above) that denotes a set of tuples, each representing a mapping from the subclass to the superclass (captured as an FTL list). The set $abstractCl$ includes all classes that are abstract.

$$\begin{array}{|l} subCl : CLASS \leftrightarrow CLASS \\ abstractCl : \mathbb{P}\ CLASS \\ rootCl : \mathbb{P}\ CLASS \\ \hline subCl = \{[\![\lessdot ChCl \gtrdot Cl \mapsto \lessdot PCl \gtrdot Cl\ ]\!]\}_{(",",\ ,"")} \\ abstractCl = \{[\![\lessdot ACl \gtrdot\ ]\!] Cl\}_{(",",\ ,"")} \\ rootCl = CLASS \setminus \mathrm{dom}\ subCl \end{array}$$

If there are no subclassings in the model then the subclass relation is empty. Likewise, for the set $abstractCl$.

The user needs to check that the inheritance hierarchy is well-formed. First, the hierarchy must not have cycles; so, the template conjecture requires that the subclass relation is a Tree (single-inheritance).[1] Second, every abstract class of a model must have at least one descendant.

$$\vdash ?\ subCl \in \mathsf{Tree}$$
$$\vdash ?\ abstractCl = abstractCl \cap \mathrm{ran}\ subCl$$

where Tree is a generic from the ZOO toolkit.

□

[1] Multiple inheritance requires that the class hierarchy forms a DAG.

Template T2 (Set of objects of a class)

Each class has a set of object atoms. This is defined by functions $\mathbb{O}$ and $\mathbb{O}_x$, which both take a class atom and return a non-empty subset of OBJ. The first gives the set of all possible objects of a class, including those that are instances of its descendants. The second gives only the direct objects of the class, and so excludes those of its descendants. Using this function, the set of objects of a class, say $Person$, can be obtained as $\mathbb{O}\ PersonCl$.

$$\begin{array}{|l} \mathbb{O} : CLASS \rightarrow \mathbb{P}_1\ OBJ \\ \mathbb{O}_x : CLASS \rightarrow \mathbb{P}_1\ OBJ \\ \hline \text{disjoint}\ \mathbb{O}_x \\ \forall\, cl : CLASS \bullet \mathbb{O}\ cl = \mathbb{O}_x\ cl \cup \bigcup(\mathbb{O}_x(\!|\ (subCl^+)^{\sim}(\!|\ \{cl\}\ |\!)\ |\!)) \\ \forall\, cl, cl' : CLASS \mid cl \mapsto cl' \in subCl \bullet \mathbb{O}\ cl \subseteq \mathbb{O}\ cl' \end{array}$$

The constraints are as follows. The first says that the exclusive sets of objects of each class are mutually disjoint. The second says that all possible objects of a class ($\mathbb{O}$) is the union of the class's exclusive set ($\mathbb{O}_x$) with all the exclusive sets of the class's descendants (a generalised union). Finally, the third constraint says that the set of all possible objects of a subclass is a subset of its superclass.
□

D.2 Class View

D.2.1 Intensional View

The intensional view templates support the construction of intensional definitions of a class. This comprises class attributes, state space, initialisation, operations, and finalisation. The templates are defined below.

Template T3 (Attribute Types)

The types of class attributes may be defined as a non-empty Z given set or as a Z free type. So, the template comprises an optional choice of a list of non-empty given sets, and a list of free types. The assertion that each given set is non-empty is important for existence proofs.

$$\begin{array}{l} (\!| \\ [\ [\![\ll tyAg \gg]\!]\ ] \\ \hline [\![\ll tyAg \gg \neq \varnothing]\!] \\ |\!)^? \\ [\![\ll tyAe \gg ::= \ll val \gg [\![\mid \ll val \gg]\!]]\!] \end{array}$$

□

Template T4 (Free Axiomatic Description)

In some occasions, a definition of a free Z axiomatic description is required. The template requires a name for the new definition, a type and a constraint:

$$\begin{array}{|l} \ll DN \gg : \ll DT \gg \\ \hline \ll DC \gg \end{array}$$

□

Template T5 (Class intensional state space and initialisation)

The state space defines class attributes, and expresses an invariant. The initialisation makes an assignment of values to state attributes and expresses an initialisation condition:

$$\begin{array}{|l} \ll Cl \gg \\ \hline [\![\ll at \gg : \ll atT \gg]\!] \\ \hline \ll CLI \gg \end{array} \qquad \begin{array}{|l} \ll Cl \gg Init \\ \hline \ll Cl \gg ' \\ \ll Cl \gg Init_I \\ \hline \ll pI \gg \\ [\![\ll at \gg' = \ll iv \gg]\!] \end{array}$$

$$\begin{array}{|l} \ll Cl \gg Init_I \\ \hline [\![\ll ii \gg ? : \ll iiT \gg]\!] \end{array}$$

There are two proof obligations. The first demonstrates that the initialisation is *well-formed*, that is, attributes are assigned valid values according to their types. The second is the usual initialisation conjecture.

$$\vdash ?\ \forall \ll Cl \gg Init_I \bullet [\![\ll iv \gg \in \ll atT \gg]\!]$$
$$\vdash ?\ \exists \ll Cl \gg Init \bullet \text{true}$$

Meta-Theorems. The meta-theorems simplify the initialisation conjecture by relying on the proof of the well-formedness conjecture. The first meta-theorem is more general: the existence of the initial state reduces to a proof that the class invariant is satisfied in the initial state. The second conjecture gives *true by construction*: if there is no invariant and no extra initialisation condition then there is an initial state.

$$\frac{\Gamma \vdash \exists \ll Cl \gg Init \bullet \text{true}}{\Gamma \vdash \exists \ll Cl \gg Init_I \bullet \ll CLI \gg' [\![\ll at \gg' := \ll iv \gg]\!] \\ \quad \wedge \ll pI \gg [\![\ll at \gg' := \ll iv \gg]\!]} \text{ [cl-init]}$$

$$\frac{\Gamma \vdash \ll CLI \gg \wedge \ll pI \gg, \Gamma \vdash \exists \ll Cl \gg Init \bullet \text{true}}{\text{true}} \begin{array}{l} \text{[cl-init-ni]} \\ [\![\ll iiT \gg \neq \varnothing]\!] \end{array}$$

□

Template T6 (Class intensional update operation)

An update operation changes the state of a class object. The operation schema, includes in the declarations the delta of the state space and a list of inputs; in the predicate, there is an operation precondition and a list of assignments to the after state attributes.[2]

$$
\begin{array}{|l}
\hline
\ll Cl\gg_{\Delta}\ll Op\gg \\
\Delta \ll Cl\gg \\
[\![\ll io\gg? : \ll ioT\gg]\!] \\
\hline
\ll pOp\gg \\
[\![\ll at\gg' = \ll nv\gg]\!] \\
\hline
\end{array}
$$

There are two conjectures associated with this template. The first demonstrates that the operation is *well-formed*; that is, the after-state attributes are given valid values. The second is the usual precondition consistency conjecture.

$$\vdash? \ \forall \Delta \ll Cl\gg; [\![\ll io\gg? : \ll ioT\gg]\!] \bullet [\![\ll nv\gg \in \ll atT\gg]\!]$$

$$\vdash? \ \exists \ \text{pre}\ll Cl\gg_{\Delta}\ll Op\gg \bullet \text{true}$$

Meta-Theorems. The following meta-theorems simplify the operation's precondition and consistency conjecture by relying on the proof of the well-formedness conjecture. The first reduces the precondition to a substitution on the class invariant and operation extra precondition. The second gives *true by construction* on the precondition consistency conjecture, provided there is no class invariant and no extra operation's precondition.

$$\frac{\text{pre}\ll Cl\gg_{\Delta}\ll Op\gg}{\begin{array}{l}[\ \ll Cl\gg; [\![\ll io\gg? : \ll ioT\gg]\!] \mid \ll CLI\gg' [\![\ll at\gg' := \ll nv\gg]\!] \ ] \\ \quad \wedge \ll pOp\gg [\ [\![\ll at\gg' := \ll nv\gg]\!] \ ] \ ]\end{array}} \quad [\text{cl-uop-pre}]$$

$$\frac{\begin{array}{l}\Gamma \vdash \ll CLI\gg \wedge \ll pOp\gg \\ \quad \Gamma \vdash \exists \ \text{pre}\ \ll Cl\gg_{\Delta}\ll Op\gg \bullet \text{true}\end{array}}{\text{true}} \quad \begin{array}{l}[\text{cl-uop-epre-np}] \\ [[\![\ll atT\gg \neq \varnothing]\!] \wedge [\![\ll ioT\gg \neq \varnothing]\!]]\end{array}$$

Meta-lemmas. The following give useful distribution laws.

$$\frac{\Gamma \vdash \exists \ll Cl\gg' \bullet \ll P\gg \wedge \ll Cl\gg_{\Delta}\ll Op\gg}{\Gamma \vdash P[\ [\![\ll at\gg' := \ll nv\gg]\!] \ ] \wedge \text{pre}\ll Cl\gg_{\Delta}\ll Op\gg} \quad [\text{cl-uop-d-post}]$$

$$\frac{\text{pre}(\ll Cl\gg_{\Delta}\ll Op\gg \wedge \ll P\gg)}{\begin{array}{l}[\ \ll Cl\gg; [\![\ll io\gg? : \ll ioT\gg]\!] \mid \ll CLI\gg' [\![\ll at\gg' := \ll nv\gg]\!] \ ] \\ \quad \wedge \ll pOp\gg [\ [\![\ll at\gg' := \ll nv\gg]\!] \ ] \ ] \\ \quad \wedge \text{pre}\ll P\gg [\ [\![\ll at\gg' := \ll nv\gg]\!] \ ]\end{array}} \quad [\text{cl-uop-d-pre}]$$

□

[2] This describes a deterministic operation; a non-deterministic operation would require another template.

Template T7 (Class intensional observe operation)

These operations perform an observation upon the state of one object. The template describes this by disallowing changes of state (Ξ declaration), by allowing some condition to be set on the operation, and by giving some value to an output (the observation), which is built from the state of the object.

$\ll Cl \gg_{\Xi} \ll Op \gg$

$\Xi \ll Cl \gg$

$\ll out \gg ! : \ll oT \gg$

$\ll pOp \gg$

$\ll out \gg ! = \ll ov \gg$

There are two proof obligations associated with this template. The first demonstrates that the operation is *well-formed* The second is the precondition conjecture.

$\vdash ?\ \forall \Xi \ll Cl \gg;\ \ll out \gg ! : \ll oT \gg \bullet \ll ov \gg \in \ll oT \gg$

$\vdash ?\ \exists \, \text{pre} \ll Cl \gg_{\Xi} \ll Op \gg \bullet \text{true}$

Meta-Theorems. The first meta-theorem simplifies the operation's precondition. The second, gives *true by construction* on the precondition conjecture, provided the operation's condition is not present.

$$\frac{\text{pre}\ \ll Cl \gg_{\Xi} \ll Op \gg}{[\ \ll Cl \gg \mid \ll pOp \gg [\ \theta \ll Cl \gg' := \theta \ \ll Cl \gg, \ll out \gg ! := \ll ov \gg\ ]\ ]} \quad [\text{cl-oop-pre}]$$

$$\frac{\Gamma\ \vdash \ll pOp \gg,\ \Gamma\ \vdash \exists\ \text{pre}\ \ll Cl \gg_{\Xi} \ll Op \gg \bullet \text{true}}{\text{true}} \quad \begin{matrix}[\text{cl-oop-epre-np}] \\ [\![\ \ll atT \gg \neq \varnothing\ ]\!]\end{matrix}$$

□

Template T8 (Class intensional finalisation)

The finalisation describes the conditions under which an class object may be deleted. This is expressed as a predicate (variable $\ll fc \gg$).

$\ll Cl \gg Fin$

$\ll Cl \gg$

$[\![\ \ll if \gg ? : \ll ifT \gg\ ]\!]$

$\ll fc \gg$

The conjecture requires the existence of one state satisfying the finalisation condition.

$\vdash ?\ \exists\ \ll Cl \gg Fin \bullet \text{true}$

Meta-Theorems. The meta-theorems simplify the finalisation consistency conjectures. The first, more general, gives a simplification of the conjecture. The second gives a stronger result when the class intension does not have an invariant.

$$\frac{\Gamma \vdash \exists \ll Cl\gg Fin \bullet \text{true}}{\Gamma \vdash \exists [\![\ll at\gg : \ll atT\gg]\!]; [\![\ll if\gg ? : \ll ifT\gg]\!] \bullet \ll CLI\gg \land \ll fc\gg} \quad [\text{cl-fin}]$$

$$\frac{\Gamma \vdash \ll CLI\gg \land \ll fc\gg,\ \Gamma \vdash \exists \ll Cl\gg Fin \bullet \text{true}}{\text{true}} \quad [\text{cl-fin-ni}]$$
$$[\![\ll ifT\gg \neq \varnothing]\!] \land [\![\ll atT\gg \neq \varnothing]\!]$$

□

Template T9 (Class with statechart, intensional state space and initialisation)

This template defines the state space and initialisation for classes with simple statecharts.[3] The definition is divided in two views: the class as defined by the state diagram and the state of objects as defined by the statechart. These two views are then conjoined to make the class intensional definition.

The following defines the class intensional state and initialisation as coming from the class diagram.

$\ll Cl\gg_0$
$[\![\ll at\gg : \ll atT\gg]\!]$

$\ll CLI\gg$

$\ll Cl\gg Init_I$
$[\![\ll ii\gg ? : \ll iiT\gg]\!]$

$\ll Cl\gg Init_0$
$\ll Cl\gg '$
$\ll Cl\gg Init_I$

$\ll pI\gg$
$[\![\ll at\gg' = \ll iv\gg]\!]$

The current state as defined by the statechart requires a Z free type that includes all possible states of the statechart; this comprises an initial state and a list of other states.

$$\ll Cl\gg ST ::= \ll iSt\gg [\![\mid \ll oSt\gg]\!]$$

The following schema keeps the current state of the statechart. It includes the *st* attribute, which is initialised to the initial state of the statechart.

$\ll Cl\gg St$
$st : \ll Cl\gg ST$

$\ll Cl\gg StInit$
$\ll Cl\gg St'$

$st' = \ll iSt\gg$

The following conjoins the class state (as coming from the class diagram) and the statechart state to make the class intensional state space and initialisation.

$\ll Cl\gg$
$\ll Cl\gg_0$
$\ll Cl\gg St$

$\ll Cl\gg Init$
$\ll Cl\gg Init_0$
$\ll Cl\gg StInit$

[3] This allows flat states only; that is, it does not support neither nested nor concurrent states.

There are two proof obligations. The first demonstrates that the initialisation is *well-formed*, that is, attributes are assigned valid values according to their types. The second is the usual initialisation conjecture.

$$\vdash?\ \forall \ll Cl \gg Init_I \bullet [\![\ \ll iv \gg \in \ll atT \gg\ ]\!]$$
$$\vdash?\ \ \exists \ll Cl \gg Init \bullet \text{true}$$

Meta-Theorems. The meta-theorems simplify the initialisation conjecture by relying on the proof of the well-formedness conjecture. The first says that the initial state exists provided the invariant and initialisation condition are satisfied in the initial state. The second gives *true by construction*: if there is no invariant and no initialisation condition then there is an initial state.

$$\frac{\Gamma\ \vdash \exists\ \ll Cl \gg Init \bullet \text{true}}{\Gamma\ \vdash \exists\ \ll Cl \gg Init_I \bullet \ll CLI \gg' [\![\ \ll at \gg' := \ll iv \gg\ ]\!] \wedge \ll pI \gg [\![\ \ll at \gg' := \ll iv \gg\ ]\!]} \quad [\text{cl-stc-init}]$$

$$\frac{\Gamma\ \vdash \ll CLI \gg \wedge \ll pI \gg, \Gamma\ \vdash \exists\ \ll Cl \gg Init \bullet \text{true}}{\text{true}} \quad [\text{cl-stc-init-ni}]\ [\![\ \ll iiT \gg \neq \varnothing\ ]\!]$$

□

Template T10 (Class with statechart, intensional update operation)

An update operation describes a change of state of one object of a class. First, the state transitions of the operation as defined by the statechart are defined; they include a list of precondition/postcondition pairs as defined in the statechart (one for each transition).

$\ll Cl \gg_\Delta \ll Op \gg St$

$\Delta \ll Cl \gg St$

$st = \ll bSt \gg \wedge\ st' = \ll aSt \gg \vee [\![\ st = \ll bSt \gg \wedge\ st' = \ll aSt \gg\ ]\!]_{(\vee,\text{false})}$

The template operation schema includes the state transitions.[4]

$\ll Cl \gg_\Delta \ll Op \gg_0$

$\Delta \ll Cl \gg_0$

$[\![\ \ll io \gg? : \ll ioT \gg\ ]\!]$

$\ll pOp \gg$

$[\![\ \ll at \gg' = \ll nv \gg\ ]\!]$

$\ll Cl \gg_\Delta \ll Op \gg$

$\ll Cl \gg_\Delta \ll Op \gg_0$

$\ll Cl \gg_\Delta \ll Op \gg St$

There are two proof obligations associated with this template. The first demonstrates that the operation is *well-formed*; that is, the state attributes are given valid values. The second is the usual precondition consistency conjecture.

$$\vdash?\ \forall \Delta \ll Cl \gg;\ [\![\ \ll io \gg? : \ll ioT \gg\ ]\!] \bullet [\![\ \ll nv \gg \in \ll atT \gg\ ]\!]$$
$$\vdash?\ \exists\, \text{pre} \ll Cl \gg_\Delta \ll Op \gg \bullet \text{true}$$

[4] This describes a deterministic operation; a non-deterministic operation would require another template.

Meta-Theorems. The first meta-theorem simplifies the operation's precondition. The second gives *true by construction* on the precondition conjecture, provided there is no class invariant and no operation condition.

$$\frac{\text{pre}\ll Cl\gg_{\Delta}\ll Op\gg}{\begin{array}{l}[\ \ll Cl\gg;\ [\![\ \ll io\gg? : \ll ioT\gg\ ]\!]\ | \\ \quad st = \ll bSt\gg \vee [\![\ st = \ll bSt\gg\ ]\!]_{(\vee,\text{false})} \\ \quad \wedge \ll CLI\gg'\ [\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ] \\ \quad \wedge \ll pOp\gg[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ] \\ \ ]\end{array}}\ [\text{cl-stc-uop-pre}]$$

$$\frac{\begin{array}{l}\Gamma\ \vdash \ll CLI\gg \wedge \ll pOp\gg, \\ \quad \Gamma\ \vdash \exists\ \text{pre}\ \ll Cl\gg_{\Delta}\ll Op\gg \bullet \text{true}\end{array}}{\text{true}}\ \begin{array}{l}[\text{cl-stc-uop-epre-np}] \\ [[\![\ \ll atT\gg \neq \varnothing\ ]\!] \wedge [\![\ \ll ioT\gg \neq \varnothing\ ]\!]]\end{array}$$

Meta-Lemmas.

$$\frac{\text{pre}(\ll Cl\gg_{\Delta}\ll Op\gg \wedge \ll P\gg)}{\begin{array}{l}[\ \ll Cl\gg;\ [\![\ \ll io\gg? : \ll ioT\gg\ ]\!]\ |\ \ll CLI\gg'\ [\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ] \\ \quad \wedge \ll pOp\gg[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ]\ ] \\ \quad \wedge \text{pre}\ll P\gg[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!], st' := \ll aSt\gg\ ]\end{array}}\ [\text{cl-stc-uop-d-pre}]$$

□

Template T11 (Class with statechart, intensional observe operation)

These operations perform an observation upon the state of one object. The template operation schema is:

$\ll Cl\gg_{\Xi}\ll Op\gg$

$\Xi \ll Cl\gg_0$
$\ll out\gg! : \ll oT\gg$

$st = \ll bSt\gg \vee [\![\ st = \ll bSt\gg\ ]\!]_{(\vee,\text{false})}$
$\ll pOp\gg$
$\ll out\gg! = \ll ov\gg$

Above, in the declarations, changes to the state of the object are disallowed (Ξ declaration), the output captures the observation (it may be of any type). In the predicate, there is an optional extra predicate, a disjunction of state transition preconditions as defined in the statechart and an equation which gives the value to the output.

There are two proof obligations associated with this template. The first demonstrates that the operation is *well-formed* The second is the precondition conjecture.

$\vdash?\ \forall \Xi \ll Cl\gg;\ \ll out\gg! : \ll oT\gg \bullet \ll ov\gg \in \ll oT\gg$
$\vdash?\ \exists \text{pre}\ll Cl\gg_{\Xi}\ll Op\gg \bullet \text{true}$

Meta-Theorems. The first meta-theorem simplifies the operation's precondition. The second, gives *true by construction* on the precondition conjecture, provided the precondition of the operation is *true*.

$$\frac{\text{pre } \lessdot Cl \gtrdot_{\Xi} \lessdot Op \gtrdot}{[\ \lessdot Cl \gtrdot \mid \lessdot pOp \gtrdot [\ \theta \lessdot Cl \gtrdot' := \theta \lessdot Cl \gtrdot, \lessdot out \gtrdot ! := \lessdot ov \gtrdot\ ] \\ \quad \wedge\ st = \lessdot bSt \gtrdot \vee [\![\ st = \lessdot bSt \gtrdot\ ]\!]_{(\vee,\text{false})}\ ]} \quad [\text{cl-stc-oop-pre}]$$

$$\frac{\Gamma\ \vdash \lessdot pOp \gtrdot,\ \Gamma\ \vdash \exists\ \text{pre } \lessdot Cl \gtrdot_{\Xi} \lessdot Op \gtrdot \bullet \text{true}}{\text{true}} \quad [\text{cl-stc-oop-epre-np}]$$

□

Template T12 (Class with statechart, intensional finalisation)

The finalisation describes a condition for the objects of the class to cease their existence. This condition is expressed as a conjunction of the precondition of the final transitions of the statechart with an extra predicate (variable $\lessdot fc \gtrdot$).

$\lessdot Cl \gtrdot FinSt$

$\lessdot Cl \gtrdot St$

$st = \lessdot bSt \gtrdot \vee [\![\ st = \lessdot bSt \gtrdot\ ]\!]_{(\vee,\text{false})}$

$\lessdot Cl \gtrdot Fin_0$

$\lessdot Cl \gtrdot_0$

$[\![\ \lessdot if \gtrdot ? : \lessdot ifT \gtrdot\ ]\!]$

$\lessdot fc \gtrdot$

$\lessdot Cl \gtrdot Fin$

$\lessdot Cl \gtrdot Fin_0$

$\lessdot Cl \gtrdot FinSt$

The consistency conjecture requires a proof that the finalisation is a valid intensional state.

$$\vdash ?\ \exists\ \lessdot Cl \gtrdot Fin \bullet \text{true}$$

Meta-Theorems The meta-theorems simplify the finalisation consistency conjecture. The first is the more general case, where an invariant is defined for the class. The second gives a stronger result when the class intension does not have an invariant.

$$\frac{\Gamma\ \vdash \exists\ \lessdot Cl \gtrdot Fin \bullet \text{true}}{\Gamma\ \vdash \exists\ [\![\ \lessdot at \gtrdot : \lessdot atT \gtrdot\ ]\!]; [\![\ \lessdot if \gtrdot ? : \lessdot ifT \gtrdot\ ]\!] \bullet \lessdot CLI \gtrdot \wedge \lessdot fc \gtrdot} \quad [\text{cl-stc-fin}]$$

$$\frac{\Gamma\ \vdash \lessdot CLI \gtrdot \wedge \lessdot fc \gtrdot,\ \Gamma\ \vdash \exists\ \lessdot Cl \gtrdot Fin \bullet \text{true}}{\text{true}} \quad [\text{cl-stc-fin-ni}]$$

$$[\![\ \lessdot ifT \gtrdot \neq \varnothing\ ]\!] \wedge [\![\ \lessdot atT \gtrdot \neq \varnothing\ ]\!]$$

□

Template T13 (Class with statechart, history invariant)

This defines a condition that must be preserved by all the operations of the superclass. To avoid subclasses changing the behaviour described by the statechart, subclass extra operations must extend the following *history invariant* schema, which disallows change to the *st* attribute.

$$
\begin{array}{|l}
\hline
\ll Cl \gg H \\
\Delta \ll Cl \gg \\
\hline
st = st' \\
\hline
\end{array}
$$

□

Template T14 (Subclass intensional state space and initialisation)

The state space schema defines the state attributes of the objects of the class, inherits the attributes of the superclass and expresses an invariant. The initialisation makes an assignment of values to state attributes. There is an auxiliary definition of the state space, which defines only what is specific to the class being defined; this helps to control the frame when defining system operations:

$$
\begin{array}{|l}
\hline
\ll Cl \gg_0 \\
[\![\ll at \gg : \ll atT \gg]\!] \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\ll Cl \gg Init_I \\
\ll PCl \gg Init_I \\
[\![\ll ii \gg ? : \ll iiT \gg]\!] \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\ll Cl \gg \\
\ll Cl \gg_0 \\
\ll PCl \gg \\
\hline
\ll CLI \gg \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\ll Cl \gg Init \\
\ll Cl \gg ' \\
\ll PCl \gg Init \\
\ll Cl \gg Init_I \\
\hline
\ll pI \gg \\
[\![\ll at \gg' = \ll iv \gg]\!] \\
\hline
\end{array}
$$

There are two proof obligations. The first demonstrates that the initialisation is *well-formed*. That is, the state attributes of the subclass are assigned valid values according to the type of the attribute. The second is the usual initialisation conjecture.

$$\vdash?\ \forall \ll Cl \gg Init_I \bullet [\![\ll iv \gg \in \ll atT \gg]\!]$$

$$\vdash?\ \exists \ll Cl \gg Init \bullet \text{true}$$

The initialisation refinement conjecture is not required because by theorem scl-init-refit is always true.

Meta-Theorems. The meta-theorems simplify the initialisation conjecture by relying on the proof of the well-formedness conjecture. The first says that the proof of the existence of the initial state reduces to a proof of satisfaction of the invariant in the initial state. The second conjecture gives *true by construction*: if there is no class invariant and no initialisation condition then there is an initial state.

$$\frac{\Gamma \vdash \exists \ll Cl \gg Init \bullet \text{true}}{\Gamma \vdash \exists \ll PCl \gg Init;\ \ll Cl \gg Init_I \bullet \ll CLI \gg' [\![\ll at \gg' := \ll iv \gg]\!]] \wedge \ll pI \gg [\![\ll at \gg' := \ll iv \gg]\!]]} \quad \text{[scl-init]}$$

$$\frac{\Gamma \vdash \ll CLI \gg \wedge \ll pI \gg, \Gamma \vdash \exists \ll Cl \gg Init \bullet \text{true}}{\text{true}} \quad \text{[scl-init-ni]} \quad [\![\ll atTs \gg \neq \varnothing]\!]$$

$$\frac{\Gamma \vdash \forall \ll Cl \gg Init \bullet \ll PCl \gg Init}{\text{true}} \quad \text{[scl-init-ref]}$$

□

Template T15 (Subclass intensional update operation, extension)

A subclass intensional update operation extends a superclass operation by specifying the extra behaviour of the subclass. This is expressed in the template by including the superclass operations and then adding in the predicate the extra behaviour.[5]

$\ll Cl \gg_\Delta \ll Op \gg$

$\Delta \ll Cl \gg$
$\ll PCl \gg_\Delta \ll Op \gg$
$[\![\ll io \gg ? : \ll ioT \gg]\!]$

$\ll pOp \gg$
$[\![\ll at \gg' = \ll nv \gg]\!]$

The conjectures of this template are as follows. The first demonstrates that the operation is *well-formed*. The second is the usual consistency precondition conjecture. The third is a conditional refinement applicability conjecture; conditional because the applicability conjecture is proved only if the superclass is not abstract. The correctness conjecture of the inheritance refinement is not required because it is always true, both for the blocking (meta-theorem scl-uopx-refc-bl) and non-blocking (meta-theorem scl-uopx-refc-nbl) interpretations of refinement.

$\vdash ?\ \forall \Delta \ll Cl \gg; [\![\ll io \gg ? : \ll ioT \gg]\!] \bullet [\![\ll nv \gg \in \ll atT \gg]\!]$
$\vdash ?\ \exists \text{pre} \ll Cl \gg_\Delta \ll Op \gg \bullet \text{true}$
$\vdash ?\ \ll PCl \gg Cl \notin abstractCl \Rightarrow \forall (\text{pre} \ll PCl \gg_\Delta \ll Op \gg \Rightarrow \text{pre} \ll Cl \gg_\Delta \ll Op \gg) \bullet \text{true}$

Meta-Theorems. There are several meta-theorems to simplify the operation's preconditions and consistency conjectures. The first simplifies the operation's precondition.

$$\frac{\text{pre} \ll Cl \gg_\Delta \ll Op \gg}{\text{pre}(\ll PCl \gg_\Delta \ll Op \gg \wedge [\ \ll Cl \gg;\ [\![\ll io \gg ? : \ll ioT \gg]\!] \mid \ll CLI \gg' [\ [\![\ll at \gg' := \ll nv \gg]\!] \ ] \wedge \ll pOp \gg [\ [\![\ll at \gg' := \ll nv \gg]\!] \ ] \ ])} \quad \text{[scl-uopx-pre]}$$

[5] This describes a deterministic operation; a non-deterministic operation would require another template.

The second gives *true by construction* on the precondition conjecture, provided there is no class invariant and no operation's condition.

$$\frac{\Gamma \vdash \lessdot CLI \gtrdot \wedge \lessdot pOp \gtrdot, \quad \Gamma \vdash \exists \ \text{pre} \ \lessdot Cl \gtrdot_{\Delta} \lessdot Op \gtrdot \bullet \text{true}}{\text{true}} \ [\text{scl-uopx-epre-np}]$$

$$[[\![\lessdot atT \gtrdot \neq \varnothing]\!] \wedge [\![\lessdot ioT \gtrdot \neq \varnothing]\!]]$$

The third meta-theorem says that the non-blocking refinement correctness conjecture is always true.

$$\frac{\Gamma \vdash \forall(\text{pre} \ \lessdot PCl \gtrdot_{\Delta} \lessdot Op \gtrdot \wedge \lessdot Cl \gtrdot_{\Delta} \lessdot Op \gtrdot) \bullet \lessdot PCl \gtrdot_{\Delta} \lessdot Op \gtrdot}{\text{true}} \ [\text{scl-uopx-refc-nbl}]$$

The fourth meta-theorem says that the blocking correctness refinement conjecture is always true.

$$\frac{\Gamma \vdash \forall(\lessdot Cl \gtrdot_{\Delta} \lessdot Op \gtrdot) \bullet \lessdot PCl \gtrdot_{\Delta} \lessdot Op \gtrdot}{\text{true}} \ [\text{scl-uopx-refc-bl}]$$

The fifth meta-theorem says that if the superclass is abstract, then the conditional applicability conjecture is true.

$$\frac{\Gamma \vdash \lessdot PCl \gtrdot Cl \in abstractCl, \quad \Gamma \vdash \lessdot PCl \gtrdot Cl \notin abstractCl \Rightarrow \forall(\text{pre} \lessdot PCl \gtrdot_{\Delta} \lessdot Op \gtrdot \Rightarrow \text{pre} \lessdot Cl \gtrdot_{\Delta} \lessdot Op \gtrdot) \bullet \text{true}}{\text{true}} \ [\text{scl-uopx-refa-pa}]$$

The sixth meta-theorem says that if the superclass is not abstract, the subclass does not have its own invariant and the operation has no extra condition, then the conditional applicability conjecture is true.

$$\frac{\Gamma \vdash \lessdot PCl \gtrdot Cl \notin abstractCl \wedge \lessdot CLI \gtrdot \wedge \lessdot pOp \gtrdot, \quad \Gamma \vdash \lessdot PCl \gtrdot Cl \notin abstractCl \Rightarrow \forall(\text{pre} \lessdot PCl \gtrdot_{\Delta} \lessdot Op \gtrdot \Rightarrow \text{pre} \lessdot Cl \gtrdot_{\Delta} \lessdot Op \gtrdot) \bullet \text{true}}{\text{true}} \ [\text{scl-uopx-refa-np}]$$

□

Template T16 (Subclass intensional observe operation, extension)

These operations perform an observation upon the state of one object. The operation is alreay defined in the superclass, and so it is extended in the subclass:

$$\lessdot Cl \gtrdot_{\Xi} \lessdot Op \gtrdot == \Xi \lessdot Cl \gtrdot \wedge \lessdot PCl \gtrdot_{\Xi} \lessdot Op \gtrdot$$

All required proof obligations do not need to be proved because they are all true by construction (see below).

Meta-Theorems. The following meta-theorems say: (a) the precondition of the subclass operation is the precondition of the parent (meta-theorem scl-oopx-pre); (b) the consistency precondition conjecture is always true (meta-theorem scl-oopx-epre); (c) the refinement correctness conjecture for the non-blocking setting is always true (meta-theorem scl-oopx-refc-nbl); (d) the refinement correctness conjecture for the blocking setting is also always true (meta-theorem scl-oopx-refc-bl); and (e) the refinement applicability conjecture is also always true (meta-theorem scl-oopx-refa).

$$\frac{\text{pre}\ll Cl\gg_{\Xi}\ll Op\gg}{\text{pre}\ll PCl\gg_{\Xi}\ll Op\gg}\ [\text{scl-oopx-pre}] \qquad \frac{\Gamma\ \vdash \exists\,\text{pre}\ll Cl\gg_{\Xi}\ll Op\gg \bullet \text{true}}{\text{true}}\ [\text{scl-oopx-epre}]$$

$$\frac{\Gamma\ \vdash \forall(\text{pre}\ \ll PCl\gg_{\Xi}\ll Op\gg \wedge \ll Cl\gg_{\Xi}\ll Op\gg) \bullet \ll PCl\gg_{\Xi}\ll Op\gg}{\text{true}}\ [\text{scl-oopx-refc-nbl}]$$

$$\frac{\Gamma\ \vdash \forall(\ll Cl\gg_{\Xi}\ll Op\gg) \bullet \ll PCl\gg_{\Xi}\ll Op\gg}{\text{true}}\ [\text{scl-oopx-refc-bl}]$$

$$\frac{\Gamma\ \vdash \forall(\text{pre}\ll PCl\gg_{\Xi}\ll Op\gg \Rightarrow \text{pre}\ll Cl\gg_{\Xi}\ll Op\gg) \bullet \text{true}}{\text{true}}\ [\text{scl-oopx-refa}]$$

□

Template T17 (Subclass intensional extra update operation)

This template describes a subclass update operation that is not a specialisation. The template operation schema is expressed in the usual way, there is an assignment of values to the attribute of the class; if the attributes of the superclass do not change then this can be said in the declarations (optional).

$\ll Cl\gg_{\Delta}\ll Op\gg$

$\Delta \ll Cl\gg$

$(\!|\ \Xi\ll PCl\gg \ [\!]\ \Delta\ll PCl\gg H\ |\!)$

$[\![\ \ll io\gg? : \ll ioT\gg\ ]\!]$

$\ll pOp\gg$

$[\![\ \ll at\gg' = \ll nv\gg\ ]\!]$

There are two proof obligations associated with this template. The first demonstrates that the operation is *well-formed*. The second is the usual precondition conjecture. Note that it is not required to prove the refinement conjectures because they are always true for extra operations in the subclass (see main text).

$\vdash?\ \forall \Delta \ll Cl\gg;\ [\![\ \ll io\gg? : \ll ioT\gg\ ]\!] \bullet [\![\ \ll nv\gg \in \ll atT\gg\ ]\!]$

$\vdash?\ \exists\,\text{pre}\ll Cl\gg_{\Delta}\ll Op\gg \bullet \text{true}$

Meta-Theorems. The first meta-theorem simplifies the operation's precondition. The second gives *true by construction* on the precondition conjecture, provided there is no class invariant and no operation's condition.

$$\frac{\text{pre}\ll Cl\gg_{\Delta}\ll Op\gg}{\begin{array}{l}\text{pre}(\llparenthesis\ \Xi\ll PCl\gg\ [\![\ \Delta\ll PCl\gg H\ ]\!]\ \rrparenthesis \wedge [\ \ll Cl\gg;\ [\![\ \ll io\gg? : \ll ioT\gg\ ]\!]\ | \\ \quad \ll CLI\gg' [\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ] \\ \quad \wedge \ll pOp\gg[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ]\ ])\end{array}} \quad [\text{scl-uopn-pre}]$$

$$\frac{\begin{array}{l}\Gamma\ \vdash \ll CLI\gg \wedge \ll pOp\gg, \\ \quad \Gamma\ \vdash \exists\ \text{pre}\ \ll Cl\gg_{\Delta}\ll Op\gg \bullet \text{true}\end{array}}{\text{true}} \quad \begin{array}{l}[\text{scl-uopn-epre-np}] \\ [[\![\ \ll atT\gg \neq \varnothing\ ]\!] \wedge [\![\ \ll ioT\gg \neq \varnothing\ ]\!]]\end{array}$$

Meta-Lemmas.

$$\frac{\Gamma\ \vdash \exists\ll Cl\gg' \bullet \ll P\gg \wedge \ll Cl\gg_{\Delta}\ll Op\gg}{\Gamma\ \vdash P[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ] \wedge \text{pre}\ll Cl\gg_{\Delta}\ll Op\gg} \quad [\text{scl-uopn-d-pre}]$$

□

Template T18 (Subclass intensional finalisation)

The finalisation describes a condition for the objects of the class to cease their existence. The subclass finalisation extends the finalisation of the superclass.

$\ll Cl\gg Fin$

$\ll PCl\gg Fin$
$\ll Cl\gg$

$\ll fc\gg$

The consistency conjecture requires a proof that the finalisation is a valid intensional state. The finalisation refinement conjecture is always true by meta-theorem int-fin-ref-scl.

$$\vdash?\ \exists\ \ll Cl\gg Fin \bullet \text{true}$$

Meta-Theorems. The meta-theorems simplify the consistency and refinement conjectures for finalisation. The first meta-theorem is more general: it gives a simplification of the consistency conjecture for the general case. The second is more specific but gives a stronger result: when the subclass does not have an extra invariant and there is no extra finalisation condition then the consistency conjecture is true by construction.

$$\frac{\Gamma\ \vdash \exists\ \ll Cl\gg Fin \bullet \text{true}}{\Gamma\ \vdash \exists\ \ll PCl\gg Fin;\ [\![\ \ll at\gg : \ll atT\gg\ ]\!] \bullet \ll CLI\gg \wedge \ll fc\gg} \quad [\text{scl-fin}]$$

$$\frac{\Gamma\ \vdash \ll CLI\gg \wedge \ll fc\gg,\ \Gamma\ \vdash \exists\ \ll Cl\gg Fin \bullet \text{true}}{\text{true}} \quad [\text{scl-fin-ni}]$$

The following says that the finalisation refinement conjecture is always true.

$$\frac{\Gamma \vdash \forall \ll Cl\gg Fin \bullet \ll PCl\gg Fin}{\text{true}} \quad [\text{scl-fin-ref}]$$

□

D.2.2 Extensional View

This view defines the extensional meaning of each class as an ADT representing the set of existing objects of a class. This ADT includes operations, which may act upon specific individual objects, or on the set as a whole. The templates of this view are given below.

Template T19 (Class extensional state space and initialisation)

The state space of class extension is defined by instantiating the $\mathbb{S}$CL generic of the ZOO toolkit (appendix C). This defines the set of existing objects and a function mapping existing objects to their states. The template describes the instantiation of this generic; the renaming in the generic instantiation is to avoid name clashes when class extensions are composed to make the system schema:

$$\mathbb{S}\ll Cl\gg == [\ \mathbb{S}\text{CL}[\mathbb{O}\ll Cl\gg Cl, \ll Cl\gg][s\ll Cl\gg / os, st\ll Cl\gg / oSt] \\ (\!|\ \ | \ \ s\ll Cl\gg \cap \mathbb{O}_x\ll Cl\gg Cl = \varnothing \ |\!)^? \ ]$$

The initialisation assigns both the set of existing objects and the set of object-to-state mappings to the empty set: in the initial state there are no objects:

$$\mathbb{S}\ll Cl\gg Init == [\ \mathbb{S}\ll Cl\gg\ ' \mid s\ll Cl\gg' = \varnothing \land st\ll Cl\gg' = \varnothing \ ]$$

$$\vdash? \ll Cl\gg Cl \in rootCl$$

$$\vdash? \ll Cl\gg Cl \in abstractCl \Leftrightarrow (\mathbb{S}\ll Cl\gg \Rightarrow s\ll Cl\gg \cap \mathbb{O}_x\ll Cl\gg Cl = \varnothing)$$

There is one well-formedness conjecture, which requires that the class is a root of some inheritance hierarchy (that is, it has no ascendants). The initialisation conjecture is not required because by meta-theorem cl-ext-init (below) it is always true.

Meta-theorems. The following gives *true by construction* on the initialisation conjecture.

$$\frac{\Gamma \vdash \exists \mathbb{S}\ll Cl\gg Init \bullet \text{true}}{\text{true}} \quad [\text{cl-ext-init}]$$

Meta-lemmas.

$$\frac{\Gamma \vdash \exists\ \mathbb{S}\ll Cl\gg Init \bullet \ll P\gg}{\Gamma \vdash \ll P\gg[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ]} \quad [\text{cl-ext-init-d}]$$

$$\frac{\Gamma \vdash \exists\ [\![\ \mathbb{S}\ll Cl\gg Init \bullet \ll P\gg\ ]\!]}{\Gamma \vdash\ \ll P\gg[\ [\![\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ]\!]\ ]} \quad [\text{cl-ext-init-lsd}]$$

□

Template T20 (Class, new promotion frame)

The new promotion frame adds a new object to the set of existing objects. There is a constraint to ensure that the new atom is not one of the existing objects of the class.

$$
\begin{array}{|l}
\Phi\mathbb{S}{\ll}Cl{\gg}N \\
\hline
\Delta\mathbb{S}{\ll}Cl{\gg} \\
{\ll}Cl{\gg}\ ' \\
o{\ll}Cl{\gg}! : \mathbb{O}_x{\ll}Cl{\gg}Cl \\
\hline
o{\ll}Cl{\gg}! \in \mathbb{O}_x{\ll}Cl{\gg}Cl \setminus s{\ll}Cl{\gg} \\
s{\ll}Cl{\gg}' = s{\ll}Cl{\gg} \cup \{o{\ll}Cl{\gg}!\} \\
st{\ll}Cl{\gg}' = st{\ll}Cl{\gg} \cup \{o{\ll}Cl{\gg}! \mapsto \theta{\ll}Cl{\gg}\ '\} \\
\hline
\end{array}
$$

□

Template T21 (Class, new operation)

A class operation of type new creates new objects of the class. That is, it adds a new object to the set of existing objects. These operations are formed as a new promotion, which is made up of the new framing schema of the class and the intensional initialisation:

$$\mathbb{S}_\Delta{\ll}Cl{\gg}New == \exists{\ll}Cl{\gg}\ ' \bullet \Phi\mathbb{S}{\ll}Cl{\gg}N \land {\ll}Cl{\gg}Init$$

$$\vdash?\ {\ll}Cl{\gg}Cl \in rootCl$$

$$\vdash?\ {\ll}Cl{\gg}Cl \notin abstractCl$$

There are two well-formedness conjectures. The first requires the class to be a root (no ascendants), and the second requires that the class is not abstract. The precondition consistency conjecture is not required because by meta-theorem cl-ext-nop-epre (below) it is always true.

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second meta-theorem says that the precondition consistency conjecture is always true, provided that the initialisation conjecture of the intension initialisation has been proved.

$$\frac{\mathrm{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}New}{[\ \mathbb{S}{\ll}Cl{\gg};\ {\ll}Cl{\gg}Init_I \mid \mathbb{O}_x{\ll}Cl{\gg}Cl \setminus s{\ll}Cl{\gg} \neq \varnothing\ ]}\ [\text{cl-ext-nop-pre}]$$

$$\frac{\Gamma\ \vdash \exists\,\mathrm{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}New}{\mathrm{true}}\ [\text{cl-ext-nop-epre}]$$

□

Template T22 (Update Promotion frame)

This frame updates the state of one of the existing objects of the class. The object to update is received as an input and its state is updated to some after state in the set of object-to-state mappings (using functional overriding).

$$
\begin{array}{|l}
\Phi\mathbb{S}{\ll}Cl{\gg}\,U \\
\hline
\Delta\mathbb{S}{\ll}Cl{\gg} \\
\Delta{\ll}Cl{\gg} \\
o{\ll}Cl{\gg}? : \mathbb{O}_x{\ll}Cl{\gg}\,Cl \\
\hline
o{\ll}Cl{\gg}? \in s{\ll}Cl{\gg} \\
\theta{\ll}Cl{\gg} = st{\ll}Cl{\gg}\ o{\ll}Cl{\gg}? \\
s{\ll}Cl{\gg}' = s{\ll}Cl{\gg} \\
st{\ll}Cl{\gg}' = st{\ll}Cl{\gg} \oplus \{o{\ll}Cl{\gg}? \mapsto \theta{\ll}Cl{\gg}\ '\} \\
\hline
\end{array}
$$

□

Template T23 (Class update operation)

An update promoted operation is formed as an update promotion made of the update promotion frame and an intensional operation:

$$\mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg} == \exists\,\Delta{\ll}Cl{\gg} \bullet \Phi\mathbb{S}{\ll}Cl{\gg}\,U \land {\ll}Cl{\gg}_\Delta{\ll}Op{\gg}$$
$$\vdash? \ {\ll}Cl{\gg}\,Cl \in rootCl$$
$$\vdash? \ {\ll}Cl{\gg}\,Cl \notin abstractCl$$

There are two well-formedness conjectures. The first requires the class to be a root (no ascendants), and the second requires that the class is not abstract. The usual consistency conjecture is not required, because by meta-theorem cl-ext-uop-epre it is always true.

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second says that the precondition consistency conjecture is always true, provided the precondition conjecture of the intensional operation (the operation being promoted) has been proved.

$$\frac{\mathrm{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg}}{\begin{array}{l}[\ \mathbb{S}{\ll}Cl{\gg};\ o{\ll}Cl{\gg}? : \mathbb{O}_x{\ll}Cl{\gg}\,Cl \mid \ o{\ll}Cl{\gg}? \in s{\ll}Cl{\gg}\ ] \\ \qquad \land\ \mathrm{pre}{\ll}Cl{\gg}_\Delta{\ll}Op{\gg}[\theta{\ll}Cl{\gg} := st{\ll}Cl{\gg}\ o{\ll}Cl{\gg}?]\end{array}}\ [\text{cl-ext-uop-pre}]$$

$$\frac{\Gamma\ \vdash \exists\,\mathrm{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg}\quad [\text{cl-ext-uop-epre}]}{\mathrm{true}}\ [\exists\,\mathrm{pre}{\ll}Cl{\gg}_\Delta{\ll}Op{\gg} \bullet \mathrm{true}]$$

□

Template T24 (Subclass extensional state space and initialisation)

Like root classes (see above), the state space is defined by instantiating the $\mathbb{S}$CL generic of the ZOO toolkit (appendix C). This defines the set of existing objects and a function mapping existing objects to their states. The template describes the instantiation of this generic; the renaming in the generic instantiation is to avoid name clashes when class extensions are composed to make the system schema:

$$\mathbb{S}{\ll}Cl{\gg} == [\ \mathbb{S}\mathrm{CL}[\mathbb{O}{\ll}Cl{\gg}Cl, {\ll}Cl{\gg}][s{\ll}Cl{\gg}/os, st{\ll}Cl{\gg}/oSt]$$
$$(\!|\ \ |\ \ s{\ll}Cl{\gg} \cap \mathbb{O}_x{\ll}Cl{\gg}Cl = \varnothing\ |\!)^?\ ]$$

The initialisation assigns both the set of existing objects and the set of object-to-state mappings to the empty set: in the initial state there are no objects:

$$\mathbb{S}{\ll}Cl{\gg}Init == [\ \mathbb{S}{\ll}Cl{\gg}\ ' \mid s{\ll}Cl{\gg}' = \varnothing \wedge st{\ll}Cl{\gg}' = \varnothing\ ]$$
$$\vdash?\ {\ll}Cl{\gg}Cl \in abstractCl \Leftrightarrow (\mathbb{S}{\ll}Cl{\gg} \Rightarrow s{\ll}Cl{\gg} \cap \mathbb{O}_x{\ll}Cl{\gg}Cl = \varnothing)$$

It is not required to prove the initialisation conjecture because by meta-theorem scl-ext-init(below) it is always true.

A subclass requires a *subclassing* schema, which expresses the relation between the extensions of the subclass and superclass:

$\mathbb{S}{\ll}ChCl{\gg}Is{\ll}PCl{\gg}$

$\mathbb{S}{\ll}ChCl{\gg};\ \mathbb{S}{\ll}PCl{\gg}$

$$s{\ll}ChCl{\gg} \subseteq s{\ll}PCl{\gg}$$
$$\forall\, o{\ll}ChCl{\gg} : s{\ll}ChCl{\gg} \bullet$$
$$(\lambda\ {\ll}ChCl{\gg} \bullet \theta{\ll}PCl{\gg})(st{\ll}ChCl{\gg}\ o{\ll}ChCl{\gg}) = st{\ll}PCl{\gg}\ o{\ll}ChCl{\gg}$$

This says that the set of existing objects of the subclass is a subset of the superclass set (first predicate), and that the state mappings of subclass and superclass must be consistent with each other (second predicate).

Meta-theorems. The following meta-theorem says that the initialisation conjecture of a subclass extension is always true.

$$\frac{\Gamma \vdash \exists\, \mathbb{S}{\ll}Cl{\gg}Init \bullet \mathrm{true}}{\mathrm{true}}\ [\text{scl-ext-init}]$$

□

Template T25 (Subclass, new intermediate promotion frame of superclass)

This defines the action of a new operation with respect to the superclass extension.

$\Phi\mathbb{S}{\ll}Cl{\gg}NI_0$

$\Delta\mathbb{S}{\ll}Cl{\gg}$
${\ll}Cl{\gg}\ '$
$o{\ll}Cl{\gg}! : \mathbb{O}{\ll}Cl{\gg}Cl$

$$s{\ll}Cl{\gg}' = s{\ll}Cl{\gg} \cup \{o{\ll}Cl{\gg}!\}$$
$$st{\ll}Cl{\gg}' = st{\ll}Cl{\gg} \cup \{o{\ll}Cl{\gg}! \mapsto \theta{\ll}Cl{\gg}\ '\}$$

Meta-theorems. The meta-theorem gives the precondition of the frame.

$$\frac{\mathrm{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}NI_0}{[\ \mathbb{S}{\ll}Cl{\gg}\ ]}\ [\text{scl-ext-ifnp-p-pre}]$$

□

Template T26 (Subclass, new intermediate promotion frame)

This extends the superclass intermediate frame by defining the action of a new operation with respect to the subclass extension.

$$
\begin{array}{|l}
\Phi\mathbb{S}{\ll}Cl{\gg}NI_0 \\
\hline
\Phi\mathbb{S}{\ll}PCl{\gg}NI_0[o{\ll}Cl{\gg}!/o{\ll}PCl{\gg}!] \\
\Delta\mathbb{S}{\ll}ChCl{\gg}Is{\ll}PCl{\gg} \\
{\ll}Cl{\gg}\ ' \\
o{\ll}Cl{\gg}! : \mathbb{O}{\ll}Cl{\gg}Cl \\
\hline
s{\ll}Cl{\gg}' = s{\ll}Cl{\gg} \cup \{o{\ll}Cl{\gg}!\} \\
st{\ll}Cl{\gg}' = st{\ll}Cl{\gg} \cup \{o{\ll}Cl{\gg}! \mapsto \theta{\ll}Cl{\gg}\ '\} \\
\hline
\end{array}
$$

Meta-theorems. The meta-theorem gives the precondition, which is the conjunction of the precondition of the subclass and that of the superclass.

$$\frac{\text{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}NI_0}{[\ \mathbb{S}{\ll}Cl{\gg}\ ] \wedge \text{pre}\ \Phi\mathbb{S}{\ll}PCl{\gg}NI_0} \quad [\text{scl-ext-ifnp-pre}]$$

□

Template T27 (Subclass, new final promotion frame)

This template represents the final subclass new frame, which extends subclass intermediate frame with the required constraint.

$$
\begin{array}{|l}
\Phi\mathbb{S}{\ll}Cl{\gg}NI \\
\hline
\Phi\mathbb{S}{\ll}Cl{\gg}NI_0 \\
\hline
o{\ll}Cl{\gg}! \in \mathbb{O}_x{\ll}Cl{\gg}Cl \setminus s{\ll}Cl{\gg} \\
\hline
\end{array}
$$

Meta-theorems. The following gives the precondition of the frame.

$$\frac{\text{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}NI}{[\ \mathbb{S}{\ll}Cl{\gg} \mid o{\ll}Cl{\gg}! \in \mathbb{O}_x{\ll}Cl{\gg}Cl \setminus s{\ll}Cl{\gg}\ ] \wedge \text{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}NI_0} \quad [\text{scl-ext-ffnp-pre}]$$

□

Template T28 (Subclass, new operation)

A new subclass operation creates new objects of the class. This means that new object needs to be added to the extension of the subclass and all its ascendant classes. These operations are formed as a new promotion, which is made up of the new inheritance framing schema of the subclass and the intensional initialisation:

$$\mathbb{S}_\Delta{\ll}Cl{\gg}New == \exists{\ll}Cl{\gg}\ ' \bullet \Phi\mathbb{S}{\ll}Cl{\gg}NI \wedge {\ll}Cl{\gg}Init$$

$$\vdash ?\ {\ll}Cl{\gg}Cl \notin rootCl \wedge {\ll}Cl{\gg}Cl \notin abstractCl$$

The well-formedness conjectures requires that the class specialises some class (it must not be a root) and that it is not abstract. The precondition consistency conjecture is not required because by meta-theorem scl-ext-nop-epre(below) it is always true.

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second meta-theorem says that the precondition consistency conjecture is always true, provided that the initialisation conjecture of the intension initialisation has been proved.

$$\frac{\text{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}New}{[\ \mathbb{S}{\ll}Cl{\gg};\ {\ll}Cl{\gg}Init_I\ ] \wedge \text{pre}\,\Phi\mathbb{S}{\ll}Cl{\gg}N} \quad [\text{scl-ext-nop-pre}]$$

$$\frac{\Gamma\ \vdash \exists\,\text{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}New}{\text{true}} \quad [\text{scl-ext-nop-epre}]$$

□

Template T29 (Subclass, update intermediate promotion frame of superclass)

This defines the action of an update promoted operation with respect to the superclass extension.

$\Phi\mathbb{S}{\ll}Cl{\gg}UI_0$

$\Delta\mathbb{S}{\ll}Cl{\gg}$
$\Delta{\ll}Cl{\gg}$
$o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl$

$s{\ll}Cl{\gg}' = s{\ll}Cl{\gg}$
$st{\ll}Cl{\gg}' = st{\ll}Cl{\gg} \oplus \{o{\ll}Cl{\gg}? \mapsto \theta{\ll}Cl{\gg}\ '\}$

Meta-theorems. The meta-theorem gives the precondition of the frame.

$$\frac{\Gamma\ \vdash \text{pre}\,\Phi\mathbb{S}{\ll}Cl{\gg}UI_0}{[\ \mathbb{S}{\ll}Cl{\gg};\ o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl\ ]} \quad [\text{scl-ext-ifup-p-pre}]$$

□

Template T30 (Subclass update intermediate promotion frame)

This extends the superclass intermediate frame by defining the action of an update operation with respect to the subclass extension.

$\Phi\mathbb{S}{\ll}Cl{\gg}UI_0$

$\Phi\mathbb{S}{\ll}PCl{\gg}UI_0[o{\ll}Cl{\gg}?/o{\ll}PCl{\gg}?]$
$\Delta\mathbb{S}{\ll}ChCl{\gg}Is{\ll}PCl{\gg}$
$\Delta{\ll}Cl{\gg}$
$o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl$

$s{\ll}Cl{\gg}' = s{\ll}Cl{\gg}$
$st{\ll}Cl{\gg}' = st{\ll}Cl{\gg} \oplus \{o{\ll}Cl{\gg}? \mapsto \theta{\ll}Cl{\gg}\ '\}$

Meta-theorems. The meta-theorem gives the precondition of the frame.

$$\frac{\text{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}\,UI_0}{\begin{array}{l}[\ \mathbb{S}{\ll}Cl{\gg};\ o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl\ ] \\ \quad \wedge\ \text{pre}\ \Phi\mathbb{S}{\ll}PCl{\gg}\,UI_0[o{\ll}Cl{\gg}?/o{\ll}PCl{\gg}?]\end{array}} \quad [\textsf{scl-ext-ifup-pre}]$$

□

Template T31 (Subclass final update promotion frame)

This template represents the final subclass update promotion frame, which extends subclass intermediate frame with the required precondition.

$$\begin{array}{|l}\Phi\mathbb{S}{\ll}Cl{\gg}\,UI \\ \hline \Phi\mathbb{S}{\ll}Cl{\gg}\,UI_0 \\ \hline o{\ll}Cl{\gg}? \in s{\ll}Cl{\gg} \cap \mathbb{O}_x{\ll}Cl{\gg}Cl \\ \theta{\ll}Cl{\gg} = st{\ll}Cl{\gg}\ o{\ll}Cl{\gg}? \\ \hline\end{array}$$

Meta-theorems. The meta-theorem gives the precondition of the frame, which extends the precondition of the superclass intermediate update frame.

$$\frac{\text{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}\,UI_0}{\begin{array}{l}[\ \mathbb{S}{\ll}Cl{\gg};\ o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl \mid o{\ll}Cl{\gg}? \in s{\ll}Cl{\gg} \cap \mathbb{O}_x{\ll}Cl{\gg}Cl\ ] \\ \quad \wedge\ \text{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}\,UI_0\end{array}} \quad [\textsf{scl-ext-ffup-pre}]$$

□

Template T32 (Subclass Update Operation)

An update promoted operation is formed as an update promotion made of the update promotion frame and an intensional operation:

$$\mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg} == \exists\,\Delta{\ll}Cl{\gg} \bullet \Phi\mathbb{S}{\ll}Cl{\gg}\,UI \wedge {\ll}Cl{\gg}_\Delta{\ll}Op{\gg}$$
$$\vdash?\ {\ll}Cl{\gg}Cl \notin rootCl \wedge {\ll}Cl{\gg}Cl \notin abstractCl$$

There are two well-formedness conjectures. The first requires that the class is not a root (that is, it must be a subclass), and the second that the class is not abstract. The precondition consistency conjecture is not required because by meta-theorem scl-ext-uop-epre (below) it is always true.

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second says that the precondition consistency conjecture is always true, provided the precondition conjecture of the intensional operation (the operation being promoted) has been proved.

$$\frac{\text{pre}\ \mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg}}{\text{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}\,UI \wedge \text{pre}{\ll}Cl{\gg}_\Delta{\ll}Op{\gg}[\theta{\ll}Cl{\gg} := st{\ll}Cl{\gg}\ o{\ll}Cl{\gg}?]} \quad [\textsf{scl-ext-uop-pre}]$$

$$\frac{\Gamma \vdash \exists\, \mathrm{pre}\, \mathbb{S}_{\Delta} \lessdot Cl \gtrdot \lessdot Op \gtrdot}{\mathrm{true}} \quad [\mathsf{scl\text{-}ext\text{-}uop\text{-}epre}]$$
$$[\exists\, \mathrm{pre} \lessdot Cl \gtrdot_{\Delta} \lessdot Op \gtrdot \bullet \mathrm{true}]$$

□

Template T33 (Observe Promotion Frame)

An observe promotion frame retrieves the state of one of the existing class objects.

$\Phi\mathbb{S} \lessdot Cl \gtrdot O$

$\Xi\mathbb{S} \lessdot Cl \gtrdot$
$\Xi \lessdot Cl \gtrdot$
$o \lessdot Cl \gtrdot ? : \mathbb{O} \lessdot Cl \gtrdot Cl$

$o \lessdot Cl \gtrdot ? \in s \lessdot Cl \gtrdot$
$\theta \lessdot Cl \gtrdot = st \lessdot Cl \gtrdot\ o \lessdot Cl \gtrdot ?$

□

Template T34 (Observe promoted operation)

The observe promoted operation is formed as an observe promotion made of the observe promotion frame and an observe intensional operation.

$$\mathbb{S}_{\Xi} \lessdot Cl \gtrdot \lessdot Op \gtrdot == \exists\ \Xi \lessdot Cl \gtrdot \bullet \Phi\mathbb{S} \lessdot Cl \gtrdot O \wedge \lessdot Cl \gtrdot_{\Xi} \lessdot Op \gtrdot$$

The consistency conjecture is not required because it is always true (meta-theorem cl-ext-oop-epre, below).

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second gives *true by construction* on the operation's consistency conjecture.

$$\frac{\mathrm{pre}\, \mathbb{S}_{\Xi} \lessdot Cl \gtrdot \lessdot Op \gtrdot}{[\ \mathbb{S} \lessdot Cl \gtrdot;\ o \lessdot Cl \gtrdot ? : \mathbb{O} \lessdot Cl \gtrdot Cl \mid o \lessdot Cl \gtrdot ? \in s \lessdot Cl \gtrdot\ ]\ \wedge\ \mathrm{pre} \lessdot Cl \gtrdot_{\Xi} \lessdot Op \gtrdot [\theta \lessdot Cl \gtrdot := st \lessdot Cl \gtrdot\ o \lessdot Cl \gtrdot ?]} \quad [\mathsf{cl\text{-}ext\text{-}oop\text{-}pre}]$$

□

Template T35 (Delete promotion frame)

A delete promotion frame retrieves the state of the object to delete and removes it from the set of existing objects of the class.

$\Phi\mathbb{S} \lessdot Cl \gtrdot D$

$\Delta\mathbb{S} \lessdot Cl \gtrdot$
$\lessdot Cl \gtrdot$
$o \lessdot Cl \gtrdot ? : \mathbb{O}_x \lessdot Cl \gtrdot Cl$

$o \lessdot Cl \gtrdot ? \in s \lessdot Cl \gtrdot$
$\theta \lessdot Cl \gtrdot = st \lessdot Cl \gtrdot\ o \lessdot Cl \gtrdot ?$
$s \lessdot Cl \gtrdot' = s \lessdot Cl \gtrdot \setminus \{o \lessdot Cl \gtrdot ?\}$
$st \lessdot Cl \gtrdot' = \{o \lessdot Cl \gtrdot ?\} \lhd st \lessdot Cl \gtrdot$

□

Template T36 (Class delete promoted operation)

A delete class operation is formed as a delete promotion made of the delete promotion frame and the intensional finalisation:

$$\mathbb{S}_\Delta{\ll}Cl{\gg}Delete == \exists{\ll}Cl{\gg} \bullet \Phi\mathbb{S}{\ll}Cl{\gg}D \land {\ll}Cl{\gg}Fin$$

$$\vdash? \ {\ll}Cl{\gg}Cl \in rootCl$$

$$\vdash? \ {\ll}Cl{\gg}Cl \notin abstractCl$$

There are two well-formedness conjectures. The first requires that the class is a root (it must not be a subclass), and the second that the class is not abstract. The consistency conjecture is not required because it is always true (meta-theorem cl-ext-dop-epre below).

Meta-theorems The first meta-theorem gives the precondition of the operation. The second gives *true by construction* on the consistency conjecture, provided the intensional finalisation is consistent.

$$\frac{\mathrm{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}Delete}{\begin{array}{l}[\ \mathbb{S}{\ll}Cl{\gg};\ o{\ll}Cl{\gg}? : \mathbb{O}_x{\ll}Cl{\gg}Cl \mid o{\ll}Cl{\gg}? \in s{\ll}Cl{\gg}\ ] \\ \quad \land {\ll}Cl{\gg}Fin[\theta{\ll}Cl{\gg} := st{\ll}Cl{\gg}\ o{\ll}Cl{\gg}?]\end{array}} \quad [\text{cl-ext-dop-pre}]$$

$$\frac{\Gamma \vdash \exists\,\mathrm{pre}\ \mathbb{S}_\Delta{\ll}Cl{\gg}Delete}{\mathrm{true}} \quad [\text{cl-ext-dop-epre}]$$

□

Template T37 (Subclass, intermediate delete promotion frame of superclass)

This defines the action of a delete promoted operation with respect to the superclass extension.

$\Phi\mathbb{S}{\ll}Cl{\gg}DI_0$

$\Delta\mathbb{S}{\ll}Cl{\gg}$

${\ll}Cl{\gg}$

$o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl$

$s{\ll}Cl{\gg}' = s{\ll}Cl{\gg} \setminus \{o{\ll}Cl{\gg}?\}$

$st{\ll}Cl{\gg}' = \{o{\ll}Cl{\gg}?\} \lhd st{\ll}Cl{\gg}$

Meta-theorems. The meta-theorem gives the precondition of the frame.

$$\frac{\mathrm{pre}\ \Phi\mathbb{S}{\ll}Cl{\gg}DI_0}{[\ \mathbb{S}{\ll}Cl{\gg};\ o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl\ ]} \quad [\text{scl-ext-ifdp-p-pre}]$$

□

Template T38 (Subclass intermediate delete promotion frame)

This extends the superclass intermediate frame by defining the action of a delete operation with respect to the subclass extension.

$\Phi\mathbb{S}{\ll}Cl{\gg}DI_0$

$\Phi\mathbb{S}{\ll}PCl{\gg}DI_0[o{\ll}Cl{\gg}?/o{\ll}PCl{\gg}?]$
$\Delta\mathbb{S}{\ll}ChCl{\gg}Is{\ll}PCl{\gg}$
$\ll Cl\gg$
$o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl$

$s{\ll}Cl{\gg}' = s{\ll}Cl{\gg} \setminus \{o{\ll}Cl{\gg}?\}$
$st{\ll}Cl{\gg}' = \{o{\ll}Cl{\gg}?\} \lhd st{\ll}Cl{\gg}$

Meta-theorems. The meta-theorem gives the precondition of the frame.

$$\frac{\text{pre } \Phi\mathbb{S}{\ll}Cl{\gg}DI_0}{[\ \mathbb{S}{\ll}Cl{\gg};\ o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl\ ] \\ \quad \wedge \text{ pre } \Phi\mathbb{S}{\ll}PCl{\gg}DI_0[o{\ll}Cl{\gg}?/o{\ll}PCl{\gg}?]} \quad \text{[scl-ext-ifdp-pre]}$$

□

Template T39 (Subclass final delete promotion frame)

This template represents the final subclass delete promotion frame, which extends subclass intermediate frame with the required precondition.

$\Phi\mathbb{S}{\ll}Cl{\gg}DI$

$\Phi\mathbb{S}{\ll}Cl{\gg}DI_0$

$o{\ll}Cl{\gg}? \in s{\ll}Cl{\gg} \cap \mathbb{O}_x{\ll}Cl{\gg}Cl$
$\theta{\ll}Cl{\gg} = st{\ll}Cl{\gg}\ o{\ll}Cl{\gg}?$

Meta-theorems. The meta-theorem gives the precondition of the frame.

$$\frac{\text{pre } \Phi\mathbb{S}{\ll}Cl{\gg}DI_0}{[\ \mathbb{S}{\ll}Cl{\gg};\ o{\ll}Cl{\gg}? : \mathbb{O}{\ll}Cl{\gg}Cl \mid o{\ll}Cl{\gg}? \in s{\ll}Cl{\gg} \cap \mathbb{O}_x{\ll}Cl{\gg}Cl\ ] \\ \quad \wedge \text{ pre } \Phi\mathbb{S}{\ll}Cl{\gg}DI_0} \quad \text{[scl-ext-ffdp-pre]}$$

□

Template T40 (Subclass delete promoted operation)

A delete class operation is formed as a delete promotion made of the delete promotion frame and the intensional finalisation:

$\mathbb{S}_\Delta{\ll}Cl{\gg}Delete == \exists{\ll}Cl{\gg} \bullet \Phi\mathbb{S}{\ll}Cl{\gg}DI \wedge {\ll}Cl{\gg}Fin$
$\vdash? \ll Cl\gg Cl \notin rootCl$
$\vdash? \ll Cl\gg Cl \notin abstractCl$

There are two well-formedness conjectures. The first requires that the class is not a root (that is, it must be a subclass), and the second that the class is not abstract. The precondition consistency conjecture is not required because by meta-theorem scl-ext-dop-epre (below) it is always true.

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second gives *true by construction* on the consistency conjecture, provided the intensional finalisation is consistent.

$$\frac{\text{pre } \mathbb{S}_\Delta{\ll}Cl{\gg}Delete}{\text{pre}\,\Phi\mathbb{S}{\ll}Cl{\gg}DI \land {\ll}Cl{\gg}Fin[\theta{\ll}Cl{\gg} := st{\ll}Cl{\gg}\ o{\ll}Cl{\gg}?]} \quad [\text{scl-ext-dop-pre}]$$

$$\frac{\Gamma\ \vdash \exists\,\text{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}Delete \bullet \text{true}}{\text{true}} \quad [\text{scl-ext-dop-epre}]$$

□

Template T41 (Class non-promoted delete operations)

This template generates operations that delete a set of objects from the set of existing objects of the class.

$\mathbb{S}_\Delta{\ll}Cl{\gg}Delete$

$\Delta\mathbb{S}{\ll}Cl{\gg}$

$os{\ll}Cl{\gg}? : \mathbb{P}(\mathbb{O}{\ll}Cl{\gg}Cl)$

$s{\ll}Cl{\gg}' = s{\ll}Cl{\gg} \setminus os{\ll}Cl{\gg}?$

$st{\ll}Cl{\gg}' = os{\ll}Cl{\gg}? \lhd st{\ll}Cl{\gg}$

□

Template T42 (Class non-promoted observe operations)

These operations perform an observation upon the set of existing objects of the class. The template describes the observation as output of some type that is assigned an expression that performs the actual observation.

$\mathbb{S}_\Xi{\ll}Cl{\gg}{\ll}Op{\gg}$

$\Xi\mathbb{S}{\ll}Cl{\gg}$

${\ll}oOp{\gg}! : {\ll}oOpT{\gg}$

${\ll}oOp{\gg}! = {\ll}oOpv{\gg}$

All that is required to prove is that the operation is *well-formed*, that is, that the output is given a value that belongs to the output's type. Provided this is so, the usual consistency conjecture is not required (meta-theorem cl-ext-onp-epre, below).

$$\vdash?\ \forall \Xi\mathbb{S}{\ll}Cl{\gg};\ {\ll}oOp{\gg}! : {\ll}oOpT{\gg} \bullet {\ll}oOpv{\gg} \in {\ll}oOpT{\gg}$$

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second says that the precondition consistency conjecture is always true, provided the operation is *well-formed*.

$$\frac{\text{pre}\,\mathbb{S}_{\Xi}\llparenthesis Cl\rrparenthesis\llparenthesis Op\rrparenthesis}{[\ \mathbb{S}\llparenthesis Cl\rrparenthesis \mid \text{true}\ ]} \quad \text{[cl-ext-onp-pre]}$$

$$\frac{\Gamma \vdash \exists\,\text{pre}\,\mathbb{S}_{\Xi}\llparenthesis Cl\rrparenthesis\llparenthesis Op\rrparenthesis \bullet \text{true}}{\text{true}} \quad \text{[cl-ext-onp-epre]}$$

□

Template T43 (New polymorphic operation of an abstract class)

This template generates polymorphic operations of type new for an abstract class. This requires some condition schemas based on the value of an input, which indicates the class object to create.

$$[\![\ CondN\llparenthesis ChCl\rrparenthesis == [\ \llparenthesis cin\rrparenthesis? : \llparenthesis cIT\rrparenthesis \mid \llparenthesis cP\rrparenthesis\ ]\ ]\!]$$
$$\mathbb{S}_{\Delta}\llparenthesis Cl\rrparenthesis New ==$$
$$\quad (\!|\ CondN\llparenthesis ChCl\rrparenthesis \wedge\ |\!)^{?}\mathbb{S}_{\Delta}\llparenthesis ChCl\rrparenthesis New[o\llparenthesis Cl\rrparenthesis!/o\llparenthesis ChCl\rrparenthesis!]$$
$$\quad\quad \wedge [\![\ \Xi\mathbb{S}\llparenthesis ChCl\rrparenthesis\ ]\!]$$
$$\quad \vee [\![\ (\!|\ CondN\llparenthesis ChCl\rrparenthesis \wedge\ |\!)^{?}\mathbb{S}_{\Delta}\llparenthesis ChCl\rrparenthesis New[o\llparenthesis Cl\rrparenthesis!/o\llparenthesis ChCl\rrparenthesis!]$$
$$\quad\quad \wedge [\![\ \Xi\mathbb{S}\llparenthesis ChCl\rrparenthesis\ ]\!]\ ]\!]$$
$$\vdash? \llparenthesis Cl\rrparenthesis Cl \in abstractCl$$

There is one well-formedness conjecture, requiring the class to be abstract. The precondition consistency conjecture is not required because it is always true (meta-theorem acl-ext-nop-poly-epre).

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second says that the precondition consistency conjecture is always true.

$$\frac{\text{pre}\,\mathbb{S}_{\Delta}\llparenthesis Cl\rrparenthesis\llparenthesis Op\rrparenthesis New}{\begin{array}{l} CondN\llparenthesis ChCl\rrparenthesis \\ \quad \wedge \text{pre}\ \mathbb{S}_{\Delta}\llparenthesis ChCl\rrparenthesis New[o\llparenthesis Cl\rrparenthesis?/o\llparenthesis ChCl\rrparenthesis?] \\ \vee [\![\ CondN\llparenthesis ChCl\rrparenthesis \\ \quad \wedge \text{pre}\ \mathbb{S}_{\Delta}\llparenthesis ChCl\rrparenthesis New[o\llparenthesis Cl\rrparenthesis?/o\llparenthesis ChCl\rrparenthesis?]\ ]\!] \end{array}} \quad \text{[acl-ext-nop-poly-pre]}$$

$$\frac{\Gamma\ \vdash \exists\,\text{pre}\,\mathbb{S}_{\Delta}\llparenthesis Cl\rrparenthesis\llparenthesis Op\rrparenthesis New \bullet \text{true}}{\text{true}} \quad \text{[acl-ext-nop-poly-epre]}$$

□

Template T44 (Abstract class, update polymorphic operation)

This template generates polymorphic operations of type update for an abstract class. This is expressed as a disjunction of update operations of subclasses of the class.

$$\mathbb{S}_{\Delta}\llparenthesis Cl\rrparenthesis\llparenthesis Op\rrparenthesis == \mathbb{S}_{\Delta}\llparenthesis ChCl\rrparenthesis\llparenthesis Op\rrparenthesis(\!|\ P\ |\!)^{?}[o\llparenthesis Cl\rrparenthesis?/o\llparenthesis ChCl\rrparenthesis?] \wedge [\![\ \Xi\mathbb{S}\llparenthesis ChCl\rrparenthesis\ ]\!]$$
$$\quad \vee [\![\ \mathbb{S}_{\Delta}\llparenthesis ChCl\rrparenthesis\llparenthesis Op\rrparenthesis(\!|\ P\ |\!)^{?}[o\llparenthesis Cl\rrparenthesis?/o\llparenthesis ChCl\rrparenthesis?] \wedge [\![\ \Xi\mathbb{S}\llparenthesis ChCl\rrparenthesis\ ]\!]\ ]\!]$$
$$\vdash? \llparenthesis Cl\rrparenthesis Cl \in abstractCl$$

There is one well-formedness conjecture, requiring the class of the operation to be abstract. The precondition consistency conjecture is not required because it is always true by meta-theorem acl-ext-uop-poly-epre.

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second says that the precondition consistency conjecture is always true.

$$\frac{\mathrm{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg}}{\begin{array}{l}\mathrm{pre}\,\mathbb{S}_\Delta{\ll}ChCl{\gg}{\ll}Op{\gg}(\!|\ P\ |\!)^?[o{\ll}Cl{\gg}?/o{\ll}ChCl{\gg}?] \\ \quad \lor [\![\ \mathrm{pre}\,\mathbb{S}_\Delta{\ll}ChCl{\gg}{\ll}Op{\gg}(\!|\ P\ |\!)^?[o{\ll}Cl{\gg}?/o{\ll}ChCl{\gg}?]\ ]\!]\end{array}}\ [\mathsf{acl\text{-}ext\text{-}uop\text{-}poly\text{-}pre}]$$

$$\frac{\Gamma \vdash \exists\,\mathrm{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg} \bullet \mathrm{true}}{\mathrm{true}}\ [\mathsf{acl\text{-}ext\text{-}uop\text{-}poly\text{-}epre}]$$

□

Template T45 (Abstract Class, delete polymorphic operation)

This template generates polymorphic operations of type delete for an abstract class. This is expressed as a disjunction of the delete operations of subclasses of the class.

$$\begin{array}{l}\mathbb{S}_\Delta{\ll}Cl{\gg}Delete == \mathbb{S}_\Delta{\ll}ChCl{\gg}Delete(\!|\ P\ |\!)^?[o{\ll}Cl{\gg}?/o{\ll}ChCl{\gg}?] \land [\![\ \Xi\mathbb{S}{\ll}ChCl{\gg}\ ]\!] \\ \quad \lor [\![\ \mathbb{S}_\Delta{\ll}ChCl{\gg}Delete(\!|\ P\ |\!)^?[o{\ll}Cl{\gg}?/o{\ll}ChCl{\gg}?] \land [\![\ \Xi\mathbb{S}{\ll}ChCl{\gg}\ ]\!]\ ]\!] \\ \vdash ?\ {\ll}Cl{\gg}Cl \in abstractCl\end{array}$$

There is one well-formedness conjecture, requiring the class of the operation to be abstract. The precondition consistency conjecture is not required because it is always true by meta-theorem acl-ext-dop-poly-pre.

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second says that the precondition consistency conjecture is always true.

$$\frac{\mathrm{pre}\,\mathbb{S}_\Delta{\ll}Cl{\gg}Delete}{\begin{array}{l}\mathrm{pre}\,\mathbb{S}_\Delta{\ll}ChCl{\gg}Delete(\!|\ P\ |\!)^?[o{\ll}Cl{\gg}?/o{\ll}ChCl{\gg}?] \\ \quad \lor [\![\ \mathrm{pre}\,\mathbb{S}_\Delta{\ll}ChCl{\gg}Delete(\!|\ P\ |\!)^?[o{\ll}Cl{\gg}?/o{\ll}ChCl{\gg}?]\ ]\!]\end{array}}\ [\mathsf{acl\text{-}ext\text{-}dop\text{-}poly\text{-}pre}]$$

$$\frac{\Gamma\ \vdash \exists\ \mathrm{pre}\ \mathbb{S}_\Delta{\ll}Cl{\gg}Delete \bullet \mathrm{true}}{\mathrm{true}}\ [\mathsf{acl\text{-}ext\text{-}dop\text{-}poly\text{-}epre}]$$

□

Template T46 (Class, update polymorphic operation)

This template generates update polymorphic operations for some class. This is expressed as a disjunction of the operation on the superclass and the same operation on its subclasses.

$$\begin{array}{l}\mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg}_P == \mathbb{S}_\Delta{\ll}Cl{\gg}{\ll}Op{\gg} \land \Xi\mathbb{S}{\ll}ChCl{\gg} \land [\![\ \Xi\mathbb{S}{\ll}ChCl{\gg}\ ]\!] \\ \quad \lor \mathbb{S}_\Delta{\ll}ChCl{\gg}{\ll}Op{\gg}(\!|\ P\ |\!)^?[o{\ll}Cl{\gg}?/o{\ll}ChCl{\gg}?] \land [\![\ \Xi\mathbb{S}{\ll}ChCl{\gg}\ ]\!] \\ \quad \lor [\![\ \mathbb{S}_\Delta{\ll}ChCl{\gg}{\ll}Op{\gg}(\!|\ P\ |\!)^?[o{\ll}Cl{\gg}?/o{\ll}ChCl{\gg}?] \land [\![\ \Xi\mathbb{S}{\ll}OChCl{\gg}\ ]\!]\ ]\!]\end{array}$$

□

D.3 Relational View

The relational view defines the associations of a OO model. Associations are defined as Z ADTs, comprising a state space, initialisation and operations.

Template T47 (State space and initialisation)

An association denotes a set of object links or tuples. This is described as a Z relation between the object sets of the classes being related, which denotes a set of object-tuple-pairs (a set of objects links). The initialisation sets the set of links to the empty set: in the initial state there are no objects, hence, no links between them.

$$\mathbb{A}{\ll}As{\gg} == [\ r{\ll}As{\gg} : \mathbb{O}{\ll}Cl_A{\gg}Cl \leftrightarrow \mathbb{O}{\ll}Cl_B{\gg}Cl\ ]$$
$$\mathbb{A}{\ll}As{\gg}Init == [\ \mathbb{A}{\ll}As{\gg}\ ' \mid r{\ll}As{\gg}' = \varnothing\ ]$$

The initialisation conjecture is not required because it is always true (meta-theorem assoc-init, below).

Each association requires a schema that links object references of associations to existing objects and that constrains the association according to its multiplicity constraint. This schema is used to build the system structure in the global view (below), based on the Name Predicates pattern [SPT03b], and is not considered further in the relational view. The required constraints are expressed using the mult generic of the ZOO toolkit (appendix C).

Link$\mathbb{A}{\ll}As{\gg}$

$\mathbb{A}{\ll}As{\gg};\ \mathbb{S}{\ll}Cl_A{\gg};\ \mathbb{S}{\ll}Cl_B{\gg}$

$\mathsf{mult}(r{\ll}As{\gg}, s{\ll}Cl_A{\gg}, s{\ll}Cl_B{\gg},\ {\ll}multE{\gg}, {\ll}MS_1{\gg}, {\ll}MS_2{\gg})$

Meta-theorems. The first meta-theorem says that the initialisation conjecture is always true.

$$\frac{\Gamma \vdash \exists\, \mathbb{A}{\ll}As{\gg}Init \bullet \text{true}}{\text{true}}\ [\mathsf{assoc\text{-}init}]$$

Meta-lemmas.

$$\frac{\Gamma;\ \exists\, \mathbb{A}{\ll}As{\gg}Init \bullet \text{true}\ \vdash \exists\, \mathbb{A}{\ll}As{\gg}Init \bullet {\ll}P{\gg}}{\Gamma\ \vdash {\ll}P{\gg}[\ r{\ll}As{\gg}' := \varnothing\ ]}\ [\mathsf{assoc\text{-}init\text{-}d}]$$

$$\frac{\Gamma\ \vdash \exists\ [\![\ \mathbb{A}{\ll}As{\gg}Init\ ]\!] \bullet {\ll}P{\gg}}{\Gamma\ \vdash {\ll}P{\gg}[\ [\![\ r{\ll}As{\gg}' := \varnothing\ ]\!]]}\ [\mathsf{assoc\text{-}init\text{-}lsd}]$$

□

Template T48 (Add tuple operation)

These operations add a new tuple to the set of tuples of the association.

$$\begin{array}{|l}
\mathbb{A}_\Delta{\ll}As{\gg}Add \\
\hline
\Delta\mathbb{A}{\ll}As{\gg} \\
o{\ll}Cl_A{\gg}? : \mathbb{O}{\ll}Cl_A{\gg}Cl \\
o{\ll}Cl_B{\gg}? : \mathbb{O}{\ll}Cl_B{\gg}Cl \\
\hline
r{\ll}As{\gg}' = r{\ll}As{\gg} \cup \{o{\ll}Cl_A{\gg}? \mapsto o{\ll}Cl_B{\gg}?\} \\
\hline
\end{array}$$

The consistency conjecture is not required: it is always true (meta-theorem assoc-atop-epre, below).

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second gives true by construction on the consistency conjecture for the operation.

$$\frac{\text{pre}\,\mathbb{A}_\Delta{\ll}As{\gg}Add \bullet \text{true}}{[\ \mathbb{A}{\ll}As{\gg};\ o{\ll}Cl_A{\gg}? : \mathbb{O}{\ll}Cl_A{\gg}Cl;\ o{\ll}Cl_B{\gg}? : \mathbb{O}{\ll}Cl_B{\gg}Cl\ ]} \quad [\text{assoc-atop-pre}]$$

$$\frac{\Gamma \vdash \exists\ \text{pre}\,\mathbb{A}_\Delta{\ll}As{\gg}Add \bullet \text{true}}{\text{true}} \quad [\text{assoc-atop-epre}]$$

Meta-lemmas.

$$\frac{\text{pre}(\mathbb{A}_\Delta{\ll}As{\gg}Add \wedge {\ll}Sc{\gg})}{\begin{array}{l}\text{pre}\,\mathbb{A}_\Delta{\ll}As{\gg}Add \\ \quad \wedge\ \text{pre}\ {\ll}Sc{\gg}[\ r{\ll}As{\gg}' := r{\ll}As{\gg} \cup \{o{\ll}Cl_A{\gg}? \mapsto o{\ll}Cl_B{\gg}?\}\ ]\end{array}} \quad [\text{assoc-atop-pre-d}]$$

□

Template T49 (Domain subtraction operation)

These operations remove all tuples from the association-set, given a set of objects (received as input) from the domain of the association.

$$\begin{array}{|l}
\mathbb{A}_\Delta{\ll}As{\gg}DelD \\
\hline
\Delta\mathbb{A}{\ll}As{\gg} \\
os{\ll}Cl_A{\gg}? : \mathbb{P}(\mathbb{O}{\ll}Cl_A{\gg}Cl) \\
\hline
r{\ll}As{\gg}' = os{\ll}Cl_A{\gg}? \lhd r{\ll}As{\gg} \\
\hline
\end{array}$$

The consistency conjecture is not required (meta-theorem assoc-dsop-epre, below).

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second gives *true by construction* on the consistency conjecture.

$$\frac{\text{pre}\ \mathbb{A}_\Delta{\ll}As{\gg}DelD \bullet \text{true}}{[\ \mathbb{A}{\ll}As{\gg};\ os{\ll}Cl_A{\gg}? : \mathbb{P}(\mathbb{O}{\ll}Cl_A{\gg}Cl)\ ]} \quad [\text{assoc-dsop-pre}]$$

$$\frac{\Gamma \vdash \exists\ \text{pre}\ \mathbb{A}_\Delta{\ll}As{\gg}DelD \bullet \text{true}}{\text{true}} \quad [\text{assoc-dsop-epre}]$$

□

Template T50 (Range subtraction operation)

These operations remove all tuples from the association-set, given a set of objects (received as input) from the range of the association.

$\mathbb{A}_\Delta{\ll}As{\gg}DelR$

$\Delta\mathbb{A}{\ll}As{\gg}$
$os{\ll}Cl_B{\gg}? : \mathbb{P}(\mathbb{O}{\ll}Cl_B{\gg}Cl)$

$r{\ll}As{\gg}' = r{\ll}As{\gg} \rhd os{\ll}Cl_B{\gg}?$

The consistency conjecture is not required because it is always true (meta-theorem assoc-rsop-epre, below).

Meta-theorems. The first meta-theorem gives the precondition of the operation. The second gives *true by construction* on the consistency conjecture.

$$\frac{\Gamma \vdash \exists\ \text{pre}\ \mathbb{A}_\Delta{\ll}As{\gg}DelR \bullet \text{true}}{[\ \mathbb{A}{\ll}As{\gg};\ os{\ll}Cl_B{\gg}? : \mathbb{P}(\mathbb{O}{\ll}Cl_B{\gg}Cl)\ ]} \quad [\text{assoc-rsop-pre}]$$

$$\frac{\Gamma \vdash \exists\ \text{pre}\ \mathbb{A}_\Delta{\ll}As{\gg}DelR \bullet \text{true}}{\text{true}} \quad [\text{assoc-rsop-epre}]$$

□

Template T51 (Association Query Forward)

These operations give the objects linked to some object (received as input) from the domain of the association; this is given by taking the relational image of the relation:

$\mathbb{A}_\Xi{\ll}As{\gg}QF$

$\Xi\mathbb{A}{\ll}As{\gg}$
$o{\ll}Cl_A{\gg}? : \mathbb{O}{\ll}Cl_A{\gg}Cl$
$os{\ll}Cl_B{\gg}! : \mathbb{P}(\mathbb{O}{\ll}Cl_B{\gg}Cl)$

$os{\ll}Cl_B{\gg}! = r{\ll}As{\gg}(\!|\ \{o{\ll}Cl_A{\gg}?\}\ |\!)$

The initialisation conjecture is not required because it is always true (meta-theorem assoc-qfop-epre, below).

Meta-theorems. The meta-theorem gives *true by construction* on the operation's consistency conjecture.

$$\frac{\Gamma \vdash \exists \ \mathrm{pre}\, \mathbb{A}_{\Xi} \ll As \gg QF \bullet \mathrm{true}}{\mathrm{true}} \quad [\mathsf{assoc\text{-}qfop\text{-}epre}]$$

□

Template T52 (Association Query Backwards)

These operations give the objects linked to some object (received as input) from the range of the association; this is given by taking the relational image of the inverse relation:

$\mathbb{A}_{\Xi} \ll As \gg QB$

$\Xi \mathbb{A} \ll As \gg$

$o \ll Cl_B \gg ? : \mathbb{O} \ll Cl_B \gg Cl$

$os \ll Cl_A \gg ! : \mathbb{P}(\mathbb{O} \ll Cl_A \gg Cl)$

$os \ll Cl_A \gg ! = r \ll As \gg^{\sim} (\!|\ \{ o \ll Cl_B \gg ? \}\ |\!)$

The initialisation conjecture is not required because it is always true (meta-theorem assoc-qbop-epre below).

Meta-theorems. The meta-theorem gives *true by construction* on the operation's consistency conjecture.

$$\frac{\Gamma \vdash \exists \ \mathrm{pre}\, \mathbb{A}_{\Xi} \ll As \gg QB \bullet \mathrm{true}}{\mathrm{true}} \quad [\mathsf{assoc\text{-}qbop\text{-}epre}]$$

□

D.4 Global View

The global view defines structures that are ensembles of classes and associations. The templates of this view are as follows.

Template T53 (Global constraint)

In the global view, we need to express constraints that can only be expressed in the scope of the ensemble (they are global). These are expressed in separate schemas, following the Name Predicates pattern [SPT03b], and then conjoined with the association linking schemas to make the system constraints schema (*SysConst*, below). The global constraints schema includes the components affected by the constraint and a predicate expressing the constraint.

$Const \ll SConst \gg$

$[\![\ \mathbb{S} \ll ConstCl \gg ;\ ]\!]\ [\![\ \mathbb{A} \ll ConstAs \gg\ ]\!]$

$\ll Const \gg$

□

Template T54 (State space, initialisation and constraints)

There is a schema, *SysConst*, which expresses all the constraints of the system. This includes association link schemas, subclassing link schemas and global constraints. This is then added as a predicate to the system schema.

$$SysConst == [\![\ Link\mathbb{A}{\ll}As{\gg}\ ]\!] \land [\![\ \mathbb{S}{\ll}ChCl{\gg}Is{\ll}PCl{\gg}\ ]\!] \land [\![\ Const{\ll}SConst{\gg}\ ]\!]$$

The system state space includes the system's components (classes and associations) and the system constraints schema. The system initialisation is defined as the initialisation of the system's components. The user is then required to prove the initialisation conjecture.

System

$[\![\ \mathbb{S}{\ll}Cl{\gg}\ ]\!][\![\ ;\ \ \mathbb{A}{\ll}As{\gg}\ ]\!]$

$SysConst$

$$SysInit == System\,' \land [\![\ \mathbb{S}{\ll}Cl{\gg}Init\ ]\!] \land [\![\ \mathbb{A}{\ll}As{\gg}Init\ ]\!]$$
$$\vdash?\ \ \exists\ SysInit \bullet \text{true}$$

Meta-theorems. There are two meta-theorem for the system initialisation conjecture. The first is the more general; it reduces the conjecture to a proof that the global constraints hold in the initial state of the system. Note that it is not required to prove that the association link invariant holds in this state because the meta-theorem demonstrates that it is so (this is a consequence of a law of the mult generic, which says that the empty initialisation of the relation and extensions is valid for any kind of multiplicity constraint, see appendix C).

$$\frac{\Gamma\ \vdash \exists\ SysInit \bullet \text{true}}{\Gamma\ \vdash [\![\ {\ll}SConst{\gg}'\ ]\!][\ [\![\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ]\!][\![\ , r{\ll}As{\gg}' := \varnothing\ ]\!]\ ]} \quad [\text{sys-init}]$$

The second meta-theorem gives *true by construction* on the initialisation conjecture, when the system has no global constraints.

$$\frac{\Gamma;\ [\![\ {\ll}SConst{\gg}\ ]\!] \vdash \exists\ SysInit \bullet \text{true}}{\text{true}} \quad [\text{sys-init-ni}]$$

□

Template T55 (Operation Constraint)

In some cases, an extra constraint, which is not expressible in the scope of the local components, needs to be added to a system operation. Such constraints are expressed in separate schemas which are then used to compose the system operation. Operation constraint schemas indicate the components (class extensions and associations) and the inputs and outputs of the operation that are effected by the constraint; the actual constraint is expressed as a predicate.

$$
\begin{array}{|l}
\hline
OpConst\langle\!\langle opConst\rangle\!\rangle \\
\hline
[\![\ \mathbb{S}\langle\!\langle cCl\rangle\!\rangle\ ]\!] \\
[\![\ \mathbb{A}\langle\!\langle cAs\rangle\!\rangle\ ]\!] \\
[\![\ \langle\!\langle cin\rangle\!\rangle? : \langle\!\langle cIT\rangle\!\rangle\ ]\!] \\
[\![\ \langle\!\langle cout\rangle\!\rangle! : \langle\!\langle cOT\rangle\!\rangle\ ]\!] \\
\hline
\langle\!\langle opCP\rangle\!\rangle \\
\hline
\end{array}
$$

□

Template T56 (Operation Connector, singleton set)

In some cases, an adjustment in the communication between components is required. In ZOO, such adjustments are described in a connector schema. The following connector template expresses a common type of connector: one that transforms one input to a singleton set of inputs. This is used to compose pairs of operations, when one operation expects a single object and the other expects a set of objects.

$$
\begin{array}{|l}
\hline
ConnSing\langle\!\langle cCl\rangle\!\rangle \\
\hline
os\langle\!\langle cCl\rangle\!\rangle? : \mathbb{P}(\mathbb{O}\langle\!\langle cCl\rangle\!\rangle Cl) \\
o\langle\!\langle cCl\rangle\!\rangle? : \mathbb{O}\langle\!\langle cCl\rangle\!\rangle Cl \\
\hline
os\langle\!\langle cCl\rangle\!\rangle? = \{o\langle\!\langle cCl\rangle\!\rangle?\} \\
\hline
\end{array}
$$

□

Template T57 (System operation frame)

System operation frames indicate the local components (classes or associations) that are allowed to change and the ones that must remain unchanged as a result of the operation. They are formed by conjoining $\Delta System$ (everything changes) with the Ξ of each component that must not change; the components that are allowed to change are the ones that are part of the system but that are not mentioned in the frame.

$$\Psi\langle\!\langle sOp\rangle\!\rangle == \Delta System[\![\ \wedge \Xi\mathbb{S}\langle\!\langle fCl\rangle\!\rangle\ ]\!][\![\ \wedge \Xi\mathbb{A}\langle\!\langle fAs\rangle\!\rangle\ ]\!]$$

□

Template T58 (New object system operation)

These operations create one object of a specific class. They are formed by conjoining the frame, zero or more operation constraints and the new class operation.

$$Sys\langle\!\langle sOp\rangle\!\rangle == \Psi\langle\!\langle sOp\rangle\!\rangle \wedge [\![\ OpConst\langle\!\langle opConst\rangle\!\rangle\ ]\!] \wedge \mathbb{S}_\Delta\langle\!\langle nCl\rangle\!\rangle New$$

There are two proof obligations. The first is a well-formedness condition for the operation's frame, which says that the components that do not change must include all components except the new class. The second is the usual precondition consistency conjecture.

$$\vdash? \{[\![\ \langle\!\langle Cl\rangle\!\rangle\ ]\!]\} = \{[\![\ \langle\!\langle nCl\rangle\!\rangle\ ]\!][\![\ , \langle\!\langle fCl\rangle\!\rangle\ ]\!]\} \wedge \{[\![\ \langle\!\langle As\rangle\!\rangle\ ]\!]\} = \{[\![\ \langle\!\langle fAs\rangle\!\rangle\ ]\!]\}$$

$$\vdash? \exists\ \text{pre}\ Sys\langle\!\langle sOp\rangle\!\rangle \bullet \text{true}$$

Meta-theorems. The meta-theorem simplifies the operation's precondition. It says that precondition is the conjunction of the system, the precondition of the new class operation, and a predicate that results from the substitutions defined by the operation for the system and operation constraints.

$$\frac{\begin{array}{l}\mathrm{pre}(\Psi{\ll}sOp{\gg} \wedge [\![\ OpConst{\ll}opConst{\gg}\ ]\!] \wedge \mathbb{S}_\Delta{\ll}nCl{\gg}New \\ \quad \wedge\ \mathbb{A}_\Delta{\ll}aAs{\gg}Add[o{\ll}nCl{\gg}!/o{\ll}nCl{\gg}?])\end{array}}{\begin{array}{l}System \wedge \mathrm{pre}\,\mathbb{S}_\Delta{\ll}nCl{\gg}New \\ \quad \wedge \exists\, o{\ll}nCl{\gg}! : \mathbb{O}{\ll}nCl{\gg}Cl;\ {\ll}nCl{\gg}\,' \bullet (SysConst' \\ \quad\quad \wedge [\![\ OpConst{\ll}opConst{\gg}\ ]\!])[\ [\![\ \theta\mathbb{S}{\ll}fCl{\gg}\,' := \theta\mathbb{S}{\ll}fCl{\gg}\ ]\!] \\ \quad\quad\quad [\![\ , \theta\mathbb{A}{\ll}fAs{\gg}\,' := \theta\mathbb{A}{\ll}fAs{\gg}\ ]\!], \\ \quad\quad\quad s{\ll}nCl{\gg}' := s{\ll}nCl{\gg} \cup \{o{\ll}nCl{\gg}!\}, \\ \quad\quad\quad st{\ll}nCl{\gg}' := st{\ll}nCl{\gg} \cup \{o{\ll}nCl{\gg}! \mapsto \theta{\ll}nCl{\gg}\,'\}\ ] \\ \quad\quad \wedge\ {\ll}nCl{\gg}Init\end{array}} \quad \text{[sys-op-no-pre]}$$

□

Template T59 (New object and add tuple system operation)

These operations create one object of a specific class and then add a link, with the new object and another object (input), to an association. These operations are formed by conjoining the operation's frame, zero or more operation constraints, the new class operation and the add tuple operation of the association.

$$\begin{array}{l}Sys{\ll}sOp{\gg} == \Psi{\ll}sOp{\gg} \wedge [\![\ OpConst{\ll}opConst{\gg}\ ]\!] \wedge \mathbb{S}_\Delta{\ll}nCl{\gg}New \\ \quad \mathbb{A}_\Delta{\ll}aAs{\gg}Add[o{\ll}nCl{\gg}!/o{\ll}nCl{\gg}?]\end{array}$$

There are two proof obligations. The first is a well formedness condition for the operation's frame, which says that the components that do not change must include all components except the class of the new object and the association to which the new object is added as part of a link. The second is the usual precondition consistency conjecture.

$$\vdash?\ \{[\![\ {\ll}Cl{\gg}\ ]\!]\} = \{{\ll}nCl{\gg}[\![\ , {\ll}fCl{\gg}\ ]\!]\} \wedge \{[\![\ {\ll}As{\gg}\ ]\!]\} = \{{\ll}aAs{\gg}[\![\ , {\ll}fAs{\gg}\ ]\!]\}$$

$$\vdash?\ \exists\ \ \mathrm{pre}\ \ Sys{\ll}sOp{\gg} \bullet \mathrm{true}$$

Meta-theorems The meta-theorem simplifies the operation's precondition. It says that the precondition is the conjunction of the system, the precondition of the new class operation, and the predicate that results from the substitutions defined by the operation for the system and operation constraints.

$$\frac{\begin{array}{l}\mathrm{pre}(\Psi{\ll}sOp{\gg} \wedge [\![\ OpConst{\ll}opConst{\gg}\ ]\!] \wedge \mathbb{S}_\Delta{\ll}nCl{\gg}New \\ \quad \wedge\ \mathbb{A}_\Delta{\ll}aAs{\gg}Add[o{\ll}nCl{\gg}!/o{\ll}nCl{\gg}?])\end{array}}{\begin{array}{l}System \wedge \mathrm{pre}\,\mathbb{S}_\Delta{\ll}nCl{\gg}New \\ \quad \wedge \exists\, o{\ll}nCl{\gg}! : \mathbb{O}{\ll}nCl{\gg}Cl;\ {\ll}nCl{\gg}\,';\ \mathbb{A}{\ll}aAs{\gg}\,' \bullet \\ \quad\quad (SysConst' \\ \quad\quad \wedge [\![\ OpConst{\ll}opConst{\gg}\ ]\!])[\ [\![\ \theta\mathbb{S}{\ll}fCl{\gg}\,' := \theta\mathbb{S}{\ll}fCl{\gg}\ ]\!] \\ \quad\quad\quad [\![\ , \theta\mathbb{A}{\ll}fAs{\gg}\,' := \theta\mathbb{A}{\ll}fAs{\gg}\ ]\!], \\ \quad\quad\quad s{\ll}nCl{\gg}' := s{\ll}nCl{\gg} \cup \{o{\ll}nCl{\gg}!\}, \\ \quad\quad\quad st{\ll}nCl{\gg}' := st{\ll}nCl{\gg} \cup \{o{\ll}nCl{\gg}! \mapsto \theta{\ll}nCl{\gg}\,'\}\ ] \\ \quad\quad \wedge\ {\ll}nCl{\gg}Init \wedge \mathbb{A}_\Delta{\ll}aAs{\gg}Add[o{\ll}nCl{\gg}!/o{\ll}nCl{\gg}?]\end{array}} \quad \text{[sys-op-no-at-pre]}$$

□

Template T60 (Update object operation)

These operations update the state of an existing class object. They comprise the operation's frame, zero or more operation constraints, and the update class operation.

$$Sys{\ll}sOp{\gg} == \Psi{\ll}sOp{\gg} \land [\![\ OpConst{\ll}opConst{\gg}\ ]\!] \land \mathbb{S}_\Delta{\ll}uCl{\gg}{\ll}uOp{\gg}$$

There are two proof obligations. The first is a well-formedness condition upon the operation's frame, which says that the components that do not change must include all components except the class that is to change. The second is the usual operation consistency conjecture.

$$\vdash? \{[\![\ll Cl\gg]\!]\} = \{\ll uCl\gg[\![, \ll fCl\gg]\!]\} \land \{[\![\ll As\gg]\!]\} = \{[\![\ll fAs\gg]\!]\}$$
$$\vdash? \exists \ \text{pre}\ Sys{\ll}sOp{\gg} \bullet \text{true}$$

Meta-theorems. There is one meta-theorem to simplify the operation's precondition. It says that precondition is the conjunction of the system, the precondition of the update class operation, and the predicate resulting from the substitutions defined by the operation for the system and operation constraints.

$$\frac{\text{pre}(\Psi{\ll}sOp{\gg} \land [\![\ OpConst{\ll}opConst{\gg}\ ]\!] \land \mathbb{S}_\Delta{\ll}uCl{\gg}{\ll}uOp{\gg})}{\begin{array}{l} System \land \text{pre}\,\mathbb{S}_\Delta{\ll}uCl{\gg}{\ll}uOp{\gg} \\ \quad \land \exists{\ll}uCl{\gg}\,' \bullet (SysConst' \\ \qquad \land [\![\ OpConst{\ll}opConst{\gg}\ ]\!])[\,[\![\ \theta\mathbb{S}{\ll}fCl{\gg}\,' := \theta\mathbb{S}{\ll}fCl{\gg}\ ]\!] \\ \qquad\quad [\![\ , \theta\mathbb{A}{\ll}fAs{\gg}\,' := \theta\mathbb{A}{\ll}fAs{\gg}\ ]\!], \\ \qquad\quad s{\ll}uCl{\gg}' := s{\ll}uCl{\gg}, \\ \qquad\quad st{\ll}uCl{\gg}' := st{\ll}uCl{\gg} \oplus \{o{\ll}uCl{\gg}? \mapsto \theta{\ll}uCl{\gg}\,'\}\ ] \\ \qquad \land {\ll}uCl{\gg}_\Delta{\ll}uOp{\gg}[\theta{\ll}uCl{\gg} := st{\ll}uCl{\gg}\ o{\ll}uCl{\gg}?] \end{array}} \quad [\text{sys-op-uo-pre}]$$

□

Template T61 (Delete object from domain system operation)

These operations delete an object of a class and all the links that refer to the object in an association; the deletion is based on the domain of the association. The operation is formed by conjoining the operation's frame, zero or more operation constraints, the delete class operation, and the operation to delete from the domain of the association.

$$\begin{array}{l} Sys{\ll}sOp{\gg} == (\Psi{\ll}sOp{\gg} \land [\![\ OpConst{\ll}opConst{\gg}\ ]\!]) \land \mathbb{S}_\Delta{\ll}dCl{\gg}Delete \\ \quad \land\ ConnSing{\ll}cCl{\gg} \land \mathbb{A}_\Delta{\ll}dAs{\gg}DelD) \setminus (os{\ll}dCl{\gg}?) \end{array}$$

There are two proof obligations. The first is a well-formedness condition for the operation's frame, which says that the components that do not change must include all components except the class from which the object is to be deleted and the association from which the link is deleted. The second is the usual precondition consistency conjecture.

$$\vdash? \{[\![\ll Cl\gg]\!]\} = \{\ll dCl\gg[\![, \ll fCl\gg]\!]\} \land \{[\![\ll As\gg]\!]\} = \{\ll dAs\gg[\![, \ll fAs\gg]\!]\}$$
$$\vdash? \exists \ \text{pre}\ Sys{\ll}sOp{\gg} \bullet \text{true}$$

Meta-theorems. The meta-theorem simplifies the operation's precondition. It says that the precondition is the conjunction of the system, the precondition of the delete class operation, and the predicate that results from the substitutions defined by the operation for the system and operation constraints.

$$\frac{\begin{array}{l}\mathrm{pre}((\Psi\llcorner sOp\lrcorner \wedge [\![\ OpConst\llcorner opConst\lrcorner\ ]\!] \wedge \mathbb{S}_\Delta\llcorner dCl\lrcorner Delete \\ \quad \wedge\ ConnSing\llcorner cCl\lrcorner \wedge \mathbb{A}_\Delta\llcorner dAs\lrcorner DelD) \setminus (os\llcorner dCl\lrcorner ?))\end{array}}{\begin{array}{l}System \wedge \mathrm{pre}\,\mathbb{S}_\Delta\llcorner dCl\lrcorner Delete \wedge (SysConst' \\ \quad \wedge [\![\ OpConst\llcorner opConst\lrcorner\ ]\!])[\ [\![\ \theta\mathbb{S}\llcorner fCl\lrcorner\ ' := \theta\mathbb{S}\llcorner fCl\lrcorner\ ]\!] \\ \qquad [\![\ , \theta\mathbb{A}\llcorner fAs\lrcorner\ ' := \theta\mathbb{A}\llcorner fAs\lrcorner\ ]\!], \\ \qquad s\llcorner dCl\lrcorner' := s\llcorner dCl\lrcorner \setminus \{o\llcorner dCl\lrcorner ?\}, \\ \qquad st\llcorner dCl\lrcorner' := \{o\llcorner dCl\lrcorner ?\} \lhd st\llcorner dCl\lrcorner, \\ \qquad r\llcorner dAs\lrcorner' := \{o\llcorner dCl\lrcorner ?\} \lhd r\llcorner dAs\lrcorner]\end{array}}\ [\text{sys-op-dod-pre}]$$

□

Template T62 (Delete object from range system operation)

These operations delete an object of a class and all the links that refer to the object in an association; the deletion is based on the range of the association. The operation is formed by conjoining the operation's frame, zero or more operation constraints, the delete class operation, and the operation to delete from the range of the association.

$$\begin{array}{l}Sys\llcorner sOp\lrcorner == (\Psi\llcorner sOp\lrcorner \wedge [\![\ OpConst\llcorner opConst\lrcorner\ ]\!]) \wedge \mathbb{S}_\Delta\llcorner dCl\lrcorner Delete \\ \quad \wedge\ ConnSing\llcorner dCl\lrcorner \wedge \mathbb{A}_\Delta\llcorner dAs\lrcorner DelR) \setminus (os\llcorner dCl\lrcorner ?)\end{array}$$

There are two proof obligations. The first is a well-formedness condition for the operation's frame, which says that the components that do not change must include all components except the class from which the object is to be deleted and the association from which the objects is to be deleted as part of a link. The second is the usual operation consistency conjecture.

$$\vdash ?\ \{[\![\ \llcorner Cl\lrcorner\ ]\!]\} = \{\llcorner dCl\lrcorner [\![\ , \llcorner fCl\lrcorner\ ]\!]\} \wedge \{[\![\ \llcorner As\lrcorner\ ]\!]\} = \{\llcorner dAs\lrcorner [\![\ , \llcorner fAs\lrcorner\ ]\!]\}$$
$$\vdash ?\ \exists\ \mathrm{pre}\ Sys\llcorner sOp\lrcorner \bullet \mathrm{true}$$

Meta-theorems. The meta-theorem simplifies the operation's precondition. It says that the precondition is the conjunction of the system, the precondition of the delete class operation, and the predicate that results from the substitutions defined by the operation for the system and operation constraints.

$$\frac{\begin{array}{l}\mathrm{pre}((\Psi\llcorner sOp\lrcorner \wedge [\![\ OpConst\llcorner opConst\lrcorner\ ]\!] \wedge \mathbb{S}_\Delta\llcorner dCl\lrcorner Delete \\ \quad \wedge\ ConnSing\llcorner cCl\lrcorner \wedge \mathbb{A}_\Delta\llcorner dAs\lrcorner DelR) \setminus (os\llcorner dCl\lrcorner ?))\end{array}}{\begin{array}{l}System \wedge \mathrm{pre}\,\mathbb{S}_\Delta\llcorner dCl\lrcorner Delete \wedge (SysConst' \\ \quad \wedge [\![\ OpConst\llcorner opConst\lrcorner\ ]\!])[\ [\![\ \theta\mathbb{S}\llcorner fCl\lrcorner\ ' := \theta\mathbb{S}\llcorner fCl\lrcorner\ ]\!] \\ \qquad [\![\ , \theta\mathbb{A}\llcorner fAs\lrcorner\ ' := \theta\mathbb{A}\llcorner fAs\lrcorner\ ]\!], \\ \qquad s\llcorner dCl\lrcorner' := s\llcorner dCl\lrcorner \setminus \{o\llcorner dCl\lrcorner ?\}, \\ \qquad st\llcorner dCl\lrcorner' := \{o\llcorner dCl\lrcorner ?\} \lhd st\llcorner dCl\lrcorner, \\ \qquad r\llcorner dAs\lrcorner' := r\llcorner dAs\lrcorner \rhd \{o\llcorner dCl\lrcorner ?\}]\end{array}}\ [\text{sys-op-dor-pre}]$$

□

Template T63 (Observe class system operation)

These operations observe the state of a class, which may involve an observation upon an individual object of the class or upon the set of objects as a whole. The operation is formed by conjoining the Ξ of the system and the observe class operation.

$$Sys \ll sOp \gg == \Xi System \land \mathbb{S}_{\Xi} \ll oCl \gg \ll oOp \gg$$

The usual consistency conjecture is not required, because by meta-theorem sys-op-oc-epre (below) it is always true.

Meta-theorems. There are two meta-theorems. The first gives the precondition of the operation. The second gives *true by construction* on the precondition consistency conjecture.

$$\frac{\text{pre } (\Xi System \land \mathbb{S}_{\Xi} \ll oCl \gg \ll Op \gg)}{System \land \text{pre}\, \mathbb{S}_{\Xi} \ll oCl \gg \ll oOp \gg} \quad [\text{sys-op-oc-pre}]$$

$$\frac{\Gamma \vdash \exists \text{ pre } (\Xi System \land \mathbb{S}_{\Xi} \ll oCl \gg \ll Op \gg) \bullet \text{true}}{\text{true}} \quad [\text{sys-op-oc-epre}]$$

□

Template T64 (Observe association system operation)

These operations observe the state of an association. The operation is formed by conjoining the Ξ of the system, and the observe association operation.

$$Sys \ll sOp \gg == \Xi System \land \mathbb{A}_{\Xi} \ll oAs \gg \ll oOp \gg$$

The usual consistency conjecture is not required, because by meta-theorem sys-op-oa-epre (below) it is always true.

Meta-theorems. There are two meta-theorems. The first gives the precondition of the operation. The second gives *true by construction* on the consistency conjecture.

$$\frac{\text{pre } (\Psi \ll sOp \gg \land \mathbb{A}_{\Xi} \ll oAs \gg \ll oOp \gg)}{System \land \text{pre}\, \mathbb{A}_{\Xi} \ll oAs \gg \ll oOp \gg} \quad [\text{sys-op-oa-pre}]$$

$$\frac{\Gamma \vdash \exists \text{pre } (\Psi \ll sOp \gg \land \mathbb{A}_{\Xi} \ll oAs \gg \ll oOp \gg)}{\text{true}} \quad [\text{sys-op-oa-epre}]$$

□

D.5 Snapshots

Template T65 (Snapshot)

This template generates the ZOO representation of single snapshots and the required validity conjectures. The first Z paragraph defines objects and their internal states:

$$
\begin{array}{l}
[\![\ll atv \gg : \ll atT \gg \ ]\!] \\
[\![\ [\![\ll o \gg \ ]\!] : \mathbb{O}_x \ll SnCl \gg Cl \ ]\!] \\
[\![\ [\![\ll o \gg St \ ]\!] : \ll SnCl \gg \ ]\!] \\
\hline
[\![\ \mathbb{O}_x \ll SnCl \gg Cl = \{ [\![\ll o \gg \ ]\!] \} \land \# \mathbb{O}_x \ll SnCl \gg Cl = \# \langle [\![\ll o \gg \ ]\!] \rangle \ ]\!] \\
[\![\ll o \gg St = \langle\!| \ [\![\ll at \gg == \ll av \gg \ ]\!] \ |\!\rangle \ ]\!]
\end{array}
$$

The next paragraph defines the instance of the modelled system being captured by the snapshot. This includes the extensions of the classes mentioned in the snapshot, made up of the existing set of objects and the object to state mappings, and the associations, made up of the object tuples; all sets corresponding to class and associations not mentioned in the snapshot should be empty:

$$
\begin{array}{l}
StSnap \ll Sn \gg \\
System \\
\hline
[\![\ s \ll SnCl \gg = \{ [\![\ll o \gg \ ]\!] \} \ ]\!] \\
[\![\ st \ll SnCl \gg = \{ [\![\ll o \gg \mapsto \ll o \gg St \ ]\!] \} \ ]\!] \\
[\![\ s \ll PCl \gg = \{ [\![\ll o \gg \ ]\!] \} [\![\ \cup s \ll ChCl \gg \ ]\!] \ ]\!] \\
[\![\ st \ll PCl \gg = \{ [\![\ll o \gg \mapsto \ll o \gg St \ ]\!] \} [\![\ \cup st \ll ChCl \gg \fatsemi (\lambda \ll ChCl \gg \bullet \ll PCl \gg) \ ]\!] \ ]\!] \\
[\![\ s \ll OCl \gg = \varnothing \land st \ll OCl \gg = \varnothing \ ]\!] \\
[\![\ r \ll SnA \gg = \{ [\![\ll oR_1 \gg \mapsto \ll oR_2 \gg \ ]\!] \} \ ]\!] \\
[\![\ r \ll OA \gg = \varnothing \ ]\!]
\end{array}
$$

Finally, the conjecture requires an existence proof or its negation:

$$\vdash ? \ (\!| \ \neg \ |\!)^? (\exists \, StSnap \ll Sn \gg \bullet \text{true})$$

□

Template T66 (Snapshot-Pair)

This template generates the ZOO representation of snapshot-pairs and required conjectures. The first Z paragraph defines objects and their internal states:

$$
\begin{array}{l}
[\![\ll atv \gg : \ll atT \gg \ ]\!] \\
[\![\ [\![\ll o \gg \ ]\!] : \mathbb{O}_x \ll SnCl \gg Cl \ ]\!] \\
[\![\ [\![\ll o \gg St_B, \ll o \gg St_A \ ]\!] : \ll SnCl \gg \ ]\!] \\
\hline
[\![\ \mathbb{O}_x \ll SnCl \gg Cl = \{ [\![\ll o \gg \ ]\!] \} \land \# \mathbb{O}_x \ll SnCl \gg Cl = \# \langle [\![\ll o \gg \ ]\!] \rangle \ ]\!] \\
[\![\ll o \gg St_B = \langle\!| \ [\![\ll at \gg == \ll bav \gg \ ]\!] \ |\!\rangle \ ]\!] \\
[\![\ll o \gg St_A = \langle\!| \ [\![\ll at \gg == \ll aav \gg \ ]\!] \ |\!\rangle \ ]\!] \\
[\![\ll Consts \gg \ ]\!]
\end{array}
$$

The next paragraphs define the before-instance, the inputs to the operation, and the after instance of the modelled system.

$$
\begin{array}{|l}
\hline
BOpSnap{\ll}Sn{\gg} \\
System \\
\hline
[\![\ s{\ll}SnCl_B{\gg} = \{[\![\ {\ll}o_B{\gg}\ ]\!]\}\ ]\!] \\
[\![\ st{\ll}SnCl_B{\gg} = \{[\![\ {\ll}o_B{\gg} \mapsto {\ll}o_B{\gg}St\ ]\!]\}\ ]\!] \\
[\![\ s{\ll}PCl_B{\gg} = \{[\![\ {\ll}o_B{\gg}\ ]\!]\}[\![\ \cup s{\ll}ChCl_B{\gg}\ ]\!]\ ]\!] \\
[\![\ st{\ll}PCl_B{\gg} = \{[\![\ {\ll}o_B{\gg} \mapsto {\ll}o_B{\gg}St\ ]\!]\}[\![\ \cup st{\ll}ChCl_B{\gg} \fatsemi (\lambda{\ll}ChCl_B{\gg} \bullet {\ll}PCl_B{\gg})\ ]\!]\ ]\!] \\
[\![\ s{\ll}OCl_B{\gg} = \varnothing \land st{\ll}OCl_B{\gg} = \varnothing\ ]\!] \\
[\![\ r{\ll}SnA_B{\gg} = \{[\![\ {\ll}oR_{1B}{\gg} \mapsto {\ll}oR_{2B}{\gg}\ ]\!]\}\ ]\!] \\
[\![\ r{\ll}OA_B{\gg} = \varnothing\ ]\!] \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
IOpSnap{\ll}Sn{\gg} \\
[\![\ {\ll}in{\gg}? : {\ll}iT{\gg}\ ]\!] \\
\hline
[\![\ {\ll}in{\gg}? = {\ll}iv{\gg}\ ]\!] \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
AOpSnap{\ll}Sn{\gg} \\
System \\
\hline
[\![\ s{\ll}SnCl_A{\gg} = \{[\![\ {\ll}o_A{\gg}\ ]\!]\}\ ]\!] \\
[\![\ st{\ll}SnCl_A{\gg} = \{[\![\ {\ll}o_A{\gg} \mapsto {\ll}o_A{\gg}St\ ]\!]\}\ ]\!] \\
[\![\ s{\ll}PCl_A{\gg} = \{[\![\ {\ll}o_A{\gg}\ ]\!]\}[\![\ \cup s{\ll}ChCl_A{\gg}\ ]\!]\ ]\!] \\
[\![\ st{\ll}PCl_A{\gg} = \{[\![\ {\ll}o_A{\gg} \mapsto {\ll}o_A{\gg}St\ ]\!]\}[\![\ \cup st{\ll}ChCl_A{\gg} \fatsemi (\lambda{\ll}ChCl_A{\gg} \bullet {\ll}PCl_A{\gg})\ ]\!]\ ]\!] \\
[\![\ s{\ll}OCl_A{\gg} = \varnothing \land st{\ll}OCl_A{\gg} = \varnothing\ ]\!] \\
[\![\ r{\ll}SnA_A{\gg} = \{[\![\ {\ll}oR_{1A}{\gg} \mapsto {\ll}oR_{2A}{\gg}\ ]\!]\}\ ]\!] \\
[\![\ r{\ll}OA_A{\gg} = \varnothing\ ]\!] \\
\hline
\end{array}
$$

Finally, there are two conjectures, which may be negated (optional choice). The first checks that there is a valid instance satisfying the operation's precondition corresponding to the before snapshot (or the negation of this). The second checks that there is a pair of valid instances satisfying the constraints of the operation corresponding to the snapshot-pair (or the negation of this):

$$\vdash? \ (\!|\ \neg\ |\!)^?(\exists \operatorname{pre} Sys{\ll}Op{\gg} \bullet BOpSnap{\ll}Sn{\gg} \land IOpSnap{\ll}Sn{\gg})$$
$$\vdash? \ (\!|\ \neg\ |\!)^?(\exists\ Sys{\ll}Op{\gg} \bullet BOpSnap{\ll}Sn{\gg} \land IOpSnap{\ll}Sn{\gg} \land AOpSnap{\ll}Sn{\gg}\ ')$$

□

Template T67 (Snapshot-sequence)

This template generates ZOO representations of snapshot-pairs and required conjectures. The first Z paragraph defines objects and their internal states (a function from object to a sequence of states):

$$
\begin{array}{l}
[\![\ll atv\gg : \ll atT\gg \ ]\!] \\
[\![\ [\![\ll o\gg \ ]\!] : \mathbb{O}_x\ll SnCl\gg Cl \ ]\!] \\
[\![\ oSt\ll SnCl\gg : \mathbb{O}_x\ll SnCl\gg Cl \nrightarrow \mathrm{seq}\ll SnCl\gg \ ]\!] \\
\hline
[\![\ \mathbb{O}_x\ll SnCl\gg Cl = \{[\![\ll o\gg \ ]\!]\} \land \#\mathbb{O}_x\ll SnCl\gg Cl = \#\langle [\![\ll o\gg \ ]\!] \rangle \ ]\!] \\
[\![\ oSt\ll SnCl\gg = \{[\![\ll o\gg \mapsto \langle [\![\ \langle\!| \ll at\gg == \ll av\gg \ |\!\rangle \ ]\!] \rangle \ ]\!]\} \ ]\!] \\
[\![\ \ll Consts\gg \ ]\!]
\end{array}
$$

The next paragraphs define the first snapshot instance, followed by a list of inputs, after snapshot instance, and conjectures. The conjectures are for the snapshot-pair.

OpSqSnap$\ll Sn\gg 1$

$$
\begin{array}{l}
[\![\ s\ll SnCl_S\gg = \{[\![\ll o_s\gg \ ]\!]\} \ ]\!] \\
[\![\ st\ll SnCl_S\gg = \{[\![\ll o_s\gg \mapsto oSt\ll SnCl_S\gg \ 1 \ ]\!]\} \ ]\!] \\
[\![\ s\ll PCl_S\gg = \{[\![\ll o_s\gg \ ]\!]\}[\![\ \cup s\ll ChCl_S\gg \ ]\!] \ ]\!] \\
[\![\ st\ll PCl_S\gg = \{[\![\ll o_s\gg \mapsto oSt\ll PCl_S\gg \ 1 \ ]\!]\}[\![\\
\quad \cup st\ll ChCl_S\gg \fatsemi (\lambda\ll ChCl_S\gg \bullet \ll PCl_S\gg) \ ]\!] \ ]\!] \\
[\![\ s\ll OCl_B\gg = \varnothing \land st\ll OCl_B\gg = \varnothing \ ]\!] \\
[\![\ r\ll SnA_B\gg = \{[\![\ll oR_{1B}\gg \mapsto \ll oR_{2B}\gg \ ]\!]\} \ ]\!] \\
[\![\ r\ll OA_B\gg = \varnothing \ ]\!]
\end{array}
$$

[\![

IOpSqSnap$\ll Sn\gg\ll N\gg$

$$
\begin{array}{l}
[\![\ \ll in\gg ? : \ll iT\gg \ ]\!] \\
\hline
[\![\ \ll in\gg ? = \ll iv\gg ? \ ]\!]
\end{array}
$$

OpSqSnap$\ll Sn\gg\ll N+1\gg$

$$
\begin{array}{l}
[\![\ s\ll SnCl_S\gg = \{[\![\ll o_s\gg \ ]\!]\} \ ]\!] \\
[\![\ st\ll SnCl_S\gg = \{[\![\ll o_s\gg \mapsto oSt\ll SnCl_S\gg \ \ll N\gg \ ]\!]\} \ ]\!] \\
[\![\ s\ll PCl_S\gg = \{[\![\ll o_s\gg \ ]\!]\}[\![\ \cup s\ll ChCl_S\gg \ ]\!] \ ]\!] \\
[\![\ st\ll PCl_S\gg = \{[\![\ll o_s\gg \mapsto oSt\ll PCl_S\gg \ \ll N\gg \ ]\!]\}[\![\\
\quad \cup st\ll ChCl_S\gg \fatsemi (\lambda\ll ChCl_S\gg \bullet \ll PCl_S\gg) \ ]\!] \ ]\!] \\
[\![\ s\ll OCl_B\gg = \varnothing \land st\ll OCl_B\gg = \varnothing \ ]\!] \\
[\![\ r\ll SnA_B\gg = \{[\![\ll oR_{1B}\gg \mapsto \ll oR_{2B}\gg \ ]\!]\} \ ]\!] \\
[\![\ r\ll OA_B\gg = \varnothing \ ]\!]
\end{array}
$$

$$
\begin{array}{l}
\vdash ?\ (\!|\ \neg\ |\!)^?(\exists \mathrm{pre}\, Sys\ll Op\gg \bullet OpSqSnap\ll Sn\gg\ll N\gg \land IOpSqSnap\ll Sn\gg\ll N\gg) \\
\vdash ?\ (\!|\ \neg\ |\!)^?(\exists\, Sys\ll Op\gg \bullet OpSqSnap\ll Sn\gg\ll N\gg \land IOpSnap\ll Sn\gg\ll N\gg \\
\qquad \land\ OpSqSnap\ll Sn\gg\ll N+1\gg\ ')
\end{array}
$$

]\!]

□

E

Z Simulation Rules for Behavioural Inheritance

This appendix derives the Z simulation rules for behavioural inheritance in the ZOO style that are discussed in chapter 5. First, the theory of data refinement for Z is given a brief overview. Then, the rules are derived based on this theory. Finally, one of the relaxations to the behavioural inheritance rules is presented in detail.

E.1 A brief overview of Z's theory of data refinement

The theory of data refinement for Z is based on the general theory of *relational* data refinement. Relational because the operations of a data type are modelled as relations between states: the states before the execution of the operation and the states afterwards.

The theory of relational data refinement is given in terms of total relations. In this setting, refinement is simply set inclusion. In Z, however, operations may be partial. This means that the operation is only applicable to a subset of all the states of the data type: the states satisfying the operation's *precondition*, the *domain* of the relation. The theory of refinement based on total relations is adapted to Z by looking at ways of totalising partial relations.

The following presents a brief overview of the theory of relational data refinement (section E.1.1), and shows how this theory is adapted to Z (section E.1.2). The reader is referred to [WD96] for a more detailed presentation.

E.1.1 Relational data refinement and simulation

An abstract data type (ADT) comprises a state space, a collection of operations on the state space, an initialisation of the state space and a finalisation. So, an ADT is a quadruple,

$$(X, x_i, x_f, \{i : I \bullet xo_i\})$$

where

- X is the state space;

- $x_i \in G \leftrightarrow X$ is an initialisation;
- $x_f \in X \leftrightarrow G$ is a finalisation;
- and $\{i : I \bullet xo_i\}$ is an indexed collection of operations, where each $xo_i \in X \leftrightarrow X$.

One concrete type is a refinement of an abstract one if every sequence of operations on the concrete type is a possible effect of the same sequence of operations on the abstract. If we restrict our attention to total relations, we are able to provide a notion of data refinement based on set inclusion [HHS86, WD96]. Let us assume ADTs A and C, where the operations are indexed by the same set I, then C refines A when:

$$c_i \fatsemi co_1 \fatsemi co_2 \fatsemi \ldots \fatsemi co_n \fatsemi c_f \subseteq a_i \fatsemi ao_1 \fatsemi ao_2 \fatsemi \ldots \fatsemi ao_n \fatsemi a_f$$

In practice, this is difficult to demonstrate. But it can be simplified by considering the concept of *simulation*: it is shown, operation by operation, that the behaviour of the concrete type simulates the behaviour of the abstract.

Where the data types have different state spaces, there is a *retrieve relation* describing the relationship between concrete and abstract state. This relation allows us to compare abstract and concrete types. A simulation may be either *forwards* (from abstract to concrete) or *backwards* (from concrete to abstract).

The simulation rules of [HHS86] for total relations are as follows.

For abstract data types A and C. Consider retrieve relations s and r such that:

$$r : A \leftrightarrow C$$
$$s : C \leftrightarrow A$$

C refines A ($C \sqsupseteq A$)with r being a forwards simulation if:

$$c_i \subseteq a_i \fatsemi r$$
$$r \fatsemi c_f \subseteq a_f$$
$$r \fatsemi c_o \subseteq a_o \fatsemi r$$

C refines A with s being a forwards simulation if:

$$c_i \fatsemi s \subseteq a_i$$
$$c_f \subseteq s \fatsemi a_f$$
$$r \fatsemi c_o \subseteq a_o \fatsemi r$$

The two kinds of simulations are sufficient for refinement (anything they can prove is a refinement) and together they are necessary (any refinement can be proved using either one of them) [HHS86].

E.1.2 Z data refinement

In the relational setting, operations are relations between states: the state before and the state afterwards. In Z, however, these relations are partial whenever the operation has a precondition; this is the domain of the relation and gives the sets of states for which the operation is applicable.

The simulation rules of [HHS86] are applied to Z by totalising partial relations (see [WD96] for further details). Depending on the choice of totalisation, there are two main semantic

models: *non-blocking* and *blocking*. They differ in the way we interpret the behaviour of an operation outside the precondition. In the non-blocking model [WD96], outside the pre-condition *anything can happen*. In the blocking model [BDW99, Bol02], *nothing can happen* outside the pre-condition — the operation is *blocked*.

We now give the non-blocking and blocking refinement simulation rules for partial relations.

Non-Blocking Simulation Rules

The non-blocking forwards simulation rules for partial relations are [WD96]:

$$\begin{array}{ll} ci \subseteq ai \fatsemi r & (\mathit{Initialisation}) \\ r \fatsemi cf \subseteq af & (\mathit{Finalisation}) \\ \mathrm{ran}((\mathrm{dom}\ ao) \lhd r) \subseteq \mathrm{dom}\ co & (\mathit{Applicability}) \\ (\mathrm{dom}\ ao) \lhd r \fatsemi co \subseteq ao \fatsemi r & (\mathit{Correctness}) \end{array}$$

The non-blocking backwards simulation rules for partial relations are [WD96]:

$$\begin{array}{ll} ci \fatsemi s \subseteq ai & (\mathit{Initialisation}) \\ \underline{cf \subseteq s \fatsemi af} & (\mathit{Finalisation}) \\ \mathrm{dom}\ co \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}\ ao)) & (\mathit{Applicability}) \\ \mathrm{dom}(s \rhd (\mathrm{dom}\ ao) \rhd \ co \fatsemi s \subseteq s \fatsemi ao & (\mathit{Correctness}) \end{array}$$

Blocking Simulation Rules

The blocking forwards simulation rules for partial relations [BDW99, Bol02] differ only in the correctness rule:

$$\begin{array}{ll} ci \subseteq ai \fatsemi r & (\mathit{Initialisation}) \\ r \fatsemi cf \subseteq af & (\mathit{Finalisation}) \\ \mathrm{ran}((\mathrm{dom}\ ao) \lhd r) \subseteq \mathrm{dom}\ co & (\mathit{Applicability}) \\ r \fatsemi co \subseteq ao \fatsemi r & (\mathit{Correctness}) \end{array}$$

Like in the forwards case, the backward simulation rules for partial relations [BDW99, Bol02] also differ only in the correctness rule::

$$\begin{array}{ll} ci \fatsemi s \subseteq ai & (\mathit{Initialisation}) \\ \underline{cf \subseteq s \fatsemi af} & (\mathit{Finalisation}) \\ \mathrm{dom}\ co \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}\ ao)) & (\mathit{Applicability}) \\ co \fatsemi s \subseteq s \fatsemi ao & (\mathit{Correctness}) \end{array}$$

E.2 Deriving Z Simulation Rules for Behavioural Inheritance

This section departs from the blocking and non-blocking simulations rules of Z refinement (above) to derive simulation rules specific to the subclassing retrieve relation of ZOO.

In the intensional view, the state of a subclass is defined by extending the state of its superclass(es). For example, suppose a class whose state is defined by schema A (standing for *abstract*), then the state of a class C (standing for *concrete*), subclass of A, is defined by:

$$C == A \land X$$

(X is the extra state of C, which is empty if the subclass does not have extra state.)

In this case, the retrieve relation between C and A is a total function from concrete to abstract:

$$f == \lambda\, C \bullet \theta A$$

We now derive simulation rules for this retrieve relation.

E.2.1 The simpler case, no communication

First, we turn our attention to the simpler case of operations with no notion of communication (inputs and outputs). We assume for now that the environment provides all the required inputs at initialisation, and that the system provides all the outputs to the environment at finalisation. The more real case where operations can receive inputs and produce outputs will be considered later.

For the purposes of our derivation, we consider the following relations corresponding to operation schemas of data types A and C:

$$ao = \{AO \bullet (\theta A) \mapsto (\theta A')\}$$
$$co = \{CO \bullet (\theta C) \mapsto (\theta C')\}$$

Initialisation and finalisation denote sets of states. But we need to transform these into relations. For initialisation we need a relation from the environment to the set of initialisation states. And for finalisation we need a relation from the set of finalisation states to the environment. We introduce the set G (stands for global) to represent the environment:

$$ai = G \times \{AI \bullet \theta A'\}$$
$$ci = G \times \{CI \bullet \theta C'\}$$
$$af = \{AF \bullet \theta A\} \times G$$
$$cf = \{CF \bullet \theta C\} \times G$$

The finalisation may be total or partial. It is partial whenever a data type has a finalisation condition (AF and CF); if it has none it is total ($AF = A$ and $CF = C$).

The forwards and backwards retrieve relations are:

$$r == f^{\sim}$$
$$s == f$$

In this setting, we derived simulation rules for Z schemas following the approach of [WD96, chapter 17]. This was done for the non-blocking and blocking interpretations of Z refinement. In both cases, forwards and backwards simulation reduce to the same of rules. The proofs of these derivations are given in appendix G.

The derived simulation rules are as follows:

Initialisation	$\forall\, C' \bullet CI \Rightarrow AI$	proofs G.1, G.2
Applicability	$\forall\, C \bullet \mathrm{pre}\, AO \Rightarrow \mathrm{pre}\, CO$	proofs G.3, G.4
Correctness, non-blocking	$\forall\, C';\ C \bullet \mathrm{pre}\, AO \wedge CO \Rightarrow AO$	proofs G.5, G.6
Correctness, blocking	$\forall\, C';\ C \bullet CO \Rightarrow AO$	proofs G.7, G.8
Finalisation	$\forall\, C \bullet CF \Rightarrow AF$	proofs G.9, G.10

If the finalisation is total (the ADT does not have a finalisation condition) the finalisation rule reduces to *true*.

E.2.2 Embedding Inputs and Outputs

We now consider operations that communicate with the environment.

To model communication, we introduce the parallel composition operator (from [WD96]):

$$[W, X, Y, Z]$$
$$\_ \parallel \_ : (W \leftrightarrow Y) \times (X \leftrightarrow Z) \rightarrow (W \times X \leftrightarrow Y \times Z)$$
$$\forall r : W \leftrightarrow Y;\ s : X \leftrightarrow Z;\ w : W;\ x : X;\ y : Y;\ z : Z \bullet$$
$$(w, x) \mapsto (y, z) \in r \parallel s \Leftrightarrow w \mapsto y \in r \land x \mapsto z \in s$$

And redefine the relations of operations, introducing inputs and outputs:

$$ao = \{AO \bullet (\theta A, i?) \mapsto (\theta A', o!)\}$$
$$co = \{CO \bullet (\theta C, i?) \mapsto (\theta C', o!)\}$$

In this setting, we derived applicability and correctness rules, for both the blocking and non-blocking interpretations. Like in the simpler setting, for both interpretations, backwards and forwards simulation reduce to the same set of rules. The derived rules are as follows:

Applicability	$\forall C;\ i? : I \bullet \text{pre}\, AO \Rightarrow \text{pre}\, CO$	proofs G.11, G.12
Correctness, non-blocking	$\forall C';\ C;\ i? : I;\ o! : O \bullet \text{pre}\, AO \land CO \Rightarrow AO$	proofs G.13, G.14
Correctness, blocking	$\forall C';\ C;\ i? : I;\ o! : O \bullet CO \Rightarrow AO$	proofs G.15, G.16

E.3 Adding operations to the subclass: Relaxing

In OO systems, subclasses may have operations that are not specialisation of superclass operations. In this section, we study this feature in the context of inheritance refinement. We want to know what are the conditions for doing this — what conjectures are we required to prove?

Suppose we add operation CO, which is not a specialisation, to class C (whose intention is defined as above). Is C a refinement of A? What do we need to prove?

To prove the refinement, we need to add an operation to A so that it simulates CO. In model refinement this is usually done by adding *skip* (ΞA), but this is very restrictive; we are very limited in terms of what we can express in CO. So, we try to find another abstract operation. Let's see how we can do this.

In the simpler case of operations with no communication, CO is defined in the world of relations as:

$$co = \{CO \bullet (\theta C) \mapsto (\theta C')\}$$

Given the properties of our refinement retrieve relation (a total function from concrete to abstract), we can obtain an abstract (or superclass) operation from the concrete one by calculation. This abstract operation is given by the formula:

$$ao = f^{\sim} \fatsemi co \fatsemi f$$

Now the question is: is it safe to add this operation to the abstract type? It is. Recall that a class is a promoted ADT. This operation is defined internally (class intension) and is

never made visible outside the scope of the class; we do not promote it. Hence, there is no problem in adding these operations to the abstract class: they are never ran.

In this setting where data types have no notion communication, we proved that $co \sqsupseteq ao$ for any CO. We did this in both the blocking and non-blocking interpretations of Z refinement. The proofs are given in appendix G; The following has been proved:

- forwards and backwards applicability (proofs G.19 and G.20);
- non-blocking forwards and backwards correctness (proofs G.21 and G.22);
- and blocking forwards and backwards correctness (proofs G.23 and G.24).

We also proved the refinement in a setting where data types can communicate. First we make some changes to the definitions. We change the relational representation of CO to include communication:

$$co = \{CO \bullet (\theta C, i?) \mapsto (\theta C', o!)\}$$

And the abstract operation to:

$$ao == (f^{\sim} \parallel id[I]) \fatsemi co \fatsemi (f \parallel id[O])$$

The proofs that $co \sqsupseteq ao$ in this setting are given in appendix G. The following has been proved:

- forwards and backwards applicability (proofs G.27 and G.28);
- non-blocking forwards and backwards correctness (proofs G.29 and G.30);
- and blocking forwards and backwards correctness (proofs G.31 and G.32).

F

ZOO Models

This appendix presents full ZOO models of the case studies used in the main text. The following presents the ZOO model that is *fully generated* from the diagrams of the Bank case study of chapter 4; the full ZOO model of Bank with inheritance that is developed in chapter 5; and the changes to this latter model resulting from the snapshot analysis performed in chapter 6.

F.1 Fully generated Bank

The following presents the ZOO model that is *fully generated* from the diagrams of the Bank case study of chapter 4.

F.1.1 Structural View

section $Bank\_model$ parents $ZOO\_toolkit$

$$CLASS ::= CustomerCl \mid AccountCl$$

$$\begin{array}{|l} subCl : CLASS \leftrightarrow CLASS \\ abstractCl : \mathbb{P}\ CLASS \\ rootCl : \mathbb{P}\ CLASS \\ \hline subCl = \{\} \\ abstractCl = \{\} \\ rootCl = CLASS \setminus \mathrm{dom}\ subCl \end{array}$$

$$\mathbb{O}_x : CLASS \rightarrow \mathbb{P}_1\, OBJ$$
$$\mathbb{O} : CLASS \rightarrow \mathbb{P}_1\, OBJ$$

---

disjoint $\mathbb{O}_x$

$$\forall\, cl : CLASS \bullet \mathbb{O}\ cl = \mathbb{O}_x\ cl \cup \bigcup(\mathbb{O}_x(\!|\ (subCl^+)^{\sim}(\!|\ \{cl\}\ |\!)\ |\!))$$

$$\forall\, cl, cl' : CLASS \mid cl \mapsto cl' \in subCl \bullet \mathbb{O}\ cl \subseteq \mathbb{O}\ cl'$$

F.1.2 Class **Customer**

Intensional View

Class Types

$$[NAME, ADDRESS]$$
$$CUSTTYPE ::= company \mid personal$$

State and Initialisation

Customer

$$name : NAME$$
$$address : ADDRESS$$
$$ctype : CUSTTYPE$$

CustomerInit

$$Customer'$$
$$name? : NAME$$
$$address? : ADDRESS$$
$$ctype? : CUSTTYPE$$

$$name' = name?$$
$$address' = address?$$
$$ctype' = ctype?$$

$$\vdash?\ \exists\, Customer';\ name? : NAME;\ address? : ADDRESS;\ ctype? : CUSTTYPE \bullet CustomerInit$$

Extensional View

State and Initialisation

$$\mathbb{S}Customer == \mathbb{S}\mathrm{CL}[\mathbb{O}CustomerCl, Customer][sCustomer/os, stCustomer/oSt]$$

$\mathbb{S}$CustomerInit

$$\mathbb{S}Customer'$$

$$sCustomer' = \varnothing \wedge stCustomer' = \varnothing$$

$$\vdash?\ \exists\ \mathbb{S}Customer' \bullet \mathbb{S}CustomerInit$$

Operations

$\Phi\mathbb{S}CustomerN$

$\Delta\mathbb{S}Customer$
$Customer'$
$oCustomer! : \mathbb{O}CustomerCl$

$oCustomer! \in \mathbb{O}CustomerCl \setminus sCustomer$
$sCustomer' = sCustomer \cup \{oCustomer!\}$
$stCustomer' = stCustomer \cup \{oCustomer! \mapsto \theta Customer'\}$

$\mathbb{S}_\Delta CustomerNew == \exists\ Customer' \bullet \Phi\mathbb{S}CustomerN \land CustomerInit$

$\mathbb{S}_\Delta CustomerDelete$

$\Delta\mathbb{S}Customer$
$osCustomer? : \mathbb{P}(\mathbb{O}CustomerCl)$

$sCustomer' = sCustomer \setminus osCustomer?$
$stCustomer' = osCustomer? \lhd stCustomer$

F.1.3 Class **Account**

Intensional View

Attribute Types

$[ACCID]$
$ACCTYPE ::= current \mid savings$

State and Initialisation

$AccountST ::= active \mid suspended$

Account

$accountNo : ACCID$
$st : AccountST$
$balance : \mathbb{N}$
$atype : ACCTYPE$

AccountInit

$Account'$
$accountNo? : ACCID$
$atype? : ACCTYPE$

$accountNo' = accountNo?$
$st' = active$
$balance' = 0$
$atype' = atype?$

$\vdash?\ \exists Account';\ accountNo? : ACCID;\ atype? : ACCTYPE \bullet AccountInit$

Operations

Account$_\Delta$ *Withdraw*

$\Delta Account$
$amount? : \mathbb{N}$

$st = active \land st' = active$

Account$_\Delta$ *Deposit*

$\Delta Account$
$amount? : \mathbb{N}$

$(st = active \land st' = active)$
$\quad \lor (st = suspended \land st' = suspended)$

Account$_\Delta$ *Suspend*

$\Delta Account$

$st = active \land st' = suspended$

Account$_\Delta$ *ReActivate*

$\Delta Account$

$st = suspended \land st' = active$

Account$_\Xi$ *GetBalance*

$\Xi Account$
$balance! : \mathbb{Z}$

$st = active \lor st = suspended$

Finalisation

AccountFin

$Account$

$st = active \lor st = suspended$
$balance = 0$

Extensional view

State Space and Initialisation

$\mathbb{S}Account == [\ \mathbb{S}\mathrm{CL}[\mathbb{O}AccountCl, Account][sAccount/os, stAccount/oSt]\ ]$
$\mathbb{S}AccountInit == [\ \mathbb{S}Account' \mid sAccount' = \varnothing \land stAccount' = \varnothing\ ]$

$\vdash \exists\ \mathbb{S}Account' \bullet \mathbb{S}AccountInit$

Operations

$\Phi\mathbb{S}AccountNI$

$\Delta\mathbb{S}Account$
$Account\ '$
$oAccount! : \mathbb{O}AccountCl$

$oAccount! \in \mathbb{O}AccountCl \setminus sAccount$
$sAccount' = sAccount \cup \{oAccount!\}$
$stAccount' = stAccount \cup \{oAccount! \mapsto \theta Account'\}$

$$\mathbb{S}_\Delta AccountNew == \exists\ Account' \bullet \Phi\mathbb{S}AccountNI \land AccountInit$$

$\Phi\mathbb{S}AccountUI$

$$\Delta\mathbb{S}Account$$
$$\Delta Account$$
$$oAccount? : \mathbb{O}AccountCl$$

$$oAccount? \in sAccount$$
$$\theta Account = stAccount\ oAccount?$$
$$sAccount' = sAccount$$
$$stAccount' = stAccount \oplus \{oAccount? \mapsto \theta Account'\}$$

$$\mathbb{S}_\Delta AccountWithdraw == \exists \Delta Account \bullet \Phi\mathbb{S}AccountUI \land Account_\Delta Withdraw$$
$$\mathbb{S}_\Delta AccountDeposit == \exists \Delta Account \bullet \Phi\mathbb{S}AccountUI \land Account_\Delta Deposit$$
$$\mathbb{S}_\Delta AccountSuspend == \exists \Delta Account \bullet \Phi\mathbb{S}AccountUI \land Account_\Delta Suspend$$
$$\mathbb{S}_\Delta AccountReActivate == \exists \Delta Account \bullet \Phi\mathbb{S}AccountUI \land Account_\Delta ReActivate$$

$\Phi\mathbb{S}AccountO$

$$\Xi\mathbb{S}Account$$
$$\Xi Account$$
$$oAccount? : \mathbb{O}AccountCl$$

$$oAccount? \in sAccount$$
$$\theta Account = stAccount\ oAccount?$$

$$\mathbb{S}_\Xi AccountGetBalance == \exists\ \Xi Account \bullet \Phi\mathbb{S}AccountO \land Account_\Xi GetBalance$$

$\Phi\mathbb{S}AccountDI$

$$\Delta\mathbb{S}Account$$
$$Account$$
$$oAccount? : \mathbb{O}AccountCl$$

$$oAccount? \in sAccount$$
$$\theta Account = stAccount\ oAccount?$$
$$sAccount' = sAccount \setminus \{oAccount?\}$$
$$stAccount' = \{oAccount?\} \lhd stAccount$$

$$\mathbb{S}_\Delta AccountDelete == \exists Account \bullet \Phi\mathbb{S}AccountDI \land AccountFin$$

F.1.4 Association Holds

$\mathbb{A}Holds$

$rHolds : \mathbb{O}CustomerCl \leftrightarrow \mathbb{O}AccountCl$

$\mathbb{A}HoldsInit$

$\mathbb{A}Holds'$

$rHolds' = \varnothing$

$\vdash ?\ \exists \mathbb{A}HoldsInit \bullet true$

F.1.5 Global View

State

$Link\mathbb{A}Holds$

$\mathbb{S}Customer;\ \mathbb{S}Account;\ \mathbb{A}Holds$

$\mathsf{mult}(rHolds, sCustomer, sAccount, om, \{\}, \{\})$

$SysCnt == Link\mathbb{A}Holds$

$System$

$\mathbb{S}Customer;\ \mathbb{S}Account;\ \mathbb{A}Holds$

$SysCnt$

Initialisation

$SysInit == System' \land \mathbb{S}CustomerInit \land \mathbb{S}AccountInit \land \mathbb{A}HoldsInit$

$\vdash \exists System' \bullet SysInit$

F.2 Bank with inheritance

The following presents the ZOO model of Bank with inheritance that is developed in chapter 5 and the changes to this model resulting from snapshot analysis performed in chapter 6. This model has been generated from templates of the $UML + Z$ catalogue.

F.2.1 Structural View

section $BankInh\_model$ parents $ZOO\_toolkit$

$$CLASS ::= CustomerCl \mid AccountCl \mid CurrentCl \mid SavingsCl \mid WBasedCl \mid BalBasedCl$$

$$\begin{array}{|l} subCl : CLASS \leftrightarrow CLASS \\ abstractCl : \mathbb{P}\ CLASS \\ rootCl : \mathbb{P}\ CLASS \\ \hline subCl = \{CurrentCl \mapsto AccountCl, \\ \quad SavingsCl \mapsto AccountCl, \\ \quad WBasedCl \mapsto SavingsCl, \\ \quad BalBasedCl \mapsto SavingsCl\} \\ abstractCl = \{AccountCl, SavingsCl\} \\ rootCl = CLASS \setminus \mathrm{dom}\ subCl \end{array}$$

$$\begin{array}{|l} \mathbb{O}_x : CLASS \to \mathbb{P}_1\ OBJ \\ \mathbb{O} : CLASS \to \mathbb{P}_1\ OBJ \\ \hline \mathrm{disjoint}\ \mathbb{O}_x \\ \forall\, cl : CLASS \bullet \mathbb{O}\ cl = \mathbb{O}_x\ cl \cup \bigcup(\mathbb{O}_x(\!|\ (subCl^+)^{\sim}(\!|\ \{cl\}\ |\!)\ |\!)) \\ \forall\, cl, cl' : CLASS \mid cl \mapsto cl' \in subCl \bullet \mathbb{O}\ cl \subseteq \mathbb{O}\ cl' \end{array}$$

F.2.2 Class Customer

Intensional View

$$[NAME, ADDRESS]$$
$$CUSTTYPE ::= company \mid personal$$

Customer

$$\begin{array}{|l} name : NAME \\ address : ADDRESS \\ type : CUSTTYPE \end{array}$$

CustomerInit

$$\begin{array}{|l} Customer\ ' \\ CustomerInit_I \\ \hline name' = name? \\ address' = address? \\ type' = type? \end{array}$$

CustomerInit$_I$

$$\begin{array}{|l} name? : NAME \\ address? : ADDRESS \\ type? : CUSTTYPE \end{array}$$

Extensional View

$$\mathbb{S}Customer == \mathbb{S}\mathrm{CL}[\mathbb{O}CustomerCl, Customer][sCustomer/os, stCustomer/oSt]$$
$$\mathbb{S}CustomerInit == [\ \mathbb{S}Customer\ ' \mid sCustomer' = \varnothing \wedge stCustomer' = \varnothing\ ]$$

$\Phi\mathbb{S}CustomerN$

$\Delta\mathbb{S}Customer$
$Customer\ '$
$oCustomer! : \mathbb{O}CustomerCl$

$oCustomer! \in (\mathbb{O}_x CustomerCl \setminus sCustomer)$
$sCustomer' = sCustomer \cup \{oCustomer!\}$
$stCustomer' = stCustomer \cup \{oCustomer! \mapsto \theta Customer'\}$

$\mathbb{S}_\Delta CustomerNew == \exists\ Customer' \bullet \Phi\mathbb{S}CustomerN \land CustomerInit$

$\mathbb{S}_\Delta CustomerDelete$

$\Delta\mathbb{S}Customer$
$osCustomer? : \mathbb{P}(\mathbb{O}CustomerCl)$

$sCustomer' = sCustomer \setminus osCustomer?$
$stCustomer' = osCustomer? \lhd stCustomer$

F.2.3 Class **Account**

Intensional View

$[ACCID]$
$AccountST ::= active \mid suspended$

$Account$

$st : AccountST$
$accountNo : ACCID$
$balance : \mathbb{Z}$

$AccountInit_I$

$accountNo? : ACCID$
$opAmount? : \mathbb{N}$

$AccountInit$

$Account\ '$
$AccountInit_I$

$st' = active$
$accountNo' = accountNo?$
$balance' = opAmount?$

$Account_\Delta Withdraw$

$\Delta Account$
$amount? : \mathbb{N}$

$st = active \land st' = active$
$accountNo' = accountNo$
$balance' = balance - amount?$

$Account_\Delta Deposit$

$\Delta Account$
$amount? : \mathbb{N}$

$(st = active \land st' = active)$
$\quad \lor (st = suspended \land st' = suspended)$
$accountNo' = accountNo$
$balance' = balance + amount?$

$Account_{\Delta} Suspend$

$\Delta Account$

$st = active \land st' = suspended$

$accountNo' = accountNo$

$balance' = balance$

$Account_{\Delta} ReActivate$

$\Delta Account$

$st = suspended \land st' = active$

$accountNo' = accountNo$

$balance' = balance$

$Account_{\Xi} GetBalance$

$\Xi Account$

$balance! : \mathbb{Z}$

$balance! = balance$

$AccountFin$

$Account$

$st = active \lor st = suspended$

$balance = 0$

$AccountH == [\ \Delta Account \mid st = st'\ ]$

Extensional View

$\mathbb{S}Account$

$\mathbb{S}\mathrm{CL}[\mathbb{O}AccountCl, Account][sAccount/os, stAccount/oSt]$

$sAccount \cap \mathbb{O}_x AccountCl = \varnothing$

$\mathbb{S}AccountInit == [\ \mathbb{S}Account' \mid sAccount' = \varnothing \land stAccount' = \varnothing\ ]$

$\Phi\mathbb{S}AccountNI_0$

$\Delta\mathbb{S}Account$

$Account'$

$oAccount! : \mathbb{O}AccountCl$

$sAccount' = sAccount \cup \{oAccount!\}$

$stAccount' = stAccount \cup \{oAccount! \mapsto \theta Account\ '\}$

$\Phi\mathbb{S}AccountUI_0$

$\Delta\mathbb{S}Account$

$\Delta Account$

$oAccount? : \mathbb{O}AccountCl$

$sAccount' = sAccount$

$stAccount' = stAccount \oplus \{oAccount? \mapsto \theta Account\ '\}$

$\Phi\mathbb{S}AccountO$

$\Xi\mathbb{S}Account$

$\Xi Account$

$oAccount? : \mathbb{O}AccountCl$

$oAccount? \in sAccount$

$\theta Account = stAccount\ \ oAccount?$

$$\mathbb{S}_\Xi AccountGetBalance == \exists\ \Xi Account \bullet \Phi\mathbb{S}AccountO \land Account_\Xi GetBalance$$

Get's all the sAccount that have negative balances.

$\mathbb{S}_\Xi AccountGetDebtAccounts$

$\Xi\mathbb{S}Account$

$osAccount! : \mathbb{P}(\mathbb{O}AccountCl)$

$osAccount! = \{a : sAccount \mid (stAccount\ a).balance < 0\}$

$\Phi\mathbb{S}AccountDI_0$

$\Delta\mathbb{S}Account$

$Account$

$oAccount? : \mathbb{O}AccountCl$

$sAccount' = sAccount \setminus \{oAccount?\}$

$stAccount' = \{oAccount?\} \lhd stAccount$

F.2.4 Class Current

Intensional View We need a function that gives the required amount to open a new account. There are 3 types of actual accounts, *current*, *WTiered* and *BalTiered*; for each of these types there is a required amount.

$$ACCTY ::= current \mid wbased \mid balbased$$

$reqOpAmount : ACCTY \to \mathbb{N}$

$Current_0$

$overdraft : \mathbb{N}$

$Current$

$Account$

$Current_0$

$balance + overdraft \geq 0$

$CurrentInit_I$

$AccountInit_I$

$overdraft? : \mathbb{N}$

$CurrentInit$

$Current\ '$

$AccountInit$

$CurrentInit_I$

$opAmount? \geq reqOpAmount\ current$

$overdraft' = overdraft?$

$$Current_\Delta Withdraw == [\ \Delta Current;\ Account_\Delta Withdraw \mid overdraft' = overdraft\ ]$$
$$Current_\Delta Deposit == [\ \Delta Current;\ Account_\Delta Deposit \mid overdraft' = overdraft\ ]$$
$$Current_\Delta Suspend == [\ \Delta Current;\ Account_\Delta Suspend \mid overdraft' = overdraft\ ]$$
$$Current_\Delta ReActivate == [\ \Delta Current;\ Account_\Delta ReActivate \mid overdraft' = overdraft\ ]$$

Current$_\Delta$ *SetOverdraft*

$\Delta Current$
$AccountH$
$\Xi Account$
$overdraft? : \mathbb{N}$

$overdraft' = overdraft?$

Current$_\Xi$ *GetOverdraft*

$\Xi Current$
$overdraft! : \mathbb{N}$

$overdraft! = overdraft$

$Current_\Xi GetBalance == \Xi Current \land Account_\Xi GetBalance$

CurrentFin

$AccountFin$

Extensional View

$\mathbb{S}Current == \mathbb{S}\mathrm{CL}[\mathbb{O}CurrentCl, Current][sCurrent/os, stCurrent/oSt]$
$\mathbb{S}CurrentInit == [\ \mathbb{S}Current\ ' \mid sCurrent' = \varnothing \land stCurrent' = \varnothing\ ]$

$\mathbb{S}CurrentIsAccount$

$\mathbb{S}Account;\ \mathbb{S}Current$

$sCurrent \subseteq sAccount$
$\forall\, oCurrent : sCurrent \bullet$
$\quad (\lambda\ \ Current \bullet \theta Account)(stCurrent\ \ oCurrent) = stAccount\ \ oCurrent$

$\Phi\mathbb{S}CurrentNI_0$

$\Phi\mathbb{S}AccountNI_0[oCurrent!/oAccount!]$
$\Delta\mathbb{S}Current$
$Current\ '$
$oCurrent! : \mathbb{O}AccountCl$

$sCurrent' = sCurrent \cup \{oCurrent!\}$
$stCurrent' =$
$\quad stCurrent \cup \{oCurrent! \mapsto \theta Current\ '\}$

$\Phi\mathbb{S}CurrentNI$

$\Phi\mathbb{S}CurrentNI_0$

$oCurrent! \in \mathbb{O}_x CurrentCl \setminus sCurrent$

$\mathbb{S}_\Delta CurrentNew == \exists\ \ Current\ ' \bullet \Phi\mathbb{S}CurrentNI \land CurrentInit$

$\Phi\mathbb{S}CurrentUI_0$

$\Phi\mathbb{S}AccountUI_0[oCurrent?/oAccount?]$
$\Delta\mathbb{S}Current$
$\Delta Current$
$oCurrent? : \mathbb{O}CurrentCl$

$sCurrent' = sCurrent$
$stCurrent' =$
$\quad stCurrent \oplus \{oCurrent? \mapsto \theta Current\ '\}$

$\Phi\mathbb{S}CurrentUI$

$\Phi\mathbb{S}CurrentUI_0$

$oCurrent? \in sCurrent \cap \mathbb{O}_x CurrentCl$
$\theta Current = stCurrent\ \ oCurrent?$

$\mathbb{S}_{\Delta}\, CurrentWithdraw == \exists\ \Delta Current \bullet \Phi\mathbb{S} CurrentUI \land Current_{\Delta}\, Withdraw$
$\mathbb{S}_{\Delta}\, CurrentDeposit == \exists\ \Delta Current \bullet \Phi\mathbb{S} CurrentUI \land Current_{\Delta}\, Deposit$
$\mathbb{S}_{\Delta}\, CurrentSetOverdraft == \exists\ \Delta Current \bullet \Phi\mathbb{S} CurrentUI \land Current_{\Delta}\, SetOverdraft$
$\mathbb{S}_{\Delta}\, CurrentSuspend == \exists \Delta Current \bullet \Phi\mathbb{S} CurrentUI \land Current_{\Delta}\, Suspend$
$\mathbb{S}_{\Delta}\, CurrentReActivate == \exists \Delta Current \bullet \Phi\mathbb{S} CurrentUI \land Current_{\Delta}\, ReActivate$

$\Phi\mathbb{S} CurrO$

$\Xi\mathbb{S} Current$
$\Xi Current$
$oCurrent? : \mathbb{O} CurrentCl$

$oCurrent? \in sCurrent$
$\theta Current = stCurrent\ oCurrent?$

$\mathbb{S}_{\Xi}\, CurrentGetBalance == \exists\ \Xi Current \bullet \Phi\mathbb{S} CurrO \land Current_{\Xi}\, GetBalance$

$\Phi\mathbb{S} CurrentDI_0$

$\Phi\mathbb{S} AccountDI_0[oCurrent?/oAccount?]$
$\Delta\mathbb{S} Current$
$Current$
$oCurrent? : \mathbb{O} CurrentCl$

$sCurrent' = sCurrent \setminus \{oCurrent?\}$
$stCurrent' = \{oCurrent?\} \lhd stCurrent$

$\Phi\mathbb{S} CurrentDI$

$\Phi\mathbb{S} CurrentDI_0$

$oCurrent? \in sCurrent \cap \mathbb{O}_x\, CurrentCl$
$\theta Current = stCurrent\ oCurrent?$

$\mathbb{S}_{\Delta}\, CurrentDelete == \exists\, Current \bullet \Phi\mathbb{S} CurrentDI \land CurrentFin$

F.2.5 Class **Savings**

Intensional view

$[DATE]$
$DATE \neq \varnothing$

$PERCENTAGE == 1 \,..\, 100$
$TERM ::= month \mid quarter \mid year$

$nxtCredDt : (DATE \times TERM) \rightarrow DATE$

$Savings_0$

$interestR : PERCENTAGE$
$term : TERM$
$interestDt : DATE$

$Savings$

$Account$
$Savings_0$

$balance \geq 0$

$SavingsInit_I$

$AccountInit_I$
$interestR? : PERCENTAGE$
$term? : TERM$
$today? : DATE$

$SavingsInit$

$Savings\ '$
$AccountInit$
$SavingsInit_I$

$interestR' = interestR?$
$term' = term?$
$interestDt' = today?$

$Savings_\Delta Withdraw == [\Delta Savings;\ Account_\Delta Withdraw]$

$Savings_\Delta Deposit == [\Delta Savings;\ Account_\Delta Deposit]$

$Savings_\Delta Suspend == [\ \Delta Savings;\ Account_\Delta Suspend \mid$
$interestR' = interestR \land term' = term \land interestDt' = interestDt\ ]$

$Savings_\Delta ReActivate == [\ \Delta Savings;\ Account_\Delta ReActivate \mid$
$interestR' = interestR \land term' = term \land interestDt' = interestDt\ ]$

$Savings_\Delta PayInterest$

$\Delta Savings$
$AccountH$
$today? : DATE$

$nxtCredDt(interestDt, term) = today?$
$accountNo' = accountNo$
$balance' = balance + (balance * interestR') \operatorname{div} 100$
$interestDt' = today?$
$term' = term$

$Savings_\Delta ChCreditTerm$

$\Delta Savings$
$AccountH$
$\Xi Account$
$newTerm? : TERM$

$term' = newTerm?$
$interestR' = interestR$
$interestDt' = interestDt$

$Savings_\Xi GetInterestRate$

$\Xi Savings$
$interestR! : PERCENTAGE$

$interestR! = interestR$

$Savings_\Xi GetBalance == \Xi Savings \land Account_\Xi GetBalance$

SavingsFin

$AccountFin$

Extensional view

$\mathbb{S}Savings$

$\mathbb{S}\mathrm{CL}[\mathbb{O}SavingsCl, Savings][sSavings/os, stSavings/oSt]$

$sSavings \cap \mathbb{O}_x SavingsCl = \varnothing$

$\mathbb{S}SavingsInit == [\ \mathbb{S}Savings\ ' \mid sSavings' = \varnothing \land stSavings' = \varnothing\ ]$

$\mathbb{S}SavingsIsAccount$

$\mathbb{S}Account;\ \mathbb{S}Savings$

$sSavings \subseteq sAccount$

$\forall\, oSavings : sSavings \bullet$
$\quad (\lambda\ \ Savings \bullet \theta Account)(stSavings\ \ oSavings) = stAccount\ \ oSavings$

$\Phi\mathbb{S}SavingsNI_0$

$\Phi\mathbb{S}AccountNI_0[oSavings!/oAccount!]$
$\Delta\mathbb{S}Savings$
$Savings\ '$
$oSavings! : \mathbb{O}SavingsCl$

$sSavings' = sSavings \cup \{oSavings!\}$
$stSavings' = stSavings \cup \{oSavings! \mapsto \theta Savings\ '\}$

$\Phi\mathbb{S}SavingsUI_0$

$\Phi\mathbb{S}AccountUI_0[oSavings?/oAccount?]$
$\Delta\mathbb{S}Savings$
$\Delta Savings$
$oSavings? : \mathbb{O}SavingsCl$

$sSavings' = sSavings$
$stSavings' = stSavings \oplus \{oSavings? \mapsto \theta Savings'\}$

$\Phi\mathbb{S}SavingsO$

$\Xi\mathbb{S}Savings$
$\Xi Savings$
$oSavings? : \mathbb{O}SavingsCl$

$oSavings? \in sSavings$
$\theta Savings = stSavings\ \ oSavings?$

$\mathbb{S}_\Xi SavingsGetBalance == \exists\ \Xi Savings \bullet \Phi\mathbb{S}SavingsO \land Savings_\Xi GetBalance$

$\mathbb{S}_\Xi SavingsGetInterestRate == \exists\ \Xi Savings \bullet \Phi\mathbb{S}SavingsO \land Savings_\Xi GetInterestRate$

$\Phi\mathbb{S}SavingsDI_0$

$\Phi\mathbb{S}AccountDI_0[oSavings?/oAccount?]$
$\Delta\mathbb{S}Savings$
$Savings$
$oSavings? : \mathbb{O}SavingsCl$

$sSavings' = sSavings \setminus \{oSavings?\}$
$stSavings' = \{oSavings?\} \lhd stSavings$

F.2.6 Class WBased

Intensional view

$maxInitInterest : (\{wbased, balbased\} \times \mathbb{Z}) \to PERCENTAGE$

$WBased_0$

$noWithds : \mathbb{N}$

$WBasedInit_I$

$SavingsInit_I$

$WBased$

$Savings$
$WBased_0$

$WBasedInit$

$WBased\ '$
$SavingsInit$
$WBasedInit_I$

$opAmount? \geq reqOpAmount\ wbased$
$interestR? \leq maxInitInterest(wbased, balance')$
$noWithds' = 0$

$WBased_\Delta Withdraw$

$\Delta WBased$
$Savings_\Delta Withdraw$

$noWithds' = noWithds + 1$
$interestR' = interestR$
$interestDt' = interestDt$
$term' = term$

$WBased_\Delta Deposit$

$\Delta WBased$
$Savings_\Delta Deposit$

$noWithds' = noWithds$
$interestDt' = interestDt$
$term' = term$

$calcInterestW : (\mathbb{N} \times PERCENTAGE) \to PERCENTAGE$

$WBased_{\Delta} PayInterest$

$\Delta WBased$
$Savings_{\Delta} PayInterest$

$interestR' = calcInterestW(noWithds, interestR)$
$noWithds' = 0$

$WBased_{\Delta} Suspend == [\ \Delta WBased;\ Savings_{\Delta} Suspend \mid noWithds' = noWithds\ ]$
$WBased_{\Delta} ReActivate == [\ \Delta WBased;\ Savings_{\Delta} ReActivate \mid noWithds' = noWithds\ ]$

$WBased_{\Delta} ChCreditTerm == \Delta WBased \land \Xi WBased_0 \land Savings_{\Delta} ChCreditTerm$
$WBased_{\Xi} GetBalance == \Xi WBased \land Savings_{\Xi} GetBalance$
$WBased_{\Xi} GetInterestRate == \Xi WBased \land Savings_{\Xi} GetInterestRate$

$WBased_{\Xi} GetNoWithdrawls$

$\Xi WBased$
$noWithds! : \mathbb{N}$

$noWithds! = noWithds$

$WBasedFin$

$SavingsFin$

Extensional view

$\mathbb{S} WBased == \mathbb{S}\mathrm{CL}[\mathbb{O} WBasedCl,\ WBased][sWBased/os, stWBased/oSt]$

$\mathbb{S} WBasedInit == [\ \mathbb{S} WBased\ ' \mid sWBased' = \varnothing \land stWBased' = \varnothing\ ]$

$\mathbb{S} WBasedIsSavings$

$\mathbb{S} Savings;\ \mathbb{S} WBased$

$sWBased \subseteq sSavings$
$\forall\, oWBased : sWBased \bullet$
$\quad (\lambda\ \ WBased \bullet \theta Savings)(stWBased\ \ oWBased) = stSavings\ \ oWBased$

$\Phi\mathbb{S} WBasedNI_0$

$\Phi\mathbb{S} SavingsNI_0[oWBased!/oSavings!]$
$\Delta\mathbb{S} WBased$
$WBased\ '$
$oWBased! : \mathbb{O} WBasedCl$

$sWBased' = sWBased \cup \{oWBased!\}$
$stWBased' =$
$\quad stWBased \cup \{oWBased! \mapsto \theta WBased\ '\}$

$\Phi\mathbb{S} WBasedNI$

$\Phi\mathbb{S} WBasedNI_0$

$oWBased! \in \mathbb{O}_x WBasedCl \setminus sWBased$

$$\mathbb{S}_\Delta\, WBasedNew == \exists\ WBased\,' \bullet \Phi\mathbb{S}\, WBasedNI \land WBasedInit$$

$\Phi\mathbb{S}\, WBasedUI_0$

$$\Phi\mathbb{S}SavingsUI_0[oWBased?/oSavings?]$$
$$\Delta\mathbb{S}\, WBased$$
$$\Delta\, WBased$$
$$oWBased? : \mathbb{O}\, WBasedCl$$

$$sWBased' = sWBased$$
$$stWBased' = stWBased \oplus \{oWBased? \mapsto \theta\, WBased\,'\}$$

$\Phi\mathbb{S}\, WBasedUI$

$$\Phi\mathbb{S}\, WBasedUI_0$$

$$oWBased? \in sWBased \cap \mathbb{O}_x\, WBasedCl$$
$$\theta\, WBased = stWBased\ oWBased?$$

$$\mathbb{S}_\Delta\, WBasedWithdraw == \exists\ \Delta\, WBased \bullet \Phi\mathbb{S}\, WBasedUI \land WBased_\Delta\, Withdraw$$
$$\mathbb{S}_\Delta\, WBasedDeposit == \exists\ \Delta\, WBased \bullet \Phi\mathbb{S}\, WBasedUI \land WBased_\Delta\, Deposit$$
$$\mathbb{S}_\Delta\, WBasedPayInterest == \exists\ \Delta\, WBased \bullet \Phi\mathbb{S}\, WBasedUI \land WBased_\Delta\, PayInterest$$
$$\mathbb{S}_\Delta\, WBasedChCreditTerm == \exists\ \Delta\, WBased \bullet \Phi\mathbb{S}\, WBasedUI \land WBased_\Delta\, ChCreditTerm$$
$$\mathbb{S}_\Delta\, WBasedSuspend == \exists \Delta\, WBased \bullet \Phi\mathbb{S}\, WBasedUI \land WBased_\Delta\, Suspend$$
$$\mathbb{S}_\Delta\, WBasedReActivate == \exists \Delta\, WBased \bullet \Phi\mathbb{S}\, WBasedUI \land WBased_\Delta\, ReActivate$$

$\Phi\mathbb{S}\, WBasedO$

$$\Xi\mathbb{S}\, WBased$$
$$\Xi\, WBased$$
$$oWBased? : \mathbb{O}\, WBasedCl$$

$$oWBased? \in sWBased$$
$$\theta\, WBased = stWBased\ oWBased?$$

$$\mathbb{S}_\Xi\, WBasedGetBalance == \exists\ \Xi\, WBased \bullet \Phi\mathbb{S}\, WBasedO \land WBased_\Xi\, GetBalance$$
$$\mathbb{S}_\Xi\, WBasedGetInterestRate == \exists\ \Xi\, WBased \bullet \Phi\mathbb{S}\, WBasedO \land WBased_\Xi\, GetInterestRate$$
$$\mathbb{S}_\Xi\, WBasedGetNoWithdrawls == \exists\ \Xi\, WBased \bullet \Phi\mathbb{S}\, WBasedO \land WBased_\Xi\, GetNoWithdrawls$$

$\Phi\mathbb{S}\, WBasedDI_0$

$$\Phi\mathbb{S}SavingsDI_0[oWBased?/oSavings?]$$
$$\Delta\mathbb{S}\, WBased$$
$$WBased$$
$$oWBased? : \mathbb{O}\, WBasedCl$$

$$sWBased' = sWBased \setminus \{oWBased?\}$$
$$stWBased' = \{oWBased?\} \lhd stWBased$$

$\Phi\mathbb{S}\, WBasedDI$

$$\Phi\mathbb{S}\, WBasedDI_0$$

$$oWBased? \in sWBased \cap \mathbb{O}_x\, WBasedCl$$
$$\theta\, WBased = stWBased\ oWBased?$$

$$\mathbb{S}_\Delta\, WBasedDelete == \exists\, WBased \bullet \Phi\mathbb{S}\, WBasedDI \land WBasedFin$$

F.2.7 Class **BalBased**

Intensional view

BalBased
$Savings$

BalBasedInit$_I$
$SavingsInit_I$

BalBasedInit
$BalBased'$
$SavingsInit$
$BalBasedInit_I$

$opAmount? \geq reqOpAmount\ balbased$
$interestR? \leq maxInitInterest(balbased, balance')$

BalBased$_\Delta$ *Withdraw*
$\Delta BalBased$
$Savings_\Delta Withdraw$

$interestR' = interestR$
$interestDt' = interestDt$
$term' = term$

BalBased$_\Delta$ *Deposit*
$\Delta BalBased$
$Savings_\Delta Deposit$

$interestR' = interestR$
$interestDt' = interestDt$
$term' = term$

$BalBased_\Delta Suspend == [\ \Delta BalBased;\ Savings_\Delta Suspend\ ]$
$BalBased_\Delta ReActivate == [\ \Delta BalBased;\ Savings_\Delta ReActivate\ ]$

$calcInterestBal : (\mathbb{Z} \times PERCENTAGE) \rightarrow PERCENTAGE$

BalBased$_\Delta$ *PayInterest*
$\Delta BalBased$
$Savings_\Delta PayInterest$

$interestR' = calcInterestBal(balance, interestR)$

$BalBased_\Delta ChCreditTerm == \Delta BalBased \land Savings_\Delta ChCreditTerm$
$BalBased_\Xi GetBalance == \Xi BalBased \land Account_\Xi GetBalance$
$BalBased_\Xi GetInterestRate == \Xi BalBased \land Savings_\Xi GetInterestRate$

$$\begin{array}{|l}\hline BalBasedFin \\ SavingsFin \\ \hline\end{array}$$

Extensional view

$$\mathbb{S}BalBased == \mathbb{S}\mathrm{CL}[\mathbb{O}BalBasedCl, BalBased][sBalBased/os, stBalBased/oSt]$$

$$\mathbb{S}BalBasedInit == [\ \mathbb{S}BalBased\ ' \mid sBalBased' = \varnothing \land stBalBased' = \varnothing\ ]$$

$$\begin{array}{|l}\hline \mathbb{S}BalBasedIsSavings \\ \mathbb{S}Savings;\ \mathbb{S}BalBased \\ \hline sBalBased \subseteq sSavings \\ \forall\, oBalBased : sBalBased \bullet \\ \quad (\lambda\ BalBased \bullet \theta Savings)(stBalBased\ oBalBased) = stSavings\ oBalBased \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \Phi\mathbb{S}BalBasedNI_0 \\ \Phi\mathbb{S}SavingsNI_0[oBalBased!/oSavings!] \\ \Delta\mathbb{S}BalBased \\ BalBased\ ' \\ oBalBased! : \mathbb{O}BalBasedCl \\ \hline sBalBased' = sBalBased \cup \{oBalBased!\} \\ stBalBased' = \\ \quad stBalBased \cup \{oBalBased! \mapsto \theta BalBased\ '\} \\ \hline\end{array} \qquad \begin{array}{|l}\hline \Phi\mathbb{S}BalBasedNI \\ \Phi\mathbb{S}BalBasedNI_0 \\ \hline oBalBased! \in \mathbb{O}_x BalBasedCl \setminus sBalBased \\ \hline\end{array}$$

$$\mathbb{S}_\Delta BalBasedNew == \exists\ BalBased\ ' \bullet \Phi\mathbb{S}BalBasedNI \land BalBasedInit$$

$$\begin{array}{|l}\hline \Phi\mathbb{S}BalBasedUI_0 \\ \Phi\mathbb{S}SavingsUI_0[oBalBased?/oSavings?] \\ \Delta\mathbb{S}BalBased \\ \Delta BalBased \\ oBalBased? : \mathbb{O}BalBasedCl \\ \hline sBalBased' = sBalBased \\ stBalBased' = \\ \quad stBalBased \oplus \{oBalBased? \mapsto \theta BalBased\ '\} \\ \hline\end{array} \qquad \begin{array}{|l}\hline \Phi\mathbb{S}BalBasedUI \\ \Phi\mathbb{S}BalBasedUI_0 \\ \hline oBalBased? \in sBalBased \cap \mathbb{O}_x BalBasedCl \\ \theta BalBased = stBalBased\ oBalBased? \\ \hline\end{array}$$

$$\mathbb{S}_\Delta BalBasedWithdraw == \exists \Delta BalBased \bullet \Phi\mathbb{S}BalBasedUI \land BalBased_\Delta Withdraw$$

$$\mathbb{S}_\Delta BalBasedDeposit == \exists \Delta BalBased \bullet \Phi\mathbb{S}BalBasedUI \land BalBased_\Delta Deposit$$

$$\mathbb{S}_\Delta BalBasedPayInterest == \exists \Delta BalBased \bullet \Phi\mathbb{S}BalBasedUI \land BalBased_\Delta PayInterest$$

$$\mathbb{S}_\Delta BalBasedChCreditTerm == \exists \Delta BalBased \bullet \Phi\mathbb{S}BalBasedUI \land BalBased_\Delta ChCreditTerm$$

$$\mathbb{S}_\Delta BalBasedSuspend == \exists \Delta BalBased \bullet \Phi\mathbb{S}BalBasedUI \land BalBased_\Delta Suspend$$

$$\mathbb{S}_\Delta BalBasedReActivate == \exists \Delta BalBased \bullet \Phi\mathbb{S}BalBasedUI \land BalBased_\Delta ReActivate$$

$$\Phi\mathbb{S}BalBasedO$$
$$\Xi\mathbb{S}BalBased$$
$$\Xi BalBased$$
$$oBalBased? : \mathbb{O}BalBasedCl$$
$$oBalBased? \in sBalBased$$
$$\theta BalBased = stBalBased\ oBalBased?$$

$$\mathbb{S}_{\Xi}BalBasedGetBalance == \exists\ \Xi BalBased \bullet \Phi\mathbb{S}BalBasedO \land BalBased_{\Xi}GetBalance$$
$$\mathbb{S}_{\Xi}BalBasedGetInterestRate == \exists\ \Xi BalBased \bullet \Phi\mathbb{S}BalBasedO \land BalBased_{\Xi}GetInterestRate$$

$$\Phi\mathbb{S}BalBasedDI_0$$
$$\Phi\mathbb{S}SavingsDI_0[oBalBased?/oSavings?]$$
$$\Delta\mathbb{S}BalBased$$
$$BalBased$$
$$oBalBased? : \mathbb{O}BalBasedCl$$
$$sBalBased' = sBalBased \setminus \{oBalBased?\}$$
$$stBalBased' = \{oBalBased?\} \lhd stBalBased$$

$$\Phi\mathbb{S}BalBasedDI$$
$$\Phi\mathbb{S}BalBasedDI_0$$
$$oBalBased? \in sBalBased \cap \mathbb{O}_x BalBasedCl$$
$$\theta BalBased = stBalBased\ oBalBased?$$

$$\mathbb{S}_{\Delta}BalBasedDelete == \exists\ BalBased \bullet \Phi\mathbb{S}BalBasedDI \land BalBasedFin$$

F.2.8 Class **Savings**, Polymorphic definitions

$$CondNWBased == [\ accTy? : ACCTY \mid accTy? = wbased\ ]$$
$$CondNBalBased == [\ accTy? : ACCTY \mid accTy? = balbased\ ]$$
$$\mathbb{S}_{\Delta}SavingsNew == CondNWBased \land \mathbb{S}_{\Delta}WBasedNew[oSavings!/oWBased!] \land \Xi\mathbb{S}BalBased \lor CondNBalBased \land \mathbb{S}_{\Delta}BalBasedNew[oSavings!/oBalBased!] \land \Xi\mathbb{S}WBased$$
$$\mathbb{S}_{\Delta}SavingsWithdraw == \mathbb{S}_{\Delta}WBasedWithdraw[oSavings?/oWBased?] \land \Xi\mathbb{S}BalBased \lor \mathbb{S}_{\Delta}BalBasedWithdraw[oSavings?/oBalBased?] \land \Xi\mathbb{S}WBased$$
$$\mathbb{S}_{\Delta}SavingsDeposit == \mathbb{S}_{\Delta}WBasedDeposit[oSavings?/oWBased?] \land \Xi\mathbb{S}BalBased \lor \mathbb{S}_{\Delta}BalBasedDeposit[oSavings?/oBalBased?] \land \Xi\mathbb{S}WBased$$
$$\mathbb{S}_{\Delta}SavingsPayInterest == \mathbb{S}_{\Delta}WBasedPayInterest[oSavings?/oWBased?] \land \Xi\mathbb{S}BalBased \lor \mathbb{S}_{\Delta}BalBasedPayInterest[oSavings?/oBalBased?] \land \Xi\mathbb{S}WBased$$
$$\mathbb{S}_{\Delta}SavingsChCreditTerm == \mathbb{S}_{\Delta}WBasedChCreditTerm[oSavings?/oWBased?] \land \Xi\mathbb{S}BalBased \lor \mathbb{S}_{\Delta}BalBasedChCreditTerm[oSavings?/oBalBased?] \land \Xi\mathbb{S}WBased$$
$$\mathbb{S}_{\Delta}SavingsDelete == \mathbb{S}_{\Delta}WBasedDelete[oSavings?/oWBased?] \land \Xi\mathbb{S}BalBased \lor \mathbb{S}_{\Delta}BalBasedDelete[oSavings?/oBalBased?] \land \Xi\mathbb{S}WBased$$
$$\mathbb{S}_{\Delta}SavingsSuspend == \mathbb{S}_{\Delta}WBasedSuspend[oSavings?/oWBased?] \land \Xi\mathbb{S}BalBased \lor \mathbb{S}_{\Delta}BalBasedSuspend[oSavings?/oBalBased?] \land \Xi\mathbb{S}WBased$$
$$\mathbb{S}_{\Delta}SavingsReActivate == \mathbb{S}_{\Delta}WBasedReActivate[oSavings?/oWBased?] \land \Xi\mathbb{S}BalBased \lor \mathbb{S}_{\Delta}BalBasedReActivate[oSavings?/oBalBased?] \land \Xi\mathbb{S}WBased$$
$$\Psi\mathbb{S}SavingsI == \Xi\mathbb{S}Savings \land \Xi\mathbb{S}BalBased \land \Xi\mathbb{S}WBased$$

F.2.9 Class **Account**, Polymorphic definitions

$$CondNCurrent == [\ accTy? : ACCTY \mid accTy? = current\ ]$$

$$\mathbb{S}_\Delta AccountNew == CondNCurrent \land \mathbb{S}_\Delta CurrentNew[oAccount!/oCurrent!] \land \Psi\mathbb{S}SavingsI \lor \mathbb{S}_\Delta SavingsNew[oAccount!/oSavings!] \land \Xi\mathbb{S}Current$$

$$\mathbb{S}_\Delta AccountWithdraw == \mathbb{S}_\Delta CurrentWithdraw[oAccount?/oCurrent?] \land \Psi\mathbb{S}SavingsI \lor \mathbb{S}_\Delta SavingsWithdraw[oAccount?/oSavings?] \land \Xi\mathbb{S}Current$$

$$\mathbb{S}_\Delta AccountDeposit == \mathbb{S}_\Delta CurrentDeposit[oAccount?/oCurrent?] \land \Psi\mathbb{S}SavingsI \lor \mathbb{S}_\Delta SavingsDeposit[oAccount?/oSavings?] \land \Xi\mathbb{S}Current$$

$$\mathbb{S}_\Delta AccountDelete == \mathbb{S}_\Delta CurrentDelete[oAccount?/oCurrent?] \land \Psi\mathbb{S}SavingsI \lor \mathbb{S}_\Delta SavingsDelete[oAccount?/oSavings?] \land \Xi\mathbb{S}Current$$

$$\mathbb{S}_\Delta AccountSuspend == \mathbb{S}_\Delta CurrentSuspend[oAccount?/oCurrent?] \land \Psi\mathbb{S}SavingsI \lor \mathbb{S}_\Delta SavingsSuspend[oAccount?/oSavings?] \land \Xi\mathbb{S}Current$$

$$\mathbb{S}_\Delta AccountReActivate == \mathbb{S}_\Delta CurrentReActivate[oAccount?/oCurrent?] \land \Psi\mathbb{S}SavingsI \lor \mathbb{S}_\Delta SavingsReActivate[oAccount?/oSavings?] \land \Xi\mathbb{S}Current$$

$$\Psi\mathbb{S}AccountI == \Xi\mathbb{S}Account \land \Xi\mathbb{S}Current \land \Psi\mathbb{S}SavingsI$$

F.2.10 Association Holds

$\mathbb{A}Holds$

$$rHolds : \mathbb{O}CustomerCl \leftrightarrow \mathbb{O}AccountCl$$

$\mathbb{A}HoldsInit$

$$\mathbb{A}Holds'$$

$$rHolds' = \varnothing$$

$$\vdash \exists\, \mathbb{A}HoldsInit \bullet true$$

$\mathbb{A}_\Delta HoldsAdd$

$$\Delta\ \mathbb{A}Holds$$
$$oCustomer? : \mathbb{O}CustomerCl$$
$$oAccount? : \mathbb{O}AccountCl$$

$$rHolds' = rHolds \cup \{oCustomer? \mapsto oAccount?\}$$

$\mathbb{A}_\Delta HoldsDel$

$$\Delta\mathbb{A}Holds$$
$$oCustomer? : \mathbb{O}CustomerCl$$
$$oAccount? : \mathbb{O}AccountCl$$

$$rHolds' = rHolds \setminus \{oCustomer? \mapsto oAccount?\}$$

$\mathbb{A}_{\Delta} HoldsDelAccount$

$\Delta \mathbb{A} Holds$
$osAccount? : \mathbb{P}(\mathbb{O} AccountCl)$

$rHolds' = rHolds \rhd osAccount?$

$\mathbb{A}_{\Xi} CustomerAccounts$

$\Xi \mathbb{A} Holds$
$oCustomer? : \mathbb{O} CustomerCl$
$osAccount! : \mathbb{P}(\mathbb{O} AccountCl)$

$osAccount! = rHolds(\!| \{oCustomer?\} |\!)$

$\mathbb{A}_{\Xi} AccountCustomer$

$\Xi \mathbb{A} Holds$
$oAccount? : \mathbb{O} AccountCl$
$osCustomer! : \mathbb{P}(\mathbb{O} CustomerCl)$

$osCustomer! = rHolds^{\sim}(\!| \{oAccount?\} |\!)$

F.2.11 Global View

$Link\mathbb{A}Holds$

$\mathbb{S} Customer;\ \mathbb{S} Account;\ \mathbb{A} Holds$

$\mathsf{mult}(rHolds, sCustomer, sAccount, om, \{\}, \{\})$

$ConstSumBalsGEQZ$

$\mathbb{S} Account$

$\Sigma\{a : sAccount \bullet a \mapsto (stAccount\ a).balance\} \geq 0$

$ConstCompanyNoSavings$

$\mathbb{S} Customer;\ \mathbb{S} Savings;\ \mathbb{A} Holds$

$\{oC : sCustomer \mid (stCustomer\ oC).type = company\} \lhd rHolds \rhd sSavings = \varnothing$

$ConstOneCurrent$

$\mathbb{S} Customer;\ \mathbb{S} Current;\ \mathbb{A} Holds$

$\mathsf{mult}(rHolds \rhd sCustomer, sCustomer, sCurrent, ozo, \{\}, \{\})$

$SysConst == Link\mathbb{A}Holds \land ConstSumBalsGEQZ \land ConstCompanyNoSavings$
$\land\ ConstOneCurrent \land \mathbb{S} SavingsIsAccount \land \mathbb{S} CurrentIsAccount \land \mathbb{S} WBasedIsSavings$
$\land\ \mathbb{S} BalBasedIsSavings$

$System$

$\mathbb{S} Customer;\ \mathbb{S} Account;\ \mathbb{A} Holds;\ \mathbb{S} Current;\ \mathbb{S} Savings;\ \mathbb{S} WBased;\ \mathbb{S} BalBased$

$SysConst$

$$SysInit == System' \land \mathbb{S}CustomerInit \land \mathbb{S}AccountInit \land \mathbb{A}HoldsInit \\ \land \mathbb{S}CurrentInit \land \mathbb{S}SavingsInit \land \mathbb{S}WBasedInit \land \mathbb{S}BalBasedInit$$

$$\Psi Customer == \Delta System \land \Xi\mathbb{S}Account \land \Xi\mathbb{A}Holds$$

$$SysNewCustomer == \Psi Customer \land \mathbb{S}_\Delta CustomerNew$$

$$\Psi SysAccountHolds == \Delta System \land \Xi\mathbb{S}Customer$$

OpConstIfSavingsHasCurrent

$$\mathbb{S}Customer;\ \mathbb{S}Current$$
$$\mathbb{A}Holds$$
$$accTy? : ACCTY$$
$$oCustomer? : \mathbb{O}CustomerCl$$

$$\neg\ accTy? = current \Rightarrow oCustomer? \in rHolds^{\sim}(\!|\ sCurrent\ |\!)$$

$$\Psi OpenAccount == \Delta System \land \Xi\mathbb{S}Customer$$

$$SysOpenAccount == \Psi OpenAccount \land OpConstIfSavingsHasCurrent \\ \land \mathbb{S}_\Delta AccountNew \land \mathbb{A}_\Delta HoldsAdd[oAccount!/oAccount?]$$

$$\Psi Withdraw == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds$$

$$SysWithdraw == \Psi Withdraw \land \mathbb{S}_\Delta AccountWithdraw$$

$$\Psi Deposit == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds$$

$$SysDeposit == \Psi Deposit \land \mathbb{S}_\Delta AccountDeposit$$

$$\Psi Suspend == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds$$

$$SysSuspend == \Psi Suspend \land \mathbb{S}_\Delta AccountSuspend$$

$$\Psi ReActivate == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds$$

$$SysReActivate == \Psi ReActivate \land \mathbb{S}_\Delta AccountReActivate$$

$$\Psi SetOverdraft == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds \land \Xi\mathbb{S}Savings \land \Xi\mathbb{S}WBased \land \Xi\mathbb{S}BalBased$$

$$SysSetOverdraft == \Psi SetOverdraft \land \mathbb{S}_\Delta CurrentSetOverdraft$$

$$\Psi PayInterest == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds \land \Xi\mathbb{S}Current$$

$$SysPayInterest == \Psi PayInterest \land \mathbb{S}_\Delta SavingsPayInterest$$

$$\Psi ChangeCreditTerm == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds \land \Xi\mathbb{S}Current$$

$$SysChangeCreditTerm == \Psi ChangeCreditTerm \land \mathbb{S}_\Delta SavingsChCreditTerm$$

ConnAccountOs

$$osAccount? : \mathbb{P}(\mathbb{O}AccountCl)$$
$$oAccount? : \mathbb{O}AccountCl$$

$$osAccount? = \{oAccount?\}$$

$$\Psi DeleteAccount == \Delta System \land \Xi\mathbb{S}Customer$$

$$SysDeleteAccount == (\Psi DeleteAccount \land \mathbb{S}_\Delta AccountDelete \\ \land ConnAccountOs \land \mathbb{A}_\Delta HoldsDelAccount) \setminus (osAccount?)$$

$$SysDeleteCustomer == \Psi Customer \land \mathbb{S}_{\Delta} CustomerDelete$$
$$SysGetBalance == \Xi System \land \mathbb{S}_{\Xi} AccountGetBalance$$
$$SysGetCustAccounts == \Xi System \land \mathbb{A}_{\Xi} CustomerAccounts$$
$$SysGetDebtAccounts == \Xi System \land \mathbb{S}_{\Xi} AccountGetDebtAccounts$$

F.3 Inheritance Bank after snapshot analysis

The following presents the changes to the ZOO model of Bank with inheritance (see above) that result from the snapshot analysis performed in chapter 6. Those changes affect the global view of the model only.

F.3.1 Global View

Link𝔸Holds

$\mathbb{S}Customer;\ \mathbb{S}Account;\ \mathbb{A}Holds$

$\mathsf{mult}(rHolds, sCustomer, sAccount, om, \{\}, \{\})$

ConstSumBalsGEQZ

$\mathbb{S}Account$

$\Sigma\{a : sAccount \bullet a \mapsto (stAccount\ a).balance\} \geq 0$

ConstCompanyNoSavings

$\mathbb{S}Customer;\ \mathbb{S}Savings;\ \mathbb{A}Holds$

$\{oC : sCustomer \mid (stCustomer\ oC).type = company\} \lhd rHolds \rhd sSavings = \varnothing$

ConstOneCurrent

$\mathbb{S}Customer;\ \mathbb{S}Current;\ \mathbb{A}Holds$

$\mathsf{mult}(rHolds \rhd sCustomer, sCustomer, sCurrent, ozo, \{\}, \{\})$

ConstHasCurrent

$\mathbb{S}Customer;\ \mathbb{S}Current;\ \mathbb{A}Holds$

$\forall oC : sCustomer \mid oC \in \operatorname{dom} rHolds \bullet oC \in rHolds^{\sim}(\!|\ sCurrent\ |\!)$

$$SysConst == Link\mathbb{A}Holds \land ConstSumBalsGEQZ \land ConstCompanyNoSavings$$
$$\land\ ConstOneCurrent \land ConstHasCurrent$$
$$\land\ \mathbb{S}SavingsIsAccount \land \mathbb{S}CurrentIsAccount \land \mathbb{S}WBasedIsSavings$$
$$\land\ \mathbb{S}BalBasedIsSavings$$

System

$\mathbb{S}Customer;\ \mathbb{S}Account;\ \mathbb{A}Holds;\ \mathbb{S}Current;\ \mathbb{S}Savings;\ \mathbb{S}WBased;\ \mathbb{S}BalBased$

$SysConst$

$$SysInit == System' \land \mathbb{S}CustomerInit \land \mathbb{S}AccountInit \land \mathbb{A}HoldsInit \land \mathbb{S}CurrentInit \land \mathbb{S}SavingsInit \land \mathbb{S}WBasedInit \land \mathbb{S}BalBasedInit$$

$$\Psi Customer == \Delta System \land \Xi\mathbb{S}Account \land \Xi\mathbb{A}Holds$$
$$SysNewCustomer == \Psi Customer \land \mathbb{S}_\Delta CustomerNew$$
$$\Psi SysAccountHolds == \Delta System \land \Xi\mathbb{S}Customer$$

$$\Psi OpenAccount == \Delta System \land \Xi\mathbb{S}Customer$$
$$SysOpenAccount == \Psi OpenAccount \land \mathbb{S}_\Delta AccountNew \land \mathbb{A}_\Delta HoldsAdd[oAccount!/oAccount?]$$
$$\Psi Withdraw == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds$$
$$SysWithdraw == \Psi Withdraw \land \mathbb{S}_\Delta AccountWithdraw$$
$$\Psi Deposit == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds$$
$$SysDeposit == \Psi Deposit \land \mathbb{S}_\Delta AccountDeposit$$
$$\Psi Suspend == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds$$
$$SysSuspend == \Psi Suspend \land \mathbb{S}_\Delta AccountSuspend$$
$$\Psi ReActivate == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds$$
$$SysReActivate == \Psi ReActivate \land \mathbb{S}_\Delta AccountReActivate$$
$$\Psi SetOverdraft == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds \land \Xi\mathbb{S}Savings \land \Xi\mathbb{S}WBased \land \Xi\mathbb{S}BalBased$$
$$SysSetOverdraft == \Psi SetOverdraft \land \mathbb{S}_\Delta CurrentSetOverdraft$$
$$\Psi PayInterest == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds \land \Xi\mathbb{S}Current$$
$$SysPayInterest == \Psi PayInterest \land \mathbb{S}_\Delta SavingsPayInterest$$
$$\Psi ChangeCreditTerm == \Delta System \land \Xi\mathbb{S}Customer \land \Xi\mathbb{A}Holds \land \Xi\mathbb{S}Current$$
$$SysChangeCreditTerm == \Psi ChangeCreditTerm \land \mathbb{S}_\Delta SavingsChCreditTerm$$

ConnAccountOs

$osAccount? : \mathbb{P}(\mathbb{O}AccountCl)$
$oAccount? : \mathbb{O}AccountCl$

$osAccount? = \{oAccount?\}$

$$\Psi DeleteAccount == \Delta System \land \Xi\mathbb{S}Customer$$
$$SysDeleteAccount == (\Psi DeleteAccount \land \mathbb{S}_\Delta AccountDelete \land ConnAccountOs \land \mathbb{A}_\Delta HoldsDelAccount) \setminus (osAccount?)$$

$$SysDeleteCustomer == \Psi Customer \land \mathbb{S}_\Delta CustomerDelete$$
$$SysGetBalance == \Xi System \land \mathbb{S}_\Xi AccountGetBalance$$
$$SysGetCustAccounts == \Xi System \land \mathbb{A}_\Xi CustomerAccounts$$
$$SysGetDebtAccounts == \Xi System \land \mathbb{S}_\Xi AccountGetDebtAccounts$$

G Proofs

G.1 Proofs of behavioural Inheritance in ZOO

G.1.1 ADTs with no communication

Proof G.1 (Forwards initialisation, no communication)

$ci \subseteq ai \fatsemi r$
$\equiv$ [def of r]
$ci \subseteq ai \fatsemi f^\sim$
$\equiv$ [f is a total function; $r \subseteq s \fatsemi f^\sim \equiv r \fatsemi f \subseteq s$, when f is a total, see [WD96, p.281]]
$ci \fatsemi f \subseteq ai$
$\equiv$ [def of $\subseteq$]
$\forall\, g : G;\ a' : A' \bullet g \mapsto a' \in ci \fatsemi f \Rightarrow g \mapsto a' \in ai$
$\equiv$ [def of $\fatsemi$]
$\forall\, g : G;\ a' : A' \bullet (\exists\, c' : C' \bullet g \mapsto c' \in ci \land c' \mapsto a' \in f \Rightarrow) g \mapsto a' \in ai$
$\equiv$ [predicate calculus; f is total]
$\forall\, g : G;\ c' : C';\ a' : A' \bullet g \mapsto c' \in ci \land f(c') = a' \Rightarrow g \mapsto a' \in ai$
$\equiv$ [1-point-rule]
$\forall\, g : G;\ c' : C \bullet g \mapsto c' \in ci \land f(c') \in A' \Rightarrow g \mapsto f(c') \in ai$
$\equiv$ [Schema-calculus]
$\forall\, g : G;\ C' \bullet g \mapsto \theta C' \in ci \land f(\theta C') \in A' \Rightarrow g \mapsto f(\theta C') \in ai$
$\equiv$ [Def of f; func application]
$\forall\, g : G;\ C' \bullet g \mapsto \theta C' \in ci \land \theta A' \in A' \Rightarrow g \mapsto \theta A' \in ai$
$\equiv$ [removing true predicates; prop of $\times$; defs of ai and ci]
$\forall\, g : G;\ C' \bullet g \in G \land \theta C' \in \{CI \bullet \theta C'\} \Rightarrow g \in G \land \theta A' \in \{AI \bullet \theta A'\}$
$\equiv$ [removing true predicates; predicate calculus; by comprehension]
$\forall\, C' \bullet CI \Rightarrow AI$

■

Proof G.2 (Backwards initialisation, no communication)

$ci \fatsemi s \subseteq ai$

$\equiv$ [by defs of s and $\subseteq$]
$\forall g : G;\ a' : A' \bullet g \mapsto a' \in ci \fatsemi f \Rightarrow g \mapsto a' \in ai$

$\equiv$ [by def of $\fatsemi$]
$\forall g : G;\ a' : A' \bullet (\exists c' : C \bullet g \mapsto c' \in ci \land c' \mapsto a' \in f) \Rightarrow g \mapsto a' \in ai$

$\equiv$ [predicate calculus; f is total]
$\forall g : G;\ a' : A';\ c' : C' \bullet g \mapsto c' \in ci \land f(c') = a' \Rightarrow g \mapsto a' \in ai$

$\equiv$ [1-point-rule]
$\forall g : G;\ c' : C \bullet g \mapsto c' \in ci \land f(c') \in A' \Rightarrow g \mapsto f(c') \in ai$

$\equiv$ [Schema-calculus]
$\forall g : G;\ C' \bullet g \mapsto \theta C' \in ci \land f(\theta C') \in A' \Rightarrow g \mapsto f(\theta C') \in ai$

$\equiv$ [def of f and func-app]
$\forall g : G;\ C' \bullet g \mapsto \theta C' \in ci \land \theta A' \in A' \Rightarrow g \mapsto \theta A' \in ai$

$\equiv$ [removing true predicates;defs of ci adn ai; prop of $\times$]
$\forall g : G;\ C' \bullet g \in G \land \theta C' \in \{CI \bullet \theta C'\} \Rightarrow g \in G \land \theta A' \in \{AI \bullet \theta A'\}$

$\equiv$ [removing true preds; predicate calculus; comprehension]
$\forall C' \bullet CI \Rightarrow AI$

■

Proof G.3 (Forwards applicability, no communication)

$$
\begin{aligned}
& ran((\mathrm{dom}\ ao) \lhd r) \subseteq \mathrm{dom}\ co \\
\equiv\ & [\text{defs of } \subseteq \text{ and } r] \\
& \forall\, c : C \bullet c \in ran((\mathrm{dom}\ ao) \lhd f^{\sim}) \Rightarrow c \in \mathrm{dom}\ co \\
\equiv\ & [\text{def of ran; comprehension}] \\
& \forall\, c : C \bullet (\exists\, a : A \bullet a \mapsto c \in (\mathrm{dom}\ ao) \lhd f^{\sim}) \Rightarrow c \in \mathrm{dom}\ co \\
\equiv\ & [\text{prop of } \lhd] \\
& \forall\, c : C \bullet (\exists\, a : A \bullet a \in \mathrm{dom}\ ao \land a \mapsto c \in f^{\sim}) \Rightarrow c \in \mathrm{dom}\ co \\
\equiv\ & [\text{by def of } {}^{\sim}] \\
& \forall\, c : C \bullet \exists\, a : A \bullet a \in \mathrm{dom}\ ao \land c \mapsto a \in f \Rightarrow c \in \mathrm{dom}\ co \\
\equiv\ & [f \text{ is total}] \\
& \forall\, c : C \bullet \exists\, a : A \bullet a \in \mathrm{dom}\ ao \land a = f(c) \Rightarrow c \in \mathrm{dom}\ co \\
\equiv\ & [\text{1-point-rule; } f \text{ is total}] \\
& \forall\, c : C \bullet f(c) \in \mathrm{dom}\ ao \Rightarrow c \in \mathrm{dom}\ co \\
\equiv\ & [\text{schema calculus}] \\
& \forall\, C \bullet f(\theta C) \in \mathrm{dom}\ ao \land f(\theta C) \in A \Rightarrow \theta C \in \mathrm{dom}\ co \\
\equiv\ & [\text{def of f and func-app}] \\
& \forall\, C \bullet \theta A \in \mathrm{dom}\ ao \land \theta A \in A \Rightarrow \theta C \in \mathrm{dom}\ co \\
\equiv\ & [\text{removing true preds}] \\
& \forall\, C \bullet \theta A \in \mathrm{dom}\ ao \Rightarrow \theta C \in \mathrm{dom}\ co \\
\equiv\ & [\text{Introducing pre, and operation schemas}] \\
& \forall\, C \bullet \mathrm{pre}\ AO \Rightarrow \mathrm{pre}\ CO
\end{aligned}
$$

■

Proof G.4 (Backwards applicability, no communication)

$$
\begin{array}{l}
\overline{\mathrm{dom}\ co} \subseteq \mathrm{dom}(s \rhd (\mathrm{dom}\ ao)) \\
\equiv [\textsf{def of } \subseteq;\ s] \\
\forall\, c : C \bullet c \in \overline{\mathrm{dom}\ co} \Rightarrow c \in \mathrm{dom}(f \rhd (\mathrm{dom}\ ao)) \\
\equiv [\textsf{def of } \overline{\mathrm{dom}\ co}] \\
\forall\, c : C \bullet c \notin \mathrm{dom}\ co \Rightarrow c \in \mathrm{dom}(f \rhd (\mathrm{dom}\ ao)) \\
\equiv [\textsf{def of dom}] \\
\forall\, c : C \bullet c \notin \mathrm{dom}\ co \Rightarrow \exists\, a : A \bullet c \mapsto a \in f \rhd (\mathrm{dom}\ ao) \\
\equiv [\textsf{prop of } \rhd] \\
\forall\, c : C \bullet c \notin \mathrm{dom}\ co \Rightarrow \exists\, a : A \bullet c \mapsto a \in f \land a \notin \mathrm{dom}\ ao \\
\equiv [f \textsf{ is total; 1 point rule}] \\
\forall\, c : C \bullet c \notin \mathrm{dom}\ co \Rightarrow f(c) \in A \land f(c) \notin \mathrm{dom}\ ao \\
\equiv [\textsf{Schema calculus}] \\
\forall\, C \bullet \theta C \notin \mathrm{dom}\ co \Rightarrow f(\theta C) \in A \land f(\theta C) \notin \mathrm{dom}\ ao \\
\equiv [\textsf{def of } f;\ \textsf{func-app}] \\
\forall\, C \bullet \theta C \notin \mathrm{dom}\ co \Rightarrow \theta A \in A \land \theta A \notin \mathrm{dom}\ ao \\
\equiv [\textsf{removing true preds; predicate calculus}] \\
\forall\, C \bullet \theta A \in \mathrm{dom}\ ao \Rightarrow \theta C \in \mathrm{dom}\ co \\
\equiv [\textsf{pre and op schemas}] \\
\forall\, C \bullet \mathrm{pre}\ AO \Rightarrow \mathrm{pre}\ CO
\end{array}
$$

■

Proof G.5 (Non-blocking forwards correctness, no communication)

$(\mathrm{dom}\, ao \lhd r) \fatsemi co \subseteq ao \fatsemi r$

$\equiv$ [defs of $\subseteq$ and r]

$\forall a : A;\ c' : C' \bullet a \mapsto c' \in (\mathrm{dom}\, ao \lhd f^\sim) \fatsemi co \Rightarrow a \mapsto c' \in ao \fatsemi f^\sim$

$\equiv$ [def of $\fatsemi$]

$\forall a : A;\ c' : C' \bullet (\exists c : C \bullet a \mapsto c \in (\mathrm{dom}\, ao \lhd f^\sim) \land c \mapsto c' \in co \Rightarrow a \mapsto c' \in ao \fatsemi f^\sim)$

$\equiv$ [predicate calculus; prop $\lhd$]

$\forall a : A;\ c' : C';\ c : C \bullet a \in \mathrm{dom}\, ao \land a \mapsto c \in f^\sim \land c \mapsto c' \in co \Rightarrow a \mapsto c' \in ao \fatsemi f^\sim$

$\equiv$ [by prop $^\sim$]

$\forall a : A;\ c' : C';\ c : C \bullet a \in \mathrm{dom}\, ao \land c \mapsto a \in f \land c \mapsto c' \in co \Rightarrow a \mapsto c' \in ao \fatsemi f^\sim$

$\equiv$ [f is total]

$\forall a : A;\ c' : C';\ c : C \bullet a \in \mathrm{dom}\, ao \land f(c) = a \land c \mapsto c' \in co \Rightarrow a \mapsto c' \in ao \fatsemi f^\sim$

$\equiv$ [1-point-rule]

$\forall c' : C';\ c : C \bullet f(c) \in \mathrm{dom}\, ao \land f(c) \in A \land c \mapsto c' \in co \Rightarrow f(c) \mapsto c' \in ao \fatsemi f^\sim$

$\equiv$ [def of $\fatsemi$; f is total]

$\forall c' : C';\ c : C \bullet f(c) \in \mathrm{dom}\, ao \land c \mapsto c' \in co$
$\quad \Rightarrow (\exists a' : A' \bullet f(c) \mapsto a' \in ao \land a' \mapsto c' \in f^\sim)$

$\equiv$ [by prop of $^\sim$; f is total]

$\forall c' : C';\ c : C \bullet f(c) \in \mathrm{dom}\, ao \land c \mapsto c' \in co \Rightarrow (\exists a' : A' \bullet f(c) \mapsto a' \in ao \land f(c') = a')$

$\equiv$ [1-point-rule; f is total]

$\forall c' : C';\ c : C \bullet f(c) \in \mathrm{dom}\, ao \land c \mapsto c' \in co \Rightarrow f(c) \mapsto f(c') \in ao$

$\equiv$ [schema-calculus]

$\forall C';\ C \bullet f(\theta C) \in \mathrm{dom}\, ao \land \theta C \mapsto \theta C' \in co \Rightarrow f(\theta C) \mapsto f(\theta C') \in ao$

$\equiv$ [def of f; func-app]

$\forall C';\ C \bullet \theta A \in \mathrm{dom}\, ao \land \theta C \mapsto \theta C' \in co \Rightarrow \theta A \mapsto \theta A' \in ao$

$\equiv$ [removing true preds; introducing pre and Op schemas]

$\forall C';\ C \bullet \mathrm{pre}\, AO \land CO \Rightarrow AO$

■

Proof G.6 (Non-blocking backwards correctness, no communication)

$\mathrm{dom}(s \rhd (\mathrm{dom}\, ao)) \rhd\ co \fatsemi s \subseteq s \fatsemi ao$

$\equiv$ [defs of $\subseteq$ and s]
$\forall\, c : C;\ a' : A' \bullet c \mapsto a' \in \mathrm{dom}(f \rhd (\mathrm{dom}\, ao)) \rhd\ co \fatsemi f \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [prop of $\rhd$]
$\forall\, c : C;\ a' : A' \bullet c \mapsto a' \in co \fatsemi f \wedge c \notin \mathrm{dom}(f \rhd (\mathrm{dom}\, ao)) \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [defs of $\notin$ and dom]
$\forall\, c : C;\ a' : A' \bullet c \mapsto a' \in co \fatsemi f \wedge \neg\, (\exists\, a : A \bullet c \mapsto a \in f \rhd (\mathrm{dom}\, ao))$
$\quad \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [prop of $\rhd$; f is total]
$\forall\, c : C;\ a' : A' \bullet c \mapsto a' \in co \fatsemi f \wedge \neg\, (\exists\, a : A \bullet f(c) = a \wedge a \notin \mathrm{dom}\, ao)$
$\quad \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [1-point-rule]
$\forall\, c : C;\ a' : A' \bullet c \mapsto a' \in co \fatsemi f \wedge \neg\, (f(c) \in A \wedge f(c) \notin \mathrm{dom}\, ao) \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [f is total; predicate calculus]
$\forall\, c : C;\ a' : A' \bullet c \mapsto a' \in co \fatsemi f \wedge f(c) \in \mathrm{dom}\, ao \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [def of $\fatsemi$]
$\forall\, c : C;\ a' : A' \bullet (\exists\, c' : C' \bullet c \mapsto c' \in co \wedge c' \mapsto a' \in f \wedge f(c) \in \mathrm{dom}\, ao)$
$\quad \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [predicate calculus; f is total]
$\forall\, c : C;\ a' : A';\ c' : C' \bullet c \mapsto c' \in co \wedge f(c') = a' \wedge f(c) \in \mathrm{dom}\, ao \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [one-point-rule; f is total]
$\forall\, c : C;\ c' : C' \bullet c \mapsto c' \in co \wedge f(c) \in \mathrm{dom}\, ao \Rightarrow c \mapsto f(c') \in f \fatsemi ao$

$\equiv$ [def of $\fatsemi$]
$\forall\, c : C;\ c' : C' \bullet c \mapsto c' \in co \wedge f(c) \in \mathrm{dom}\, ao \Rightarrow (\exists\, a : A \bullet c \mapsto a \in f \wedge a \mapsto f(c') \in ao)$

$\equiv$ [f is total; one-point-rule]
$\forall\, c : C;\ c' : C' \bullet c \mapsto c' \in co \wedge f(c) \in \mathrm{dom}\, ao \Rightarrow f(c) \mapsto f(c') \in ao$

$\equiv$ [schema-calculus]
$\forall\, C;\ C' \bullet \theta C \mapsto \theta C' \in co \wedge f(\theta C) \in \mathrm{dom}\, ao \Rightarrow f(\theta C) \mapsto f(\theta C') \in ao$

$\equiv$ [def of f; func-app]
$\forall\, C;\ C' \bullet \theta C \mapsto \theta C' \in co \wedge \theta A \in \mathrm{dom}\, ao) \Rightarrow \theta A \mapsto \theta A' \in ao$

$\equiv$ [removing true preds; predicate calculus]
$\forall\, C;\ C' \bullet \theta C \mapsto \theta C' \in co \wedge \theta A \in \mathrm{dom}\, ao \Rightarrow \theta A \mapsto \theta A' \in ao$

$\equiv$ [pre and op schemas]
$\forall\, C;\ C' \bullet CO \wedge \mathrm{pre}\, AO \Rightarrow AO$

■

Proof G.7 (Blocking forwards correctness, no communication)

$r \fatsemi co \subseteq ao \fatsemi r$

$\equiv$ [defs of $\subseteq$ and r]
$\forall a : A;\ c' : C' \bullet a \mapsto c' \in f^{\sim} \fatsemi co \Rightarrow a \mapsto c' \in ao \fatsemi f^{\sim}$

$\equiv$ [by def $\fatsemi$]
$\forall a : A;\ c' : C' \bullet (\exists c : C \bullet a \mapsto c \in f^{\sim} \land c \mapsto c' \in co) \Rightarrow a \mapsto c' \in ao \fatsemi f^{\sim}$

$\equiv$ [by def $^{\sim}$; predicate calculus]
$\forall a : A;\ c, c' : C' \bullet c \mapsto a \in f \land c \mapsto c' \in co \Rightarrow a \mapsto c' \in ao \fatsemi f^{\sim}$

$\equiv$ [f is functional]
$\forall a : A;\ c, c' : C' \bullet f(c) = a \land c \mapsto c' \in co \Rightarrow a \mapsto c' \in ao \fatsemi f^{\sim}$

$\equiv$ [one-point-rule; f is total]
$\forall c, c' : C \bullet c \mapsto c' \in co \Rightarrow f(c) \mapsto c' \in ao \fatsemi f^{\sim}$

$\equiv$ [def of $\fatsemi$]
$\forall c, c' : C \bullet c \mapsto c' \in co \Rightarrow \exists a' : A \bullet f(c) \mapsto a' \in ao \land a' \mapsto c' \in f^{\sim}$

$\equiv$ [def of $^{\sim}$, and f is functional]
$\forall a : A;\ c, c' : C \bullet c \mapsto c' \in co \Rightarrow \exists a' : A \bullet f(c) \mapsto a' \in ao \land f(c') = a'$

$\equiv$ [one-point-rule; f is total]
$\forall c, c' : C \bullet c \mapsto c' \in co \Rightarrow f(c) \mapsto f(c') \in ao$

$\equiv$ [Schema-Calulcus]
$\forall C;\ C' \bullet \theta C \mapsto \theta C' \in co \Rightarrow f(\theta C) \mapsto f(\theta C') \in ao$

$\equiv$ [fun-app and def of f]
$\forall C' \bullet \exists C \bullet \theta C \mapsto \theta C' \in co \Rightarrow \theta A \mapsto \theta A' \in ao$

$\equiv$ [removing true preds and introducing op schemas]
$\forall C;\ C' \bullet CO \Rightarrow AO$

■

Proof G.8 (Blocking backwards correctness, no communication)

$co \fatsemi s \subseteq s \fatsemi ao$

$\equiv$ [defs of $\subseteq$ and s]
$\forall\, c : C;\ a' : A \bullet c \mapsto a' \in co \fatsemi f \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [def of $\fatsemi$]
$\forall\, c : C;\ a' : A \bullet (\exists\, c' : C \bullet c \mapsto c' \in co \land c' \mapsto a' \in f) \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [Predicate calculus; f is a total function]
$\forall\, c, c' : C;\ a' : A \bullet c \mapsto c' \in co \land f(c') = a' \Rightarrow c \mapsto a' \in f \fatsemi ao$

$\equiv$ [$\forall$ one-point rule; f is total]
$\forall\, c, c' : C \bullet c \mapsto c' \in co \Rightarrow c \mapsto f(c') \in f \fatsemi ao$

$\equiv$ [def of $\fatsemi$]
$\forall\, c, c' : C \bullet c \mapsto c' \in co \Rightarrow \exists\, a : A \bullet c \mapsto a \in f \land a \mapsto f(c') \in ao$

$\equiv$ [f is function]
$\forall\, c, c' : C \bullet c \mapsto c' \in co \Rightarrow \exists\, a : A \bullet f(c) = a \land a \mapsto f(c') \in ao$

$\equiv$ [$\exists$ one-point rule; f is total]
$\forall\, c, c' : C \bullet c \mapsto c' \in co \Rightarrow f(c) \mapsto f(c') \in ao$

$\equiv$ [schema-calculus]
$\forall\, C;\ C' \bullet \theta C \mapsto \theta C' \in co \Rightarrow f(\theta C) \mapsto f(\theta C') \in ao$

$\equiv$ [func-app and def of f]
$\forall\, C;\ C' \bullet \theta C \mapsto \theta C' \in co \Rightarrow \theta A \mapsto \theta A' \in ao$

$\equiv$ [removing true preds and introducing op schemas]
$\forall\, C;\ C' \bullet CO \Rightarrow AO$

■

Proof G.9 (Forwards finalisation, no communication)

$r \fatsemi cf \subseteq af$

$\equiv$ [defs of $\subseteq$ and r]
$\forall\, a : A;\ g : G \bullet a \mapsto g \in f^{\sim} \fatsemi cf \Rightarrow a \mapsto g \in af$

$\equiv$ [def of $\fatsemi$]
$\forall\, a : A;\ g : G \bullet \exists\, c : C \bullet a \mapsto c \in f^{\sim} \land c \mapsto g \in cf \Rightarrow a \mapsto g \in af$

$\equiv$ [prop of $^{\sim}$; f is total; predicate calculus]
$\forall\, a : A;\ g : G;\ c : C \bullet f(c) = a \land c \mapsto g \in cf \Rightarrow a \mapsto g \in af$

$\equiv$ [l-point-rule]
$\forall\, g : G;\ c : C \bullet f(c) \in A \land c \mapsto g \in cf \Rightarrow f(c) \mapsto g \in af$

$\equiv$ [schema-calculus]
$\forall\, g : G;\ C \bullet f(\theta C) \in A \land \theta C \mapsto g \in cf \Rightarrow f(\theta C) \mapsto g \in af$

$\equiv$ [def of f; func-app]
$\forall\, g : G;\ C \bullet \theta A \in A \land \theta C \mapsto g \in cf \Rightarrow \theta A \mapsto g \in af$

$\equiv$ [removing true preds; defs of cf and af]
$\forall\, g : G;\ C \bullet \theta C \in \{CF \bullet \theta C\} \land g \in G \Rightarrow \theta A \in \{AF \bullet \theta A\} \land g \in G$

$\equiv$ [removing true preds; by compreension]
$CF \Rightarrow AF$

If the finalisations are total; the last derivation reduces to true:

$\forall g : G;\ C \bullet \theta C \in \{CF \bullet \theta C\} \land g \in G \Rightarrow \theta A \in \{AF \bullet \theta A\} \land g \in G$
$\equiv$ [removing true preds; finalisations are total]
true

■

Proof G.10 (Backwards finalisation, no communication)

$cf \subseteq f \fatsemi af$
$\equiv$ [def of $\subseteq$]
$\forall c : C;\ g : G \bullet c \mapsto g \in cf \Rightarrow c \mapsto g \in f \fatsemi af$
$\equiv$ [def of $\fatsemi$]
$\forall c : C;\ g : G \bullet c \mapsto g \in cf \Rightarrow (\exists a : A \bullet c \mapsto a \in f \land a \mapsto g \in af)$
$\equiv$ [f is total]
$\forall c : C;\ g : G \bullet c \mapsto g \in cf \Rightarrow (\exists a : A \bullet f(c) = a \land a \mapsto g \in af)$
$\equiv$ [1-point rule]
$\forall c : C;\ g : G \bullet c \mapsto g \in cf \Rightarrow f(c) \in A \land f(c) \mapsto g \in af$
$\equiv$ [Schema calculus]
$\forall C;\ g : G \bullet \theta C \mapsto g \in cf \Rightarrow f(\theta C) \in A \land f(\theta c) \mapsto g \in af$
$\equiv$ [def of f and func-app]
$\forall C;\ g : G \bullet \theta C \mapsto g \in cf \Rightarrow \theta A \in A \land \theta A \mapsto g \in af$
$\equiv$ [removing true preds; defs of cf and af; prop of $\times$]
$\forall C;\ g : G \bullet \theta C \in \{CF \bullet \theta C\} \land g \in G \Rightarrow \theta A \in \{AF \bullet \theta A\} \land g \in G$
$\equiv$ [removing true preds; by comprehension]
$CF \Rightarrow AF$

If the finalisation sets are total, this rule reduces to:

true

■

G.1.2 ADTs with communication

Proof G.11 (Forwards applicability with communication)

$$
\begin{array}{l}
ran((\mathrm{dom}\ ao) \lhd (r \parallel id[I])) \subseteq \mathrm{dom}\ co \\
\equiv [\mathsf{defs\ of}\ \subseteq\ \mathsf{and}\ r] \\
\forall\, c : C;\ i? : I \bullet (c, i?) \in ran((\mathrm{dom}\ ao) \lhd (f^{\sim} \parallel id[I])) \Rightarrow (c, i?) \in \mathrm{dom}\ co \\
\equiv [\mathsf{def\ of}\ \mathrm{ran}] \\
\forall\, c : C;\ i? : I \bullet (\exists\, a : A;\ i' : I \bullet (a, i') \mapsto (c, i?) \in (\mathrm{dom}\ ao) \lhd (f^{\sim} \parallel id[I])) \\
\qquad \Rightarrow (c, i?) \in \mathrm{dom}\ co \\
\equiv [\mathsf{prop\ of}\ \lhd;\ \mathsf{def\ of}\ \parallel] \\
\forall\, c : C;\ i? : I \bullet (\exists\, a : A;\ i' : I \bullet (a, i') \in \mathrm{dom}\ ao \land (a, c) \in f^{\sim} \land (i', i?) \in id[I]) \\
\qquad \Rightarrow (c, i?) \in \mathrm{dom}\ co \\
\equiv [\mathsf{prop\ of}\ \mathrm{id};\ \mathsf{1\text{-}point\text{-}rule\ in}\ i';\ \mathsf{def\ of}\ {}^{\sim};\ \mathsf{f\ is\ total}] \\
\forall\, c : C;\ i? : I \bullet (\exists\, a : A \bullet (a, i?) \in \mathrm{dom}\ ao \land f(c) = a) \Rightarrow (c, i?) \in \mathrm{dom}\ co \\
\equiv [\mathsf{1\text{-}point\text{-}rule};\ f\ \mathsf{is\ total}] \\
\forall\, c : C;\ i? : I \bullet (f(c), i?) \in \mathrm{dom}\ ao \Rightarrow (c, i?) \in \mathrm{dom}\ co \\
\equiv [\mathsf{Schema\text{-}calculus};\ \mathsf{def\ of}\ f;\ \mathsf{func\text{-}app}] \\
\forall\, C;\ i? : I \bullet (\theta A, i?) \in \mathrm{dom}\ ao \Rightarrow (\theta C, i?) \in \mathrm{dom}\ co \\
\equiv [\mathsf{remove\ true\ preds};\ \mathsf{introducing\ pre\ and\ op\ schemas}] \\
\forall\, C;\ i? : I \bullet \mathrm{pre}\ AO \Rightarrow \mathrm{pre}\ CO
\end{array}
$$

■

Proof G.12 (Backwards applicability with communication)

$$
\begin{array}{l}
\overline{\mathrm{dom}\ co} \subseteq \mathrm{dom}(s \parallel id[I] \rhd (\mathrm{dom}\ ao)) \\
\equiv [\text{defs of } \subseteq \text{ and } s] \\
\forall\, c : C;\ i? : I \bullet (c, i?) \in \overline{\mathrm{dom}\ co} \Rightarrow (c, i?) \in \mathrm{dom}(f \parallel id[I] \rhd (\mathrm{dom}\ ao)) \\
\equiv [\text{def of dom}] \\
\forall\, c : C;\ i? : I \bullet (c, i?) \notin \mathrm{dom}\ co \Rightarrow \\
\qquad (\exists\, a : A;\ i' : I \bullet (c, i?) \mapsto (a, i') \in f \parallel id[I] \rhd (\mathrm{dom}\ ao)\ ) \\
\equiv [\text{prop of } \rhd] \\
\forall\, c : C;\ i? : I \bullet (c, i?) \notin \mathrm{dom}\ co \Rightarrow \\
\qquad (\exists\, a : A;\ i' : I \bullet (c, i?) \mapsto (a, i') \in f \parallel id[I] \wedge (a, i') \notin \mathrm{dom}\ ao\ ) \\
\equiv [\text{def of } \|] \\
\forall\, c : C;\ i? : I \bullet (c, i?) \notin \mathrm{dom}\ co \Rightarrow \\
\qquad (\exists\, a : A;\ i' : I \bullet c \mapsto a \in f \wedge i? \mapsto i' \in id[I] \wedge (a, i') \notin \mathrm{dom}\ ao\ ) \\
\equiv [f \text{ is total; def of } id\text{; one-point-rule; remove true preds}] \\
\forall\, c : C;\ i? : I \bullet (c, i?) \notin \mathrm{dom}\ co \Rightarrow (f(c), i?) \notin \mathrm{dom}\ ao \\
\equiv [\text{Schema calculus}] \\
\forall\, C;\ i? : I \bullet (\theta C, i?) \notin \mathrm{dom}\ co \Rightarrow (f(\theta C), i?) \notin \mathrm{dom}\ ao \\
\equiv [\text{def of } f\text{; func-app}] \\
\forall\, C;\ i? : I \bullet (\theta C, i?) \notin \mathrm{dom}\ co \Rightarrow (\theta A, i?) \notin \mathrm{dom}\ ao \\
\equiv [\text{remove true preds; def of } \notin\text{; predicate calculus}] \\
\forall\, C;\ i? : I \bullet (\theta A, i?) \in \mathrm{dom}\ ao \Rightarrow (\theta C, i?) \in \mathrm{dom}\ co \\
\equiv [\text{pre and op schemas}] \\
\forall\, C;\ i? : I \bullet \mathrm{pre}\ AO \Rightarrow \mathrm{pre}\ CO
\end{array}
$$

■

Proof G.13 (Non-blocking forwards correctness with communication)

$$
\begin{array}{l}
(\mathrm{dom}\, ao) \lhd (r \parallel id[I]);\ co \subseteq ao \fatsemi (r \parallel id[O]) \\
\equiv [\text{defs of } \subseteq \text{ and } r] \\
\forall\, a : A;\ c' : C';\ i? : I;\ o! : O \bullet (a, i?) \mapsto (c', o!) \in (\mathrm{dom}\, ao) \lhd (f^\sim \parallel id[I]);\ co \\
\quad \Rightarrow (a, i?) \mapsto (c', o!) \in ao \fatsemi (f^\sim \parallel id[O]) \\
\equiv [\text{def of } \fatsemi] \\
\forall\, a : A;\ c' : C';\ i? : I;\ o! : O \bullet (\exists\, c : C \bullet \\
\quad (a, i?) \mapsto (c, i?) \in (\mathrm{dom}\, ao) \lhd (f^\sim \parallel id[I]) \land (c, i?) \mapsto (c', o!) \in co \\
\quad\quad \Rightarrow (a, i?) \mapsto (c', o!) \in ao \fatsemi (f^\sim \parallel id[O])\ ) \\
\equiv [\text{predicate calculus; prop of } \lhd \text{; def of } \parallel] \\
\forall\, a : A;\ c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (a, i?) \in \mathrm{dom}\, ao \land (a, c) \in f^\sim \land (i?, i?) \in id[I] \land (c, i?) \mapsto (c', o!) \in co \\
\quad\quad \Rightarrow (a, i?) \mapsto (c', o!) \in ao \fatsemi (f^\sim \parallel id[O]) \\
\equiv [\text{remove true predicates; def of } \sim \text{; f is total}] \\
\forall\, a : A;\ c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (a, i?) \in \mathrm{dom}\, ao \land f(c) = a \land (c, i?) \mapsto (c', o!) \in co \\
\quad\quad \Rightarrow (a, i?) \mapsto (c', o!) \in ao \fatsemi (f^\sim \parallel id[O]) \\
\equiv [\text{1-point-rule; } f \text{ is total}] \\
\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (f(c), i?) \in \mathrm{dom}\, ao \land (c, i?) \mapsto (c', o!) \in co \\
\quad\quad \Rightarrow (f(c), i?) \mapsto (c', o!) \in ao \fatsemi (f^\sim \parallel id[O]) \\
\equiv [\text{def of } \fatsemi] \\
\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (f(c), i?) \in \mathrm{dom}\, ao \land (c, i?) \mapsto (c', o!) \in co \\
\quad\quad \Rightarrow (\exists\, a' : A' \bullet (f(c), i?) \mapsto (a', o!) \in ao \land (a', o!) \mapsto (c', o!) \in f^\sim \parallel id[O]\ ) \\
\equiv [\text{def of } \parallel \text{; prop of } id \text{; def of } \sim \text{; } f \text{ is total}] \\
\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (f(c), i?) \in \mathrm{dom}\, ao \land (c, i?) \mapsto (c', o!) \in co \\
\quad\quad \Rightarrow (\exists\, a' : A' \bullet (f(c), i?) \mapsto (a', o!) \in ao \land f(c') = a'\ ) \\
\equiv [\text{1-point-rule}] \\
\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (f(c), i?) \in \mathrm{dom}\, ao \land (c, i?) \mapsto (c', o!) \in co \\
\quad\quad \Rightarrow (f(c), i?) \mapsto (f(c'), o!) \in ao \\
\equiv [\text{Schema-calculus; def of } f \text{; func-app}] \\
\forall\, C;\ C';\ i? : I;\ o! : O \bullet \\
\quad (\theta A, i?) \in \mathrm{dom}\, ao \land (\theta C, i?) \mapsto (\theta C', o!) \in co \\
\quad\quad \Rightarrow (\theta A, i?) \mapsto (\theta A', o!) \in ao \\
\equiv [\text{remove true preds; introducing pre and op schemas}] \\
\forall\, C;\ C';\ i? : I;\ o! : O \bullet \mathrm{pre}\, AO \land CO \Rightarrow AO
\end{array}
$$

■

Proof G.14 (Non-blocking backwards correctness with communication)

$$
\begin{aligned}
&\mathrm{dom}((s \parallel id[I]) \rhd (\mathrm{dom}\, ao)) \rhd \ co \fatsemi (s \parallel id[O]) \subseteq (s \parallel id[I]) \fatsemi ao \\
&\equiv [\text{defs of } \subseteq \text{ and } s] \\
&\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet \\
&\qquad (c, i?) \mapsto (a', o!) \in \mathrm{dom}((f \parallel id[I]) \rhd (\mathrm{dom}\, ao)) \rhd \ co \fatsemi (f \parallel id[O]) \\
&\qquad\qquad \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [\text{def of } \rhd] \\
&\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet \\
&\qquad (c, i?) \mapsto (a', o!) \in co \fatsemi (f \parallel id[O]) \wedge (c, i?) \notin \mathrm{dom}((f \parallel id[I]) \rhd (\mathrm{dom}\, ao)) \\
&\qquad\qquad \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [\text{defs of } \notin \text{ and dom}] \\
&\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (a', o!) \in co \fatsemi (f \parallel id[O]) \\
&\qquad \wedge \neg\, (\exists\, a : A;\ i' : I \bullet (c, i?) \mapsto (a, i') \in (f \parallel id[I]) \rhd (\mathrm{dom}\, ao)) \\
&\qquad\qquad \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [\text{prop of } \rhd\text{; def of } \parallel] \\
&\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (a', o!) \in co \fatsemi (f \parallel id[O]) \\
&\qquad \wedge \neg\, (\exists\, a : A;\ i' : I \bullet c \mapsto a \in f \wedge i? \mapsto i' \in id[I] \wedge (a, i') \notin \mathrm{dom}\, ao) \\
&\qquad\qquad \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [f \text{ is total; prop of } id\text{; 1-point rule}] \\
&\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (a', o!) \in co \fatsemi (f \parallel id[O]) \\
&\qquad \wedge \neg\, (i? \in I \wedge (f(c), i?) \notin \mathrm{dom}\, ao) \\
&\qquad\qquad \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [\text{remove true preds; def of } \notin\text{; predicate calculus}] \\
&\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (a', o!) \in co \fatsemi (f \parallel id[O]) \\
&\qquad \wedge (f(c), i?) \in \mathrm{dom}\, ao \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [\text{def of } \fatsemi] \\
&\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet \\
&\qquad (\exists\, c' : C';\ o' : O \bullet (c, i?) \mapsto (c', o') \in co \wedge (c', o') \mapsto (a', o!) \in (f \parallel id[O])\ ) \\
&\qquad\qquad \wedge (f(c), i?) \in \mathrm{dom}\, ao \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [\text{def of } \parallel\text{; f is total; prop of } id\text{; 1 point rule}] \\
&\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet \\
&\qquad (\exists\, c' : C' \bullet (c, i?) \mapsto (c', o!) \in co \wedge f(c') = a'\ ) \\
&\qquad\qquad \wedge (f(c) \in A \Rightarrow (f(c), i?) \in \mathrm{dom}\, ao) \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [\text{predicate calculus; 1-point-rule; } f \text{ is total}] \\
&\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
&\qquad (c, i?) \mapsto (c', o!) \in co \wedge (f(c), i?) \in \mathrm{dom}\, ao \\
&\qquad\qquad \Rightarrow (c, i?) \mapsto (f(c'), o!) \in (f \parallel id[I]) \fatsemi ao \\
&\equiv [\text{def of } \fatsemi\text{; def of } \parallel\text{; } f \text{ is total; prop of } id] \\
&\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
&\qquad (c, i?) \mapsto (c', o!) \in co \wedge (f(c), i?) \in \mathrm{dom}\, ao \\
&\qquad\qquad \Rightarrow (\exists\, a : A;\ i' : I \bullet f(c) = a \wedge i? = i' \wedge (a, i') \mapsto (f(c'), o!) \in ao\ )
\end{aligned}
$$

$\equiv$ [1-point-rule; f is total]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet$
$\quad (c, i?) \mapsto (c', o!) \in co \land (f(c), i?) \in \operatorname{dom} ao \Rightarrow (f(c), i?) \mapsto (f(c'), o!) \in ao$

$\equiv$ [schema-calculus]
$\forall\, C;\ C';\ i? : I;\ o! : O \bullet$
$\quad (\theta C, i?) \mapsto (\theta C', o!) \in co (f(\theta C), i?) \in \operatorname{dom} ao \Rightarrow (f(\theta C), i?) \mapsto (f(\theta C'), o!) \in ao$

$\equiv$ [def of f; func-app]
$\forall\, C;\ C';\ i? : I;\ o! : O \bullet$
$\quad (\theta C, i?) \mapsto (\theta C', o!) \in co (\theta A, i?) \in \operatorname{dom} ao \Rightarrow (\theta A, i?) \mapsto (\theta A', o!) \in ao$

$\equiv$ [remove true preds; predicate calculus; pre and op schemas]
$\forall\, C;\ C';\ i? : I;\ o! : O \bullet CO \land \operatorname{pre} AO \Rightarrow AO$

■

Proof G.15 (Blocking forwards correctness with communication)

$$
\begin{array}{l}
(r \parallel \mathrm{id}[I]) \fatsemi co \subseteq ao \fatsemi (r \parallel \mathrm{id}[O]) \\
\equiv [\text{defs of } \subseteq \text{ and } r] \\
\forall\, a : A;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (a, i?) \mapsto (c', o!) \in (f^{\sim} \parallel \mathrm{id}[I]) \fatsemi co \Rightarrow (a, i?) \mapsto (c', o!) \in ao \fatsemi (f^{\sim} \parallel \mathrm{id}[O]) \\
\equiv [\text{def of } \fatsemi] \\
\forall\, a : A;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (\exists\, c : C;\ i' : I \bullet (a, i?) \mapsto (c, i') \in f^{\sim} \parallel \mathrm{id}[I] \land (c, i') \mapsto (c', o!) \in co) \\
\qquad \Rightarrow (a, i?) \mapsto (c', o!) \in ao \fatsemi (f^{\sim} \parallel \mathrm{id}[O]) \\
\equiv [\text{def of } \parallel;\ \text{prop of } {}^{\sim};\ \text{prop of id}] \\
\forall\, a : A;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (\exists\, c : C;\ i' : I \bullet a \mapsto c \in f \land i? = i' \land (c, i') \mapsto (c', o!) \in co) \\
\qquad \Rightarrow (a, i?) \mapsto (c', o!) \in ao \fatsemi (f^{\sim} \parallel \mathrm{id}[O]) \\
\equiv [\text{1-point-rule; remove true preds; predicate calculus; } f \text{ is total}] \\
\forall\, a : A;\ c' : C';\ c : C;\ i? : I;\ o! : O \bullet \\
\quad f(c) = a \land (c, i?) \mapsto (c', o!) \in co \Rightarrow (a, i?) \mapsto (c', o!) \in ao \fatsemi (f^{\sim} \parallel \mathrm{id}[O]) \\
\equiv [\text{1-point-rule; } f \text{ is total}] \\
\forall\, c' : C';\ c : C;\ i? : I;\ o! : O \bullet \\
\quad (c, i?) \mapsto (c', o!) \in co \Rightarrow (f(c), i?) \mapsto (c', o!) \in ao \fatsemi (f^{\sim} \parallel \mathrm{id}[O]) \\
\equiv [\text{def of } \fatsemi] \\
\forall\, c' : C';\ c : C;\ i? : I;\ o! : O \bullet \\
\quad (c, i?) \mapsto (c', o!) \in co \\
\quad \Rightarrow (\exists\, a' : A';\ o' : O \bullet (f(c), i?) \mapsto (a', o') \in ao \land (a', o') \mapsto (c', o!) \in (f^{\sim} \parallel \mathrm{id}[O])\ ) \\
\equiv [\text{defs of } \parallel \text{ and } {}^{\sim};\ f \text{ is total; prop of id}] \\
\forall\, c' : C';\ c : C;\ i? : I;\ o! : O \bullet \\
\quad (c, i?) \mapsto (c', o!) \in co \\
\quad \Rightarrow (\exists\, a' : A';\ o' : O \bullet (f(c), i?) \mapsto (a', o') \in ao \land f(c') = a' \land o' = o!\ ) \\
\equiv [\text{1-point-rule; } f \text{ is total; removing true preds}] \\
\forall\, c' : C';\ c : C;\ i? : I;\ o! : O \bullet \\
\quad (c, i?) \mapsto (c', o!) \in co \Rightarrow (f(c), i?) \mapsto (f(c'), o!) \in ao \\
\equiv [\text{Schema Calulus}] \\
\forall\, C, C';\ i? : I;\ o! : O \bullet \\
\quad (\theta C, i?) \mapsto (\theta C', o!) \in co \Rightarrow (f(\theta C), i?) \mapsto (f(\theta C'), o!) \in ao \\
\equiv [\text{def of } f;\ \text{func-app}] \\
\forall\, C, C';\ i? : I;\ o! : O \bullet \\
\quad (\theta C, i?) \mapsto (\theta C', o!) \in co \Rightarrow (\theta A, i?) \mapsto (\theta A', o!) \in ao \\
\equiv [\text{op-schemas}] \\
\forall\, C, C';\ i? : I;\ o! : O \bullet CO \Rightarrow AO
\end{array}
$$

■

Proof G.16 (Blocking backwards correctness with communication)

$$
\begin{array}{l}
co \fatsemi (s \parallel \mathrm{id}[O]) \subseteq (s \parallel \mathrm{id}[I]) \fatsemi ao \\
\equiv [\text{def of } \subseteq;\ \text{def of } s] \\
\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet \\
\quad (c, i?) \mapsto (a', o!) \in co \fatsemi (f \parallel \mathrm{id}[O]) \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel \mathrm{id}[I]) \fatsemi ao \\
\equiv [\text{def of } \fatsemi] \\
\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet \\
\quad (\exists\, c' : C';\ o' : O \bullet (c, i?) \mapsto (c', o') \in co \land (c', o') \mapsto (a', o!) \in f \parallel \mathrm{id}[O]) \\
\qquad \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel \mathrm{id}[I]) \fatsemi ao \\
\equiv [\text{def of } \parallel;\ \text{prop of } \mathrm{id}] \\
\forall\, c : C;\ a' : A';\ i? : I;\ o! : O \bullet \\
\quad (\exists\, c' : C';\ o' : O \bullet (c, i?) \mapsto (c', o') \in co \land c' \mapsto a' \in f \land o' = o!) \\
\qquad \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel \mathrm{id}[I]) \fatsemi ao \\
\equiv [\text{1-point-rule; predicate calculus; } f \text{ is total}] \\
\forall\, c : C;\ a' : A';\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (c, i?) \mapsto (c', o!) \in co \land f(c') = a' \Rightarrow (c, i?) \mapsto (a', o!) \in (f \parallel \mathrm{id}[I]) \fatsemi ao \\
\equiv [\text{1-point-rule; } f \text{ is total}] \\
\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \\
\quad (c, i?) \mapsto (c', o!) \in co \Rightarrow (c, i?) \mapsto (f(c'), o!) \in (f \parallel \mathrm{id}[I]) \fatsemi ao \\
\equiv [\text{def of } \fatsemi] \\
\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co \\
\quad \Rightarrow (\exists\, a : A;\ i' : I \bullet (c, i?) \mapsto (a, i') \in f \parallel \mathrm{id}[I] \land (a, i') \mapsto (f(c'), o!) \in ao\ ) \\
\equiv [\text{def of } \parallel;\ f \text{ is total; prop of } \mathrm{id}] \\
\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co \\
\quad \Rightarrow (\exists\, a : A;\ i' : I \bullet f(c) = a \land i? = i' \land (a, i') \mapsto (f(c'), o!) \in ao\ ) \\
\equiv [\text{1-point-rule; } f \text{ is total}] \\
\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co \\
\quad \Rightarrow (f(c), i?) \mapsto (f(c'), o!) \in ao \\
\equiv [\text{schema-calculus}] \\
\forall\, C;\ C';\ i? : I;\ o! : O \bullet (\theta C, i?) \mapsto (\theta C', o!) \in co \\
\quad \Rightarrow (f(\theta C), i?) \mapsto (f(\theta C'), o!) \in ao \\
\equiv [\text{def of } f;\ \text{func-app; remove true preds}] \\
\forall\, C;\ C';\ i? : I;\ o! : O \bullet (\theta C, i?) \mapsto (\theta C', o!) \in co \Rightarrow (\theta A, i?) \mapsto (\theta A', o!) \in ao \\
\equiv [\text{introducing pre and op schemas}] \\
\forall\, C;\ C';\ i? : I;\ o! : O \bullet CO \Rightarrow AO
\end{array}
$$

■

G.1.3 Relaxation for extra subclass operations

Proof G.17 (Lemma I, relaxation, no communication)

$\forall\, c : C \bullet f(c) \in \mathrm{dom}\, ao$
$\equiv$ [def of ao]
$\forall\, c : C \bullet f(c) \in \mathrm{dom}(f^{\sim} \fatsemi co \fatsemi f)$
$\equiv$ [def of dom]
$\forall\, c : C \bullet \exists\, a' : A' \bullet f(c) \mapsto a' \in f^{\sim}$
$\equiv$ [def of $\fatsemi$ twice]
$\forall\, c : C \bullet \exists\, a' : A' \bullet \exists\, c_2 : C;\ c' : C \bullet f(c) \mapsto c_2 \in f^{\sim} \wedge c_2 \mapsto c' \in co \wedge c' \mapsto a' \in f$
$\equiv$ [def of $^{\sim}$; f is function]
$\forall\, c : C \bullet \exists\, a' : A' \bullet \exists\, c_2 : C;\ c' : C \bullet f(c_2) = f(c) \wedge c_2 \mapsto c' \in co \wedge f(c') = a'$
$\equiv$ [witness: $c2 = c$; one-point-rule on a']
$\forall\, c : C \bullet \exists\, c' : C \bullet f(c) = f(c) \wedge c \mapsto c' \in co \wedge f(c') \in A'$
$\equiv$ [f is total; def of dom]
$\forall\, c : C \bullet c \in \mathrm{dom}\, co$

■

Proof G.18 (Lemma II, relaxation, no communication)

$\forall\, c' : C';\ c : C \bullet f(c) \mapsto f(c') \in ao$
$\equiv$ [def of ao]
$\forall\, c' : C';\ c : C \bullet f(c) \mapsto f(c') \in (f^{\sim} \fatsemi co \fatsemi f)$
$\equiv$ [def of $\fatsemi$ twice]
$\forall\, c' : C';\ c : C \bullet$
$\quad \exists\, c_2 : C;\ c_2' : C' \bullet f(c) \mapsto c_2 \in f^{\sim} \wedge c_2 \mapsto c_2' \in co \wedge c_2' \mapsto f(c') \in f$
$\equiv$ [prop of $^{\sim}$; f is total]
$\forall\, c' : C';\ c : C \bullet$
$\quad \exists\, c_2 : C;\ c_2' : C' \bullet f(c_2) = f(c) \wedge c_2 \mapsto c_2' \in co \wedge f(c_2') = f(c')$
$\equiv$ [witness: $c_2 = c$ and $c_2' = c'$]
$\forall\, c' : C';\ c : C \bullet c \mapsto c' \in co$

■

Proof G.19 (Forwards applicability, relaxation, no communication)

$ran((\mathrm{dom}\, ao) \lhd f^{\sim}) \subseteq \mathrm{dom}\, co$
$\equiv$ [see proof G.3 (page 302)]
$\forall\, c : C \bullet f(c) \in \mathrm{dom}\, ao \Rightarrow c \in \mathrm{dom}\, co$
$\equiv$ [lemma I, above (proof G.17)]
$\forall\, c : C \bullet c \in \mathrm{dom}\, co \Rightarrow c \in \mathrm{dom}\, co$
$\equiv$ [propositional calculus]
true

■

Proof G.20 (Backwards applicability, relaxation, no communication)

$\overline{\text{dom } co} \subseteq \text{dom}(f \rhd (\text{dom } ao))$

$\equiv$ [see proof G.4 (page 303)]
$\forall\, c : C \bullet c \notin \text{dom } co \Rightarrow f(c) \in A \wedge f(c) \notin \text{dom } ao$

$\equiv$ [f is total]
$\forall\, c : C \bullet c \notin \text{dom } co \Rightarrow f(c) \notin \text{dom } ao$

$\equiv$ [def of $\notin$ and propositional calculus]
$\forall\, c : C \bullet f(c) \in \text{dom } ao \Rightarrow c \in \text{dom } co$

$\equiv$ [lemma I, above (proof G.17)]
$\forall\, c : C \bullet c \in \text{dom } co \Rightarrow c \in \text{dom } co$

$\equiv$ [propositional calculus]
true

■

Proof G.21 (Non-blocking forwards correctness, relaxation, no communication)

$(\text{dom } ao \lhd r) \fatsemi co \subseteq ao \fatsemi r$

$\equiv$ [see proof G.5 (page 304)]
$\forall\, c' : C';\ c : C \bullet f(c) \in \text{dom } ao \wedge c \mapsto c' \in co \Rightarrow f(c) \mapsto f(c') \in ao$

$\equiv$ [using lemmas I (proof G.17) and II (proof G.18) above]
$\forall\, c' : C';\ c : C \bullet c \mapsto c' \in co \Rightarrow c \mapsto c' \in co$

$\equiv$ [by propositional calculus]
true

■

Proof G.22 (Non-blocking backwards correctness, relaxation, no communication)

$\text{dom}(f \rhd (\text{dom } ao)) \rhd\ co \fatsemi f \subseteq f \fatsemi ao$

$\equiv$ [see proof G.6 (page 305)]
$\forall\, c : C;\ c' : C' \bullet c \mapsto c' \in co \wedge f(c) \in \text{dom } ao \Rightarrow f(c) \mapsto f(c') \in ao$

$\equiv$ [using lemmas I (proof G.17) and II (proof G.18) above]
$\forall\, c : C;\ c' : C' \bullet c \mapsto c' \in co \Rightarrow c \mapsto c' \in co$

$\equiv$ [propositional calculus]
true

■

Proof G.23 (Blocking forwards correctness, relaxation, no communication)

$f^{\sim} \fatsemi co \subseteq ao \fatsemi f^{\sim}$

$\equiv$ [see G.7 (page 306)]
$\forall\, c, c' : C \bullet c \mapsto c' \in co \Rightarrow f(c) \mapsto f(c') \in ao$

$\equiv$ [lemma II (proof G.18) above]
$\forall\, c, c' : C \bullet c \mapsto c' \in co \Rightarrow c \mapsto c' \in ao$

$\equiv$ [propositional calculus]
true

■

Proof G.24 (Blocking backwards correctness, relaxation, no communication)

$co \fatsemi f \subseteq f \fatsemi ao$
$\equiv$ [see G.8 (page 307)]
$\forall\, c, c' : C \bullet c \mapsto c' \in co \Rightarrow f(c) \mapsto f(c') \in ao$
$\equiv$ [lemma II (proof G.18) above]
$\forall\, c, c' : C \bullet c \mapsto c' \in co \Rightarrow c \mapsto c' \in co$
$\equiv$ [propositional calculus]
true

■

Proof G.25 (Lemma I, relaxation, with communication)

$\forall\, c : C;\ i? : I \bullet (f(c), i?) \in \operatorname{dom} ao$
$\equiv$ [def of ao]
$\forall\, c : C;\ i? : I \bullet (f(c), i?) \in \operatorname{dom}(f^\sim \parallel id[I]) \fatsemi co \fatsemi (f \parallel id[O])$
$\equiv$ [def of dom]
$\forall\, c : C;\ i? : I \bullet \exists\, a' : A';\ o : O \bullet$
$\quad (f(c), i?) \mapsto (a', o) \in (f^\sim \parallel id[I]) \fatsemi co \fatsemi (f \parallel id[O])$
$\equiv$ [def of $\fatsemi$ twice]
$\forall\, c : C;\ i? : I \bullet \exists\, a' : A';\ o : O \bullet \exists\, c_2 : C;\ c' : C';\ i : I;\ o_2 : O \bullet$
$\quad (f(c), i?) \mapsto (c_2, i) \in f^\sim \parallel id[I] \wedge (c_2, i) \mapsto (c', o_2) \in co$
$\quad \wedge\, (c', o_2) \mapsto (a', o) \in f \parallel id[O]$
$\equiv$ [def of $\|$]
$\forall\, c : C;\ i? : I \bullet \exists\, a' : A';\ o : O \bullet \exists\, c_2 : C;\ c' : C';\ i : I;\ o_2 : O \bullet$
$\quad f(c) \mapsto c_2 \in f^\sim \wedge\, i? \mapsto i \in id[I] \wedge (c_2, i) \mapsto (c', o_2) \in co$
$\quad \wedge\, c' \mapsto a' \in f \wedge o_2 \mapsto o \in id[O]$
$\equiv$ [def of $^\sim$; prop of id; f is functional]
$\forall\, c : C;\ i? : I \bullet \exists\, a' : A';\ o : O \bullet \exists\, c_2 : C;\ c' : C';\ i : I;\ o_2 : O \bullet$
$\quad f(c_2) = f(c) \wedge i? = i \wedge (c_2, i) \mapsto (c', o_2) \in co \wedge f(c') = a' \wedge o_2 = o$
$\equiv$ [witness: $c_2 = c$; applications of 1-point-rule]
$\forall\, c : C;\ i? : I \bullet \exists\, c' : C';\ o : O \bullet f(c) = f(c) \wedge (c, i?) \mapsto (c', o) \in co$
$\equiv$ [def of dom; removing true preds]
$\forall\, c : C;\ i? : I \bullet (c, i?) \in \operatorname{dom} co$

■

Proof G.26 (Lemma II, relaxation, with communication)

$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (f(c), i?) \mapsto (f(c'), o!) \in ao$

$\equiv$ [def of ao]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (f(c), i?) \mapsto (f(c'), o!) \in (f^\sim \parallel id[I]) \fatsemi co \fatsemi (f \parallel id[O])$

$\equiv$ [def of $\fatsemi$ twice]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \exists\, c_2 : C;\ c_2' : C';\ i : I;\ o : O \bullet$
$\quad (f(c), i?) \mapsto (c2, i) \in f^\sim \parallel id[I]$
$\quad \wedge\ (c2, i) \mapsto (c2', o) \in co \wedge (c2', o) \mapsto (f(c'), o!) f \parallel id[O]$

$\equiv$ [def of $\|$]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \exists\, c_2 : C;\ c_2' : C';\ i : I;\ o : O \bullet$
$\quad (f(c), c2) \in f^\sim \wedge (i?, i) \in id[I] \wedge (c2, i) \mapsto (c2', o) \in co$
$\quad \wedge\ (c2', f(c')) \in f \wedge (o, o!) \in id[O]$

$\equiv$ [prop id and $^\sim$; f is function]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \exists\, c_2 : C;\ c_2' : C';\ i : I;\ o : O \bullet$
$\quad (c_2, f(c)) \in f \wedge i? = i \wedge (c_2, i) \mapsto (c_2', o) \in co$
$\quad \wedge\ f(c_2') = f(c') \wedge o = o!$

$\equiv$ [1-point-rule on i and o; f is function]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet \exists\, c_2 : C;\ c_2' : C' \bullet$
$\quad f(c_2) = f(c)) \wedge (c_2, i?) \mapsto (c_2', o!) \in co \wedge f(c_2') = f(c')$

$\equiv$ [witness: $c_2 = c$ and $c_2' = c'$; removing true preds]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co$

■

Proof G.27 (Forwards applicability, relaxation, with communication)

$ran((\mathrm{dom}\ ao) \lhd (f^\sim \parallel id[I])) \subseteq \mathrm{dom}\ co$

$\equiv$ [see proof G.11 (page 310)]
$\forall\, c : C;\ i? : I \bullet (f(c), i?) \in \mathrm{dom}\ ao \Rightarrow (c, i?) \in \mathrm{dom}\ co$

$\equiv$ [Lemma I (proof G.25), above]
$\forall\, c : C;\ i? : I \bullet (c, i?) \in \mathrm{dom}\ co \Rightarrow (c, i?) \in \mathrm{dom}\ co$

$\equiv$ [propositional calculus]
$true$

■

Proof G.28 (Backwards applicability, relaxation, with communication)

$\overline{\mathrm{dom}\ co} \subseteq \mathrm{dom}(s \parallel id[I] \rhd (\mathrm{dom}\ ao))$

$\equiv$ [see proof G.12 (page 310)]
$\forall\, c : C;\ i? : I \bullet (c, i?) \notin \mathrm{dom}\ co \Rightarrow (f(c), i?) \notin \mathrm{dom}\ ao$

$\equiv$ [def of $\notin$; propositional calculus]
$\forall\, c : C;\ i? : I \bullet (f(c), i?) \in \mathrm{dom}\ ao \Rightarrow (c, i?) \in \mathrm{dom}\ co$

$\equiv$ [Lemma I (proof G.25), above]
$\forall\, c : C;\ i? : I \bullet (c, i?) \in \mathrm{dom}\ co \Rightarrow (c, i?) \in \mathrm{dom}\ co$

$\equiv$ [propositional calculus]
$true$

■

Proof G.29 (Non-blocking forwards correctness, relaxation, with communication)

$(\operatorname{dom} ao) \lhd (r \parallel id[I]);\ co \subseteq ao \fatsemi (r \parallel id[O])$
$\equiv$ [see proof G.13 (page 311)]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet$
$\quad (f(c), i?) \in \operatorname{dom} ao \wedge (c, i?) \mapsto (c', o!) \in co \Rightarrow (f(c), i?) \mapsto (f(c'), o!) \in ao$
$\equiv$ [using lemmas I (proof G.25) and II (proof G.26)]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co \Rightarrow (c, i?) \mapsto (c', o!) \in co$
$\equiv$ [propositional calculus]
true

■

Proof G.30 (Non-blocking backwards correctness, relaxation, with communication)

$\operatorname{dom}((f \parallel id[I]) \rhd (\operatorname{dom} ao)) \rhd\ co \fatsemi (f \parallel id[O]) \subseteq (f \parallel id[I]) \fatsemi ao$
$\equiv$ [see proof G.14 (page 312)]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet$
$\quad (c, i?) \mapsto (c', o!) \in co \wedge (f(c), i?) \in \operatorname{dom} ao \Rightarrow (f(c), i?) \mapsto (f(c'), o!) \in ao$
$\equiv$ [using lemmas I (proof G.25) and II (proof G.26)]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co \Rightarrow (c, i?) \mapsto (c', o!) \in co$
$\equiv$ [propositional calculus]
true

■

Proof G.31 (Blocking forwards correctness, relaxation, with communication)

$(r \parallel id[I]);\ co \subseteq ao \fatsemi (r \parallel id[O])$
$\equiv$ [see proof G.15 (page 314)]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co \Rightarrow (f(c), i?) \mapsto (f(c'), o!) \in ao$
$\equiv$ [using lemmas I (proof G.25) and II (proof G.26)]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co \Rightarrow (c, i?) \mapsto (c', o!) \in co$
$\equiv$ [propositional calculus]
true

■

Proof G.32 (Blocking backwards correctness, relaxation, with communication)

$co \fatsemi (f \parallel id[O]) \subseteq (f \parallel id[I]) \fatsemi ao$
$\equiv$ [see proof G.16 (page G.16)]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet$
$\quad (c, i?) \mapsto (c', o!) \in co \Rightarrow (f(c), i?) \mapsto (f(c'), o!) \in ao$
$\equiv$ [using lemmas I (proof G.25) and II (proof G.26)]
$\forall\, c : C;\ c' : C';\ i? : I;\ o! : O \bullet (c, i?) \mapsto (c', o!) \in co \Rightarrow (c, i?) \mapsto (c', o!) \in co$
$\equiv$ [propositional calculus]
true

■

G.2 Proof of template inference rules

Proof G.33 (Rule T-one-point)

$\Gamma \vdash \exists\ \lessdot x \gtrdot : \lessdot t \gtrdot \bullet\ \lessdot P \gtrdot\ \wedge\ \lessdot x \gtrdot = \lessdot v \gtrdot$
$\quad \Rightarrow\ \lessdot P \gtrdot [\ \lessdot x \gtrdot := \lessdot v \gtrdot\ ]\ \wedge\ \lessdot v \gtrdot \in \lessdot t \gtrdot$
$\equiv$ [apply T-I-PN with $\{x \mapsto x_1\}$; apply T-I-PS with $\{t \mapsto t_1\}$ apply T-I-PE with $\{v \mapsto v_1\}$]
$\quad$ [t_1 and v_1 arbitrary type and expression such that $x_1 \notin FV(v_1)$]
$\vdash \exists\ x_1 : t_1 \bullet\ \lessdot P \gtrdot\ \wedge\ x_1 = v_1 \Rightarrow\ \lessdot P \gtrdot [\ x_1 := v_1\ ]\ \wedge\ v_1 \in t_1$
$\equiv$ [apply T-I-PP, P arbitrary predicate]
$\vdash \exists\ x_1 : t_1 \bullet\ P\ \wedge\ x_1 = v_1 \Rightarrow\ P [\ x_1 := v_1\ ]\ \wedge\ v_1 \in t_1$
$\equiv$ [by one-point-rule]
true

■

Proof G.34 (Rule T-∃-SC) The truth of the inference rule is demonstrated by proving the sequent:

$\vdash \exists\ [\lessdot ScD \gtrdot \mid \lessdot ScP \gtrdot] \bullet \lessdot P \gtrdot \Rightarrow \exists\ \lessdot ScD \gtrdot \bullet \lessdot P \gtrdot \wedge \lessdot ScP \gtrdot$

The proof is as follows:

$\equiv$ [apply $\mathcal{IP}_{\mathcal{E}}$ with $\{Sc \mapsto$ "Sc", $ScD \mapsto$ "ScD", $ScP \mapsto$ "ScP"; $P \mapsto$ "P"$\}$]
$\vdash \exists\ Sc \bullet P \Rightarrow \exists\, ScD \bullet P \wedge ScP$
$\equiv$ [by $\exists\ Sc$; hyp]
true
□

■

Proof G.35 (Rule T-∃-SC-2) The truth of the inference rule is demonstrated by proving the sequent:

$\vdash \exists\ ScA \bullet [\lessdot ScA \gtrdot;\ D \mid \lessdot ScP \gtrdot] \Rightarrow [D \mid \exists\ \lessdot ScA \gtrdot \bullet \lessdot ScP \gtrdot]$

The proof is as follows:

$\equiv$ [apply $\mathcal{IP}_{\mathcal{E}}$ with $\{ScA \mapsto \text{“}ScA\text{”}, ScP \mapsto \text{“}ScP\text{”}\}$]
$\vdash \exists\ ScA \bullet [ScA;\ D \mid ScP] \Rightarrow [D \mid \exists \llcorner ScA \lrcorner \bullet \llcorner ScP \lrcorner]$
$\equiv$ [by $\exists$ Sc-2; hyp]
true
□

■

Proof G.36 (Rule T-ls-one-point) Template conjecture:

$$\Gamma \vdash \exists [\![\llcorner x \lrcorner : \llcorner t \lrcorner]\!] \bullet \llcorner P \lrcorner [\![\wedge \llcorner x \lrcorner = \llcorner v \lrcorner]\!] \Rightarrow \llcorner P \lrcorner [[\![\llcorner x \lrcorner := \llcorner v \lrcorner]\!]] [\![\wedge \llcorner v \lrcorner \in \llcorner t \lrcorner]\!]$$

It can easily be shown that the conjecture holds for one particular instantiation of the lists. For example, when there is just one variable in the existential quantifier:

$\equiv$ [apply $\mathcal{IL}_{\mathcal{E}}$ with $\langle\{x \mapsto x_1, t \mapsto t_1, v \mapsto v_1\}\rangle$; t_1 and v_1 arbitrary type and expression]
[and $x_1 \notin FV(v_1)$]
$\vdash \exists\ x_1 : t_1 \bullet \llcorner P \lrcorner \wedge x_1 = v_1 \Rightarrow \llcorner P \lrcorner [\ x_1 := v_1\ ] \wedge v_1 \in t_1$
$\equiv$ [apply $\mathcal{IP}_{\mathcal{E}}$ with $\{P \mapsto P\}$, P arbitrary predicate; apply $\rightsquigarrow_Z$]
$\vdash \exists\ x_1 : t_1 \bullet P \wedge x_1 = v_1 \Rightarrow P [\ x_1 := v_1\ ] \wedge v_1 \in t_1$
$\equiv$ [by one-point-rule]
true
□

The proof above just covers one case. To establish the truth of the inference rule it needs to be proved for any valid instantiation of the lists. This requires a proof by induction on the instantiation sequence.

The proof pattern outlined above is the same in the induction proof: (a) the instantiation of the list, (b) the instantiation of the parameter and (c) the casting into Z. The variation lies in the instantiation of the lists.

So, given the template abbreviations, involved:

$$T_1 == \exists [\![\llcorner x \lrcorner : \llcorner t \lrcorner]\!] \bullet \llcorner P \lrcorner [\![\wedge \llcorner x \lrcorner = \llcorner v \lrcorner]\!]$$
$$T_2 == \llcorner P \lrcorner [[\![\llcorner x \lrcorner := \llcorner v \lrcorner]\!]] [\![\wedge \llcorner v \lrcorner \in \llcorner t \lrcorner]\!]$$

The property to prove can be defined as:

$$P(is) == \Gamma \vdash \mathcal{IL}_{\mathcal{E}}\ T_1\ is \Rightarrow \mathcal{IL}_{\mathcal{E}}\ T_2\ is$$

This needs to be proved for all valid instantiations:

$$\forall\, is : \operatorname{seq} Env \mid valid(is) \bullet P(is)$$

Base Case. The instantiation sequence is empty: $P(\langle\rangle)$.

$\Gamma \vdash \mathcal{IL}_{\mathcal{E}}\ T_1\ \langle\ \rangle \Rightarrow \mathcal{IL}_{\mathcal{E}}\ T_2\ \langle\rangle$

$\equiv$ [Apply $\mathcal{IL}_{\mathcal{E}}$]

$\Gamma \vdash \exists\ \{\} \bullet\ \ll P \gg\ \Rightarrow\ \ll P \gg [\ \{\}\ ]$

$\equiv$ [apply $\mathcal{IP}_{\mathcal{E}}$; P is arbitrary predicate; apply $\rightsquigarrow_Z$]

$\Gamma \vdash \exists\ \{\} \bullet\ P\ \Rightarrow\ P [\ \{\}\ ]$

$\equiv$ [Predicate calculus]

$\Gamma \vdash P\ \Rightarrow\ P$

$\equiv$ [by $\Rightarrow\ I$ and hyp]

true

□

Induction Step. The instantiation sequence has at least one environment. Assuming a valid instantiation sequence se, we need to prove:

$P(se) \Rightarrow P(se \frown \langle e \rangle)$

The following lemmas (proved below) are introduced:

$\vdash?\ \ \forall\, se : seqEnv;\ e : Env \bullet$
$\mathcal{IL}_{\mathcal{E}}\ (T_1)(se\ \frown\ \langle\ e\ \rangle)$
$= \mathcal{IP}_{\mathcal{E}}(\exists\ \ll x \gg : \ll t \gg \bullet (\mathcal{IL}_{\mathcal{E}}\ (T_1)\ se)\ \wedge\ \ll x \gg = \ll v \gg)\ e$

$\vdash?\ \ \forall\, se : seqEnv;\ e : Env \bullet$
$\mathcal{IL}_{\mathcal{E}}(T_2)(se \frown \langle e \rangle)$
$= \mathcal{IP}_{\mathcal{E}}((\mathcal{IL}_{\mathcal{E}}\ T_2\ se) [\ \ll x \gg := \ll v \gg\ ]\ \wedge\ \ll v \gg \in \ll t \gg)\ e$

The proof is as follows

$\Gamma \vdash \mathcal{IL}_{\mathcal{E}}\ T_1\ (se \frown \langle e \rangle) \Rightarrow \mathcal{IL}_{\mathcal{E}}\ T_2\ (se \frown \langle e \rangle)$

$\equiv$ [by lemmas; thin]

$\vdash\ \mathcal{IP}_{\mathcal{E}}(\exists\ \ll x \gg : \ll t \gg \bullet (\mathcal{IL}_{\mathcal{E}}\ (T_1)\ se)\ \wedge\ \ll x \gg = \ll v \gg)\ e$
$\Rightarrow \mathcal{IP}_{\mathcal{E}}((\mathcal{IL}_{\mathcal{E}}\ T_2\ se) [\ \ll x \gg := \ll v \gg\ ]\ \wedge\ \ll v \gg \in \ll t \gg)\ e$

$\equiv$ [by induction hypothesis]

$\vdash\ \mathcal{IP}_{\mathcal{E}}(\exists\ \ll x \gg : \ll t \gg \bullet (\mathcal{IL}_{\mathcal{E}}\ (T_2)\ se)\ \wedge\ \ll x \gg = \ll v \gg)\ e$
$\Rightarrow \mathcal{IP}_{\mathcal{E}}((\mathcal{IL}_{\mathcal{E}}\ T_2\ se) [\ \ll x \gg := \ll v \gg\ ]\ \wedge\ \ll v \gg \in \ll t \gg)\ e$

$\equiv$ [apply $\mathcal{IP}_{\mathcal{E}}$ with $e = \{x \mapsto x_{n+1}, t \mapsto t_{n+1}, v \mapsto v_{n+1}\}$]
[t_{n+1} and v_{n+1} are arbitrary types and expression; $x_{n+1} \notin FV(v_{n+1})$]

$\vdash\ \exists\ x_{n+1} : t_{n+1} \bullet (\mathcal{IL}_{\mathcal{E}}\ (T_2)\ se)\ \wedge\ x_{n+1} = v_{n+1}$
$\Rightarrow (\mathcal{IL}_{\mathcal{E}}\ T_2\ se) [\ x_{n+1} := v_{n+1}\ ]\ \wedge\ v_{n+1} \in t_{n+1}$

$\equiv$ [one-point rule]

$\vdash\ (\mathcal{IL}_{\mathcal{E}}\ (T_2)\ se) [\ x_{n+1} := v_{n+1}\ ]\ \wedge\ v_{n+1} \in t_{n+1}$
$\Rightarrow (\mathcal{IL}_{\mathcal{E}}\ T_2\ se) [\ x_{n+1} := v_{n+1}\ ]\ \wedge\ v_{n+1} \in t_{n+1}$

$\equiv$ [$\Rightarrow\ I$ and hyp]

true

□

■

G.3 Meta-proofs

G.3.1 Intensional View

Proof G.37 (Template T5, cl-init) The sequent to prove is:

$$\vdash \exists \ll Cl\gg Init \bullet \text{true} \Rightarrow \exists \ll Cl\gg Init_I \bullet \ll CLI\gg' [\![\ll at\gg' := \ll iv\gg]\!] \wedge \ll pI\gg [\![\ll at\gg' := \ll iv\gg]\!]$$

The proof derives the consequent of the implication from the antecedent.

$\vdash \exists \ll Cl\gg Init \bullet \text{true}$

$\equiv$ [by T $\exists$ *Sc*]

$\vdash \exists \ll Cl\gg '; \ll Cl\gg Init_I \bullet [\![\ll at\gg' = \ll iv\gg]\!] \wedge \ll pI\gg$

$\equiv$ [by T $\exists$ *Sc*]

$\vdash \exists [\![\ll at\gg' : \ll atT\gg]\!]; \ll Cl\gg Init_I \bullet \ll CLI\gg' \wedge \ll pI\gg \wedge [\![\ll at\gg' = \ll iv\gg]\!]$

$\equiv$ [by T-ls-one-point; well-formedness proof]

$\vdash \exists \ll Cl\gg Init_I \bullet \ll CLI\gg' [\![\ll at\gg' := \ll iv\gg]\!] \wedge \ll pI\gg [\![\ll at\gg' := \ll iv\gg]\!]$

■

Proof G.38 (Template T5, cl-init-ni)

$\ll CLI\gg \wedge \ll pI\gg \vdash \exists \ll Cl\gg Init \bullet \text{true}$

$\equiv$ [using theorem above]

$\ll CLI\gg \wedge \ll pI\gg$

$\vdash \exists \ll Cl\gg Init_I \bullet \ll CLI\gg' [\![\ll at\gg' := \ll iv\gg]\!] \wedge \ll pI\gg [\![\ll at\gg' := \ll iv\gg]\!]$

$\equiv$ [by hyp; thin]

$\vdash \exists \ll Cl\gg Init_I \bullet \text{true}$

$\equiv$ [by proviso]

true

■

Proof G.39 (Template T6, cl-uop-pre)

pre $\ll Cl\gg \Delta \ll Op\gg$

$\equiv$ [def of pre]

$\exists \ll Cl\gg ' \bullet \ll Cl\gg \Delta \ll Op\gg$

$\equiv$ [def of $\ll Cl\gg \Delta \ll Op\gg$, T-$\exists$ *Sc*-2]

$[\ll Cl\gg; [\![\ll io\gg? : \ll ioT\gg]\!] |$

$\quad \exists \ll Cl\gg ' \bullet$

$\quad\quad \ll pOp\gg' \wedge [\![\ll at\gg' = \ll nv\gg]\!]$

$]$

$\equiv$ [def of $\ll Cl\gg'$; T-one-point-rule; well-formedness proof]

$[\ll Cl\gg; [\![\ll io\gg? : \ll ioT\gg]\!] |$

$\quad\quad \ll CLI\gg' [\![\ll at\gg' := \ll nv\gg]\!] \wedge \ll pOp\gg' [\![\ll at\gg' := \ll nv\gg]\!]$

$]$

■

Proof G.40 (Template T6, cl-uop-epre-np)

$\ll pOp\gg;\ \ll CLI\gg \vdash \exists\ \text{pre}\ \ll Cl\gg\Delta\ll Op\gg \bullet \text{true}$
$\equiv$ [by theorem above]
$\ll pOp\gg;\ \ll CLI\gg$
$\vdash \exists\ [\ll Cl\gg;\ [\![\ \ll io\gg? : \ll ioT\gg\ ]\!]\ |$
$\quad \ll CLI\gg'[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ] \wedge \ll pOp\gg'[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ]]$
$\bullet$ true
$\equiv$ [by hyp; by T-$\exists$ Sc]
$\ll pOp\gg;\ \ll CLI\gg \vdash \exists\ \ll Cl\gg[\![\ ;\ \ll io\gg? : \ll ioT\gg\ ]\!] \bullet \text{true}$
$\equiv$ [by def of $\ll Cl\gg$; by T-$\exists$ Sc]
$\ll pOp\gg;\ \ll CLI\gg \vdash \exists\ [\![\ \ll at\gg : \ll atT\gg\ ]\!][\![\ ;\ \ll io\gg? : \ll ioT\gg\ ]\!] \bullet \ll CLI\gg$
$\equiv$ [by hyp; by proviso]
true

■

Proof G.41 (Template T6, meta-lemma cl-uop-d-post)

$\vdash \exists \ll Cl\gg' \bullet \ll P\gg \wedge \ll Cl\gg_{\Delta}\ll Op\gg$
$\equiv$ [by def of $\ll Cl\gg_{\Delta}\ll Op\gg$; by t-one-point]
$\ll P\gg[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ]$
$\quad \wedge\ [\ \ll Cl\gg;\ [\![\ \ll io\gg? : \ll ioT\gg\ ]\!]\ |$
$\quad\quad \ll CLI\gg'[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ] \wedge \ll pOp\gg'[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ]\ ]$
$\equiv$ [by def of pre$\ll Cl\gg_{\Delta}\ll Op\gg$]
$\ll P\gg[\ [\![\ \ll at\gg' := \ll nv\gg\ ]\!]\ ] \wedge \text{pre}\ll Cl\gg_{\Delta}\ll Op\gg$

■

Proof G.42 (Template T7, cl-oop-pre)

$\text{pre}\ \ll Cl\gg_{\Xi}\ll Op\gg$
$\equiv$ [def of pre]
$\exists\ \ll Cl\gg\ ';\ \ll out\gg! : \ll oT\gg \bullet \ll Cl\gg_{\Xi}\ll Op\gg$
$\equiv$ [def of $\ll Cl\gg_{\Xi}\ll Op\gg$, by T-$\exists$ Sc-2]
$[\ll Cl\gg\ |\ \exists \ll Cl\gg\ ';\ \ll out\gg! : \ll oT\gg \bullet$
$\quad \ll CLI\gg' \wedge \ll pOp\gg \wedge \ll out\gg! = \ll ov\gg \wedge \theta\ll Cl\gg = \theta\ll Cl\gg'\ ]$
$\equiv$ [def of $\ll Cl\gg$, T-θ-point; by T-one-point;]
$[\ll Cl\gg\ |\ \ll CLI\gg'[\theta\ll Cl\gg' := \theta\ll Cl\gg] \wedge \ll pOp\gg[\theta\ll Cl\gg' := \theta\ll Cl\gg, \ll out\gg! := \ll ov\gg]$
$\quad \wedge \ll ov\gg \in \ll oT\gg\ ]$
$\equiv$ [T-schema-calculus; well-formedness conjecture]
$[\ll Cl\gg \mid \ll pOp\gg[\ \theta\ll Cl\gg\ ' := \theta\ll Cl\gg, \ll out\gg! := \ll ov\gg\ ]$

■

Proof G.43 (Template T7, cl-oop-pre) As proof G.40.
■

Proof G.44 (Template T8, cl-fin)

$\vdash \exists\ \lessdot Cl \gtrdot Fin \bullet \text{true}$
$\equiv$ [by T-$\exists$ Sc]
$\vdash \exists\ \lessdot Cl \gtrdot \bullet \lessdot fc \gtrdot$
$\equiv$ [by T-$\exists$ Sc]
$\vdash \exists\ [\![\ \lessdot at \gtrdot : \lessdot atT \gtrdot\ ]\!] \bullet \lessdot CLI \gtrdot \wedge \lessdot fc \gtrdot$

■

Proof G.45 (Template T8, cl-fin-ni)

$\lessdot CLI \gtrdot \wedge \lessdot fc \gtrdot$
$\vdash \exists\ \lessdot Cl \gtrdot Fin \bullet \text{true}$
$\equiv$ [by theorem above]
$\lessdot CLI \gtrdot \wedge \lessdot fc \gtrdot$
$\vdash \exists\ [\![\ \lessdot at \gtrdot : \lessdot atT \gtrdot\ ]\!] \bullet \lessdot CLI \gtrdot \wedge \lessdot fc \gtrdot$
$\equiv$ [by hyp]
$\vdash \exists\ [\![\ \lessdot at \gtrdot : \lessdot atT \gtrdot\ ]\!] \bullet \text{true}$
$\equiv$ [by proviso]
true

■

Proof G.46 (Template T9, cl-stc-init)

$\vdash \exists\ \lessdot Cl \gtrdot Init \bullet \text{true}$
$\equiv$ [by def of $\lessdot Cl \gtrdot Init$]
$\vdash \exists\ \lessdot Cl \gtrdot Init_0;\ \lessdot Cl \gtrdot StInit \bullet \text{true}$
$\equiv$ [by $T\ \exists\ Sc-\wedge$]
$\vdash \exists\ \lessdot Cl \gtrdot Init_0 \bullet \text{true} \wedge \exists \lessdot Cl \gtrdot StInit \bullet \text{true}$
$\equiv$ [by T $\exists\ Sc$]
$\vdash \exists\ \lessdot Cl \gtrdot '; \lessdot Cl \gtrdot Init_I \bullet [\![\ \lessdot at \gtrdot' = \lessdot iv \gtrdot\ ]\!] \wedge \lessdot pI \gtrdot$
$\wedge \exists \lessdot Cl \gtrdot St\ ' \bullet st' = \lessdot iSt \gtrdot$
$\equiv$ [by T $\exists$ Sc twice]
$\vdash \exists\ [\![\ \lessdot at \gtrdot' : \lessdot atT \gtrdot\ ]\!];\ \lessdot Cl \gtrdot Init_I \bullet$
$\quad \lessdot CLI \gtrdot' \wedge \lessdot pI \gtrdot \wedge [\![\ \lessdot at \gtrdot' = \lessdot iv \gtrdot\ ]\!]$
$\wedge \exists st' : \lessdot Cl \gtrdot ST \bullet st' = \lessdot iSt \gtrdot$
$\equiv$ [by T-one-point; T-ls-one-point; well-formedness proof]
$\vdash \exists\ \lessdot Cl \gtrdot Init_I \bullet$
$\quad \lessdot CLI \gtrdot' [\ [\![\ \lessdot at \gtrdot' := \lessdot iv \gtrdot\ ]\!]\ ] \wedge \lessdot pI \gtrdot [\![\ \lessdot at \gtrdot' := \lessdot iv \gtrdot\ ]\!]$

■

Proof G.47 (Template T9, cl-stc-init-ni) As proof G.38.
■

Proof G.48 (Template T10, cl-stc-uop-pre) First, the precondition of $\lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot St$ is calculated:

$\mathrm{pre} \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot St$
$\equiv$ [def of pre]
$\exists \lessdot Cl \gtrdot St\ ' \bullet \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot St$
$\equiv$ [def of $\lessdot Cl \gtrdot \Delta \lessdot Op \gtrdot St$ by T-$\exists$ Sc-2; def of $\lessdot Cl \gtrdot St\ '$]
$[\ \lessdot Cl \gtrdot St \mid \exists st' : \lessdot Cl \gtrdot ST \bullet$
$\quad st = \lessdot bSt \gtrdot \wedge st' = \lessdot aSt \gtrdot \vee [\![\ st = \lessdot bSt \gtrdot \wedge st' = \lessdot aSt \gtrdot\ ]\!]_{(\vee,\mathrm{false})}\ ]$
$\equiv$ [T-one-point-rule; predicate calculus]
$[\ \lessdot Cl \gtrdot St \mid st = \lessdot bSt \gtrdot \vee [\![\ st = \lessdot bSt \gtrdot\ ]\!]_{(\vee,\mathrm{false})}\ ]$

Then, $\mathrm{pre} \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot_0$ is calculated. The calculation is as proof G.39 to give:

$[\lessdot Cl \gtrdot; [\![\ \lessdot io \gtrdot ? : \lessdot io T \gtrdot\ ]\!] \mid$
$\quad \lessdot CLI \gtrdot'[\ [\![\ \lessdot at \gtrdot' := \lessdot nv \gtrdot\ ]\!]\ ] \wedge \lessdot pOp \gtrdot'[\ [\![\ \lessdot at \gtrdot' := \lessdot nv \gtrdot\ ]\!]\ ]$
$]$

This used to derive the pre-condition of the composed operation:

$\mathrm{pre}\ \lessdot Cl \gtrdot \Delta \lessdot Op \gtrdot$
$\equiv$ [def of $\lessdot Cl \gtrdot \Delta \lessdot Op \gtrdot$]
$\mathrm{pre}(\lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot_0 \wedge \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot St)$
$\equiv$ [property of pre]
$\mathrm{pre} \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot_0 \wedge \mathrm{pre} \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot St$
$\equiv$ [$pre \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot St$ and $\mathrm{pre} \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot_0$ above]
$[\ \lessdot Cl \gtrdot St \mid st = \lessdot bSt \gtrdot \vee [\![\ st = \lessdot bSt \gtrdot\ ]\!]_{(\vee,\mathrm{false})}\ ]$
$\wedge [\lessdot Cl \gtrdot; [\![\ \lessdot io \gtrdot ? : \lessdot io T \gtrdot\ ]\!] \mid$
$\quad \wedge \lessdot CLI \gtrdot'[\ [\![\ \lessdot at \gtrdot' := \lessdot nv \gtrdot\ ]\!]\ ] \wedge \lessdot pOp \gtrdot'[\ [\![\ \lessdot at \gtrdot' := \lessdot nv \gtrdot\ ]\!]\ ]$
$]$

■

Proof G.49 (Template T10, cl-stc-uop-epre-np) First, we prove:

$\vdash \exists\, \mathrm{pre} \lessdot Cl \gtrdot_\Delta \lessdot Op \gtrdot St \bullet \mathrm{true}$
$\equiv$ [theorem above; by T-$\exists$ Sc]
$\vdash \exists \lessdot Cl \gtrdot St \bullet st = \lessdot bSt \gtrdot \vee [\![\ st = \lessdot bSt \gtrdot\ ]\!]_{(\vee,\mathrm{false})}$
$\equiv$ [def of $\lessdot Cl \gtrdot St$]
$\vdash \exists\, st : \lessdot Cl \gtrdot St \bullet st = \lessdot bSt \gtrdot \vee [\![\ st = \lessdot bSt \gtrdot\ ]\!]_{(\vee,\mathrm{false})}$
$\equiv$ [T-one-point rule]
true

The same is performed for the other component operation:

$\lessdot CLI \gtrdot \wedge \lessdot pOp \gtrdot \vdash \exists\, \mathrm{pre}\ \lessdot Cl \gtrdot \Delta \lessdot Op \gtrdot_0 \bullet \mathrm{true}$

This is true the proof is as proof G.38.
The proof of the sequent that is required to prove becomes trivial:

$\ll CLI\gg \wedge \ll pOp\gg \vdash \exists\,\mathrm{pre}\ \ll Cl\gg\Delta\ll Op\gg \bullet \mathrm{true}$
$\equiv$ [def of $\ll Cl\gg\Delta\ll Op\gg$]
$\ll CLI\gg \wedge \ll pOp\gg \vdash \exists\,\mathrm{pre}\ (\ll Cl\gg\Delta\ll Op\gg_0 \wedge \ll Cl\gg_\Delta\ll Op\gg St) \bullet \mathrm{true}$
$\equiv$ [prop of pre]
$\ll CLI\gg \wedge \ll pOp\gg \vdash \exists\,\mathrm{pre}\ \ll Cl\gg\Delta\ll Op\gg_0 \wedge \mathrm{pre}\ll Cl\gg_\Delta\ll Op\gg St \bullet \mathrm{true}$
$\equiv$ [T-$\exists$ Sc-$\wedge$]
$\ll CLI\gg \wedge \ll pOp\gg \vdash \exists\,\mathrm{pre}\ \ll Cl\gg\Delta\ll Op\gg_0 \bullet \mathrm{true} \wedge \exists\,\mathrm{pre}\ll Cl\gg_\Delta\ll Op\gg St \bullet \mathrm{true}$
$\equiv$ [by the two lemmas proved above]
true

■

G.3.2 Extensional View

Proof G.50 (Template T19, cl-ext-init)

$\exists\ \mathbb{S}\ll Cl\gg Init \bullet \mathrm{true}$
$\equiv$ [by T-$\exists$ Sc]
$\exists\ \mathbb{S}\ll Cl\gg\ ' \bullet s\ll Cl\gg' = \varnothing \wedge st\ll Cl\gg' = \varnothing \wedge \mathrm{true}$
$\equiv$ [by T-$\exists$ Sc; $\wedge$-Id]
$\exists\ s\ll Cl\gg' : \mathbb{P}(\mathbb{O}\ll Cl\gg Cl);\ st\ll Cl\gg' : (\mathbb{O}\ll Cl\gg Cl) \nrightarrow \ll Cl\gg \bullet$
$\quad \mathrm{dom}\ st\ll Cl\gg' = s\ll Cl\gg' \wedge s\ll Cl\gg' = \varnothing \wedge st\ll Cl\gg' = \varnothing$
$\equiv$ [by T-one-point twice]
$\mathrm{dom}\ \varnothing = \varnothing \wedge \varnothing \in \mathbb{P}(\mathbb{O}\ \ll Cl\gg Cl) \wedge \varnothing \in (\mathbb{O}\ \ll Cl\gg Cl) \nrightarrow \ll Cl\gg$
$\equiv$ [by set theory]
true

■

Proof G.51 (Template T19, meta-lemma cl-ext-init-d)

$\exists\ \mathbb{S}\ll Cl\gg Init \bullet \ll P\gg$
$\equiv$ [by T-$\exists$ Sc twice; and by T-one-point]
$\mathrm{dom}\ st\ll Cl\gg' = s\ll Cl\gg'[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ]$
$\quad \wedge \ll P\gg[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ]$
$\quad \wedge \varnothing \in \mathbb{P}(\mathbb{O}\ \ll Cl\gg Cl) \wedge \varnothing \in (\mathbb{O}\ \ll Cl\gg Cl) \nrightarrow \ll Cl\gg$
$\equiv$ [by prop of substitution and hyp]
$\ll P\gg[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ]$

■

Proof G.52 (Template T19, cl-ext-init-lsd) The sequent to prove is:

$\exists\ [\![\ \mathbb{S}\ll Cl\gg Init\ ]\!] \bullet \mathrm{true}$
$\vdash (\exists\ [\![\ \mathbb{S}\ll Cl\gg Init\ ]\!] \bullet \ll P\gg) \Rightarrow \ll P\gg[\ [\![\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ]\!]\ ]$

By applying T-I-L and conj, the sequent is divided in two corresponding to the base case and the induction step of a proof by induction.

Base Case.

$$
\begin{array}{l}
\mathcal{IL}_{\mathcal{E}}(\exists\ [\![\ \mathbb{S}{\ll}Cl{\gg}Init\ ]\!] \bullet \text{true} \\
\vdash (\exists\ [\![\ \mathbb{S}{\ll}Cl{\gg}Init\ ]\!] \bullet {\ll}P{\gg}) \Rightarrow {\ll}P{\gg}[\ [\![\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ]\!]\ ])(\{Cl\}, \langle\rangle) \\
\equiv [\text{By } \mathcal{IL}_{\mathcal{E}}] \\
\vdash (\exists\ \{\} \bullet {\ll}P{\gg}) \Rightarrow {\ll}P{\gg}[\ \{\}\ ] \\
\equiv [\text{By predicate calculus and substitution}] \\
\vdash {\ll}P{\gg} \Rightarrow {\ll}P{\gg} \\
\equiv [\text{by } \Rightarrow\text{-hyp and hyp}] \\
\text{true}
\end{array}
$$

Induction Step.

The list ins instantiated at least once. So, it is assumed that the following sequent is true:

$$
\begin{array}{l}
\exists\ [\![\ \mathbb{S}{\ll}Cl{\gg}Init\ ]\!] \bullet \text{true} \\
\vdash (\exists\ [\![\ \mathbb{S}{\ll}Cl{\gg}Init\ ]\!] \bullet {\ll}P{\gg}) \Rightarrow {\ll}P{\gg}[\ [\![\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ]\!]\ ]
\end{array}
$$

And by induction, it has to be proved that:

$$
\begin{array}{l}
\exists\ [\![\ \mathbb{S}{\ll}Cl{\gg}Init\ ]\!];\ \mathbb{S}{\ll}Cl{\gg}Init \bullet \text{true} \\
\vdash (\exists\ [\![\ \mathbb{S}{\ll}Cl{\gg}Init\ ]\!];\ \mathbb{S}{\ll}Cl{\gg}Init \bullet {\ll}P{\gg}) \\
\qquad \Rightarrow {\ll}P{\gg}[\ [\![\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ]\!], s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing]
\end{array}
$$

The antecedent of the implication is simplified as follows:

$$
\begin{array}{l}
\vdash \exists\ [\![\ \mathbb{S}{\ll}Cl{\gg}Init\ ]\!];\ \mathbb{S}{\ll}Cl{\gg}Init \bullet {\ll}P{\gg} \\
\equiv\ [\text{By induction hypothesis}] \\
\vdash \exists\ \mathbb{S}{\ll}Cl{\gg}Init \bullet {\ll}P{\gg}[\ [\![\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ]\!]\ ] \\
\equiv\ [\text{By dist-ext-init}] \\
\vdash\ {\ll}P{\gg}[\ [\![\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ]\!], s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing]
\end{array}
$$

And the resulting sequent is trivially true:

$$
\begin{array}{l}
\vdash\ {\ll}P{\gg}[\ [\![\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ]\!], s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing] \\
\qquad \Rightarrow {\ll}P{\gg}[\ [\![\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ]\!], s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing] \\
\equiv [\text{by } \Rightarrow\text{-hyp and hyp}] \\
\text{true}
\end{array}
$$

■

G.3.3 Relational View

Proof G.53 (Template T47, assoc-init)

$\vdash \exists\ \mathbb{A}{\ll}As{\gg}Init \bullet \text{true}$
$\equiv$ [by T-$\exists$ Sc]
$\vdash \exists\ As{\ll}As{\gg}' \bullet r{\ll}As{\gg}' = \varnothing \wedge \text{true}$
$\equiv$ [by T-$\exists$ Sc; $\wedge$-Id]
$\vdash \exists\ r{\ll}As{\gg} : \mathbb{O}{\ll}Cl_A{\gg}Cl \leftrightarrow \mathbb{O}{\ll}Cl_B{\gg}Cl \bullet r{\ll}As{\gg}' = \varnothing$
$\equiv$ [by T-one-point]
$\vdash \varnothing \in \mathbb{O}{\ll}Cl_A{\gg}Cl \leftrightarrow \mathbb{O}{\ll}Cl_B{\gg}Cl$
$\equiv$ [by set theory; $\wedge$-Id]
$\vdash$ true

■

Proof G.54 (Template T47, assoc-init-d)

$\exists\ \mathbb{A}{\ll}As{\gg}Init \bullet {\ll}P{\gg}$
$\equiv$ [by T-$\exists$ Sc]
$\vdash \exists\ \mathbb{A}{\ll}As{\gg}' \bullet r{\ll}As{\gg}' = \varnothing \wedge {\ll}P{\gg}$
$\equiv$ [by T-$\exists$ Sc]
$\vdash \exists\ r{\ll}As{\gg} : \mathbb{O}{\ll}Cl_A{\gg}Cl \leftrightarrow \mathbb{O}{\ll}Cl_B{\gg}Cl \bullet r{\ll}As{\gg}' = \varnothing \wedge {\ll}P{\gg}$
$\equiv$ [by T-one-point]
$\vdash {\ll}P{\gg}'\ [r{\ll}As{\gg}' := \varnothing] \wedge \varnothing \in \mathbb{O}{\ll}Cl_A{\gg}Cl \leftrightarrow \mathbb{O}{\ll}Cl_B{\gg}Cl$
$\equiv$ [by set theory; $\wedge$-Id]
${\ll}P{\gg}'\ [r{\ll}As{\gg}' := \varnothing]$

■

Proof G.55 (Template T47, assoc-init-lsd) The sequent to prove is:

$\vdash \exists\ [\![\ \mathbb{A}{\ll}As{\gg}Init\ ]\!] \bullet {\ll}P{\gg} \Rightarrow {\ll}P{\gg}[\ [\![\ r{\ll}As{\gg}' := \varnothing\ ]\!]\ ]$

By applying T-I-L and conj, the sequent is divided in two corresponding to the base case and the induction step of a proof by induction.

Base Case.

$\mathcal{IL}_{\mathcal{E}}(\vdash (\exists\ [\![\ \mathbb{A}{\ll}As{\gg}Init\ ]\!] \bullet {\ll}P{\gg}) \Rightarrow {\ll}P{\gg}[\ [\![\ r{\ll}As{\gg}' := \varnothing\ ]\!]\ ])(\{As\}, \langle\rangle)$
$\equiv$ [By $\mathcal{IL}_{\mathcal{E}}$]
$\vdash (\exists\ \{\} \bullet {\ll}P{\gg}) \Rightarrow {\ll}P{\gg}[\ \{\}\ ]$
$\equiv$ [By predicate calculus and substitution]
$\vdash {\ll}P{\gg} \Rightarrow {\ll}P{\gg}$
$\equiv$ [by $\Rightarrow$-hyp and hyp]
true

Induction Step: The list is instantiated at least once. So, it is assumed that the following sequent is true:

$$\vdash (\exists \llbracket\ \mathbb{A}{\ll}As{\gg}Init\ \rrbracket \bullet {\ll}P{\gg}) \Rightarrow {\ll}P{\gg}[\llbracket\ r{\ll}As{\gg}' := \varnothing\ \rrbracket]$$

And by induction, it has to be proved that:

$$\vdash (\exists \llbracket\ \mathbb{A}{\ll}As{\gg}Init\ \rrbracket;\ \mathbb{S}{\ll}As{\gg}Init \bullet {\ll}P{\gg}) \Rightarrow {\ll}P{\gg}[\llbracket\ r{\ll}As{\gg}' := \varnothing\ \rrbracket, r{\ll}As{\gg}' := \varnothing]$$

The antecedent of the implication is simplified as follows:

$$\vdash \exists \llbracket\ \mathbb{A}{\ll}As{\gg}Init\ \rrbracket;\ \mathbb{A}{\ll}As{\gg}Init \bullet {\ll}P{\gg}$$
$$\equiv \text{ [By induction hypothesis]}$$
$$\vdash \exists\ \mathbb{A}{\ll}As{\gg}Init \bullet {\ll}P{\gg}[\llbracket\ r{\ll}As{\gg}' := \varnothing\ \rrbracket]$$
$$\equiv \text{ [By dist-assoc-init]}$$
$$\vdash\ {\ll}P{\gg}[\llbracket\ r{\ll}As{\gg}' := \varnothing\ \rrbracket, r{\ll}As{\gg}' := \varnothing]$$

And the resulting sequent is trivially true:

$$\vdash\ {\ll}P{\gg}[\llbracket\ r{\ll}As{\gg}' := \varnothing\ \rrbracket, r{\ll}As{\gg}' := \varnothing] \Rightarrow {\ll}P{\gg}[\llbracket\ r{\ll}As{\gg}' := \varnothing\ \rrbracket, r{\ll}As{\gg}' := \varnothing]$$
$$\equiv \text{[by } \Rightarrow\text{-hyp and hyp]}$$
$$\text{true}$$

■

G.3.4 Global View

Proof G.56 (Template T54, sys-init) First, we prove a lemma:

$$\vdash Link\mathbb{A}{\ll}As{\gg}[r{\ll}As{\gg} := \varnothing, s{\ll}Cl_A{\gg} := \varnothing, s{\ll}Cl_B{\gg} := \varnothing]$$
$$\equiv \text{[by def of } Link\mathbb{A}{\ll}As{\gg}\text{; schema calculus]}$$
$$\vdash \varnothing \in \mathsf{mult}[\varnothing, \varnothing]\ {\ll}multE{\gg}\ ({\ll}MS_1{\gg}, {\ll}MS_2{\gg})$$
$$\equiv \text{[by law of mult generic]}$$
$$\text{true}$$

Given that this is true, then,

$$\llbracket\ Link\mathbb{A}{\ll}As{\gg}'\ \rrbracket[\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ ][\llbracket\ s{\ll}Cl{\gg}' := \varnothing, st{\ll}Cl{\gg}' := \varnothing\ \rrbracket\ ] [\![\ r{\ll}As{\gg}' := \varnothing\ ]\!]$$

is obviously true.

$\exists\ SysInit \bullet \text{true}$

$\equiv$ [by T-$\exists$ Sc]

$\exists\ (System\ ' \llbracket\ \wedge\ \mathbb{S}\ll Cl\gg Init\ \rrbracket \llbracket\ \wedge\ \mathbb{A}\ll As\gg Init\ \rrbracket) \bullet \text{true}$

$\equiv$ [by T-$\exists$ Sc; $\wedge$-Id]

$\exists \llbracket\ \mathbb{S}\ll Cl\gg Init\ \rrbracket \llbracket\ ;\ \ \mathbb{A}\ll As\gg Init\ \rrbracket) \bullet$
$\quad SysConst\ '$

$\equiv$ [by dist-ext-init]

$\exists \llbracket\ \mathbb{S}\ll Cl\gg Init\ \rrbracket \llbracket\ ;\ \ \mathbb{A}\ll As\gg Init\ \rrbracket) \bullet$
$\quad SysConst\ '[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ]$

$\equiv$ [by dist-mult-ext-init]

$\exists \llbracket\ \mathbb{A}\ll As\gg Init\ \rrbracket) \bullet$
$\quad SysConst\ '[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ][\ \llbracket\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ \rrbracket\ ]$

$\equiv$ [by dist-mult-assoc-init]

$(\llbracket\ Link\mathbb{A}\ll As\gg'\ \rrbracket \wedge \llbracket\ Const\ll SConst\gg'\ \rrbracket)[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ]$
$\qquad [\ \llbracket\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ \rrbracket\ ][\llbracket\ r\ll As\gg' := \varnothing\ \rrbracket]$

$\equiv$ [by prop of substitution]

$\llbracket\ Link\mathbb{A}\ll As\gg'\ \rrbracket[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ][\ \llbracket\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ \rrbracket\ ]$
$\qquad [\llbracket\ r\ll As\gg' := \varnothing\ \rrbracket]$
$\quad \wedge \llbracket\ Const\ll SConst\gg'\ \rrbracket[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ ][\ \llbracket\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ \rrbracket\ ]$
$\qquad [\llbracket\ r\ll As\gg' := \varnothing\ \rrbracket]$

$\equiv$ [by conj and lemma]

$\llbracket\ Const\ll SConst\gg'\ \rrbracket[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing \llbracket\ , s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ \rrbracket$
$\qquad \llbracket\ , r\ll As\gg' := \varnothing\ \rrbracket\ ]$

■

Proof G.57 (Template T54, sys-init-ni)

$\llbracket\ Const\ll SConst\gg'\ \rrbracket \vdash \exists\ SysInit \bullet \text{true}$

$\equiv$ [by sys-init]

$\llbracket\ Const\ll SConst\gg'\ \rrbracket$
$\vdash \llbracket\ Const\ll SConst\gg'\ \rrbracket[\ s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing \llbracket\ , s\ll Cl\gg' := \varnothing, st\ll Cl\gg' := \varnothing\ \rrbracket$
$\qquad \llbracket\ , r\ll As\gg' := \varnothing\ \rrbracket\ ]$

$\equiv$ [eq-sub; schema calculus]

true

■