FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Making Object-Z Perfect

Verifikation von Object-Z Spezifikationen unter Verwendung von Perfect Developer

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsingenieurwesen Informatik

eingereicht von

Sylvia Swoboda, Bakk.techn.
Matrikelnummer 00225646

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ-Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Wien, 9. November 2017

Sylvia Swoboda — Gernot Salzer

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Making Object-Z Perfect

Verifying Object-Z Specifications Using Perfect Developer

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Wirtschaftsingenieurwesen Informatik

by

Sylvia Swoboda, Bakk.techn.
Registration Number 00225646

to the Faculty of Informatics

at TU Wien

Advisor: Ao.Univ-Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Vienna, 9th November, 2017

Sylvia Swoboda   Gernot Salzer

Erklärung zur Verfassung der Arbeit

Sylvia Swoboda, Bakk.techn.
Weintraubengasse 17/4, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. November 2017

Sylvia Swoboda

Acknowledgements

I want to take this opportunity to express my gratefulness to all those people, who made my academic studies possible, and especially enabled me to finish them.

First and foremost, I would like to thank my advisor, Gernot Salzer, for the inspiring topic of this thesis, which I found so exciting that I could hardly stop myself diving further into it. I appreciate all your critical and valuable feedback that helped improve this work, towards the finish even on short-notice.

I would like to express my gratitude to my companion in life, Marcus Meisel, for all his motivating words and his encouragement to finish my studies. His constructive feedback and experience in proofreading English texts, helped me enhance this work linguistically. Taking our three year old for some weekend trips during the last weeks, made it possible for me to tap into enough additional time to concentrate on the finalization of this diploma thesis.

I want to thank...
...my son Florian, for your patience during all the hours when mummy did not have time to play with you.
...my parents, Ernestine and Karl-Heinz Swoboda, for your constant support in all respects throughout the whole years of my studies. You kept away worries about money by financially supporting me and enabled me to dedicate even more weekends to my studies when I knew Florian was in Grandma's care.
...my sister Karin Malek for nagging me to finish my studies, but foremost for being a role model graduate.
...the family of my partner, especially Martin Meisel, Margit Meisel, and Renate Kraßnig, who also granted me enough time to juggle family, gainful employment, and completion of my diploma thesis and even spare time for activities to keep my work-life balance.

Finally, I want to thank all my friends and colleagues for words of motivation that encouraged me to get back on the track whenever I was in doubt.

Thank you very much indeed, without all of you, this would not have been possible.

Kurzfassung

Gerade in sicherheitskritischen Anwendungen ist die Korrektheit von Software ein maßgeblicher Faktor. Neben dem Ansatz mit statischen und dynamischen Testmethoden Fehler zu finden, kann eine formale Verifikation Fehler in der Software ausschließen. Die so durchgeführte Überprüfung bildet einen wichtigen Bestandteil festzustellen, ob Spezifikationen in sich schlüssig und korrekt sind. Eine der meist verwendeten Spezifikationssprachen ist die Z Notation. Mit ihrer objekt-orientierten Erweiterung Object-Z können auch komplexe Systeme in modularer Weise formal beschrieben werden. Die Toolunterstützung für die Verifikation oder gar eine Generierung von Programmcode beschränkt sich bislang primär auf Typüberprüfungen. Perfect Developer wurde mit dem Ziel erstellt, formale Spezifikationen automatisiert zu verifizieren, Implementierungen zu validieren und daraus Code zu generieren. Die Arbeit verknü beide Technologien miteinander, indem Object-Z Spezifikation automatisch nach Perfect überführt werden. Mit dieser Arbeit wird gezeigt werden, dass eine automatische Übersetzung der Object-Z Sprachkonstrukte in semantisch gleichwertige Konstrukte in Perfect möglich ist. Nach einer detaillierten Analyse von Syntax und Semantik der beiden Sprachen werden Regeln entwickelt, mit Hilfe derer die Sprachkonstrukte der Quell- in die Zielsprache übersetzt werden können. Bei der Ausführung ist es notwendig, die Originalspezifikation in mehreren Durchläufen zu analysieren, um alle für die Transformation relevanten semantischen Informationen in strukturierter Weise zugänglich zu machen. Anhand eines Fallbeispiels evaluieren wir die Umsetzung der Regeln. Da das entwickelte Tool auf GitHub öffentlich zur Verfügung steht, können nun gleichzeitig die Vorteile der übersichtlichen Object-Z-Notation und die automatisierten Verifikationsmechanismen von Perfect genutzt werden.

Abstract

Especially in security-critical applications, the correctness of software is a major factor. In addition to static and dynamic test approaches to find errors in a software or system, formal verification methods target on proving the correctness of software. It constitutes an important component of determining whether a specification is coherent and sound and is satisfied by an implementation. One of the most widely used specification languages is the Z notation. With its object-oriented extension Object-Z, it offers a possibility to formalize even complex systems in a modular manner. However, the tool support for the verification of specifications or even the generation of program code remains limited to type checking. Perfect Developer with its specification language Perfect was created with the idea of automatically verifying formal specifications, validating implementations against them and generating code. The idea of this work is to link these two technologies and to automatically convert a given Object-Z specification automatically into a Perfect specification making thus the functionalities of Perfect Developer available to the specifier. This work demonstrates that an automated translation of Object-Z language constructs into semantically equivalent constructs in Perfect is possible. After a detailed analysis of the relevant parts of the syntax and semantics of the two languages, we develop rules with the help of which the individual language constructs of the source language can be translated into the target language. It is necessary to analyze the internal representation of the original specification in several passes to obtain all the semantic information relevant to the transformation in a structured manner. With the help of a case study, we evaluate the implementation of the rules. The developed tool is available on GitHub.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

The idea of formally describing or specifying a system and then being able to not just generate code from the written requirements, but even verifying the correctness of the implementation with regard to the specification is a magical vision that is often wished for. However, even in times when artificial intelligence is all around, there are still limitations, especially as it is not only essential to prove correctness, but also make sure that the software works in the context that can be safety, security, or life critical.

1.1 Motivation

Whenever it is crucial that a system works correctly, because lives could be threatened in case of an error, the means how software is verified become even more important. If there is a formal specification of how a system should work, it is necessary to make sure that on the one hand this reflects how one actually wanted the system to behave and on the other hand take care that the implementation fully satisfies the specification. While testing is only able to prove the existence of bugs in a system with a certain code coverage, formal verification approaches may actually prove correctness.

However, as specifications evolve over time due to new insights or feature updates, instead of just being built as a whole upfront, it is important that the process of proving correctness can be done again and again after modifications have been added. Doing this manually is tedious work and especially makes it necessary for the person who performs this process to have sufficient skills and experience with formal methods while still not eliminating human errors. Therefore, tool support as comprehensive as possible is desirable.

Perfect Developer, as a commercial tool, was created to verify specifications and refinements. There is even the option of creating source code for the program to be designed. Providing these functionalities to creators of Object-Z specifications without the need of manually translating the existing specification into another language may bring the benefits of Perfect Developer to a broader audience. Even in other domains of software and system development a more formal way of specifying requirements can be an interesting topic, specially, if this way of describing the interactions of components and domain objects provides tool support for further work on the product itself, like code generation or verifying the quality of the product. If there is no obligation to use formal methods, this way of describing and verifying is usually considered as too expensive in comparison to informal approaches like drawing diagrams without a mandatory structure, testing, or reviews, as the implementers need to have the necessary skills to be able to actually work through the formal proofs manually.

The Z notation is one of the most renowned languages to support the formal description of a system. As software became more and more complex with the evolution of software development tools and techniques, also the need for an object-oriented approach on defining the behavior of a system arose. That was when Object-Z came into existence as an extension of the Z notation.

1.2 Problem Statement

Especially, in the domain of safety and security critical applications such as transportation (train, avionics, space travel), critical infrastructure (power gird, water supply, telecommunications), industry (heavy machinery, heavy long elevators, handling explosive atmospheres) or military (missile control systems, emergency response), there are often mandatory standards regarding formal specification and/or verification of the system, without which it would not even be possible to bring them into a productive environment.

Object-Z as an extension of the Z notation has some limited tool support, mainly provided by the *Community Z Tools*. There is assistance in writing Object-Z specifications using Unicode and LaTeX syntax. With regard to verification of an Object-Z specification, only a type checker is currently available. The Community Z Tools do provide a module to generate verification conditions for Z and some other Z dialects, unfortunately, Object-Z specifications are not yet supported by this functionality. Another tool called Wizard described by Wendy Johnston in [Joh96], facilitates type checking. Although Graeme Smith describes rules for reasoning about Object-Z specifications in [Smi95], and Wendy Johnston and Gordon Rose wrote a report [JR93] how to manually convert Object-Z specifications to C++, no implementations to automate the reasoning process or a conversion to some programming language are known.

Therefore, a conversion from Object-Z to Perfect is desirable to provide these functions to people who usually write their specifications in Object-Z or who intend to learn Object-Z and are already somewhat familiar with Perfect, a tool for transforming Object-Z specifications into Perfect, can help to gain a better understanding of the Object-Z specification language, especially its compact mathematical notations, if a mapping between the two languages is available.

1.3 Aim of the Work

The goal of this work is to provide a means to bridge the gap between Object-Z and Perfect. This means the computer-aided translation from Object-Z into a semantically equivalent specification in Perfect shall be done automatically. The resulting Perfect specification can then be used in the Perfect Developer environment with all its functionality (automated verification, code generation). This work shows how such a transformation can be done programmatically and points out obstacles and limitations by the capabilities of the language itself or due to the complexity of the transformation rules. The implemented tool and all its functionality is provided publicly at https://github.com/sylviaswoboda/objectz-2-perfect to make it available as foundation for future work as well as sparking new ideas and opening new opportunities for verified software in more domains.

1.4 Methodological Approach

The topic of this work is approached in several steps as illustrated in Figure 1.1. The first step is to thoroughly analyze the syntax and semantics of Perfect and Object-Z to gain an overview of the available language constructs and also to provide a deeper understanding of what each part of the specification language means and which constructs are available in which context. Following the necessary detailed context, mapping rules for each single language construct of Object-Z are developed and described in detail. These rules aim at being applicable in every possible context. However, whenever it is not feasible to cover all special cases, these limitations are added to the rule descriptions. The next step is creating an implementation of all the given rules to translate Object-Z specifications into Perfect. The arrow pointing back to the previous step depicts that this step also has an influence there, because rules have to be adapted due to implementation constraints. The next step is to transform an instructive example using the provided implementation to demonstrate which verifications will be processed by the Perfect Developer verification engine This verification step also influences either only the implementation or even the transformation rules in the sense of a feedback loop to improve design or implementation iteratively. The final step is an evaluation of the whole work, especially with regard to related work.

1.5 Outline of the Document

Chapter 2 provides a discussion of the state of the art and related work. Afterwards a short introduction to the Object-Z language is provided in Chapter 3, presenting the language characteristics and features as an overview as far as they are further considered in this work. More detailed insight into the semantics of each language construct will be given in Chapter 5. Chapter 4 provides an introduction to the Perfect language in the same way as for Object-Z.

Chapter 5 describes the requirements and transformations that are necessary to implement

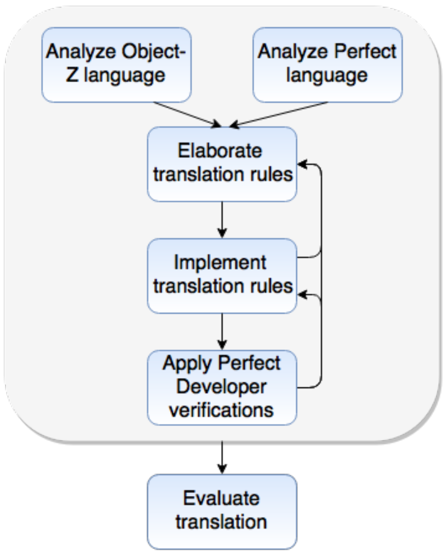

Analyze Object-Z language
Analyze Perfect language
Elaborate translation rules
Implement translation rules
Apply Perfect Developer verifications
Evaluate translation

Figure 1.1: Methodical steps of this work

a mapping from Object-Z to Perfect, which aims at maintaining the semantics defined by Object-Z. Chapter 6 describes in detail how the mapping requirements defined in Chapter 5 have been implemented, which tools have been used for implementation, and how the tool can be used. Chapter 7 presents the results that can be achieved by using the Object-Z to Perfect mapping tool as well as the limitations that have become visible by applying the tool to exemplary Object-Z specifications. The thesis is concluded with a brief summary and a discussion comparing the findings of this work to the state of the art.

CHAPTER 2

State of the Art

This chapter provides an overview of formal specification and verification. A first insight into the languages Object-Z and Perfect is given in the context of formal specification and verification, while also discussing already existing tooling approaches supporting the use or verification of these languages. The chapter ends with a presentation of related work, especially with regard to language transformations.

2.1 Formal Specification and Verification

Formal specification aims at describing the behavior of a system using mathematically based techniques. Syntax and semantics of the used specification language are well-defined and unambiguous. Formal verification, on the other hand, tries to find an answer to the question whether a program is correct with regard to a given formal specification.

Although the process of formalizing requirements given in natural language takes some effort, there are also benefits from having a formalized representation. First, the formal specification may serve as a documentation of the system without ambiguities that cannot be prevented in natural language. Even the process of representing the requirements correctly itself can help in finding inconsistencies that would otherwise stay unnoticed. Therefore, having a formal specification also influences the design of a system, as it is necessary to clarify inconsistencies earlier in the development process. Additionally, having a formal specification builds the necessary basis if any kind of formal reasoning regarding the correctness of the design shall be performed.

Formal specification and verification techniques have been around for more than 40 years, but only safety- or security-critical system normally make use of these techniques, especially when it comes to verification. Sometimes, the reason is the lack of knowledge of formal methods by the acting persons (software developers), sometimes it is because the effort does not seem worthwhile.

2.2 The Language Object-Z

The specification language Object-Z[1] is based on the Z notation. Jean-Raymond Abrial first used the Z notation in his work [Abr74, Data Semantics]. Spivey gives a comprehensive description of this specification language in [Spi89, The Z Notation - A Reference Manual]. An introduction how to use Z is provided by Lightfood in [Lig01]. The [ISO02, ISO Standard of Standard Z] published in 2002 specifies Z formally. Both languages use a graphical notation with named boxes that structure the building blocks of the specification.

In the early nineties the need for expressing object-oriented specifications arose [SRBC92] and the Object-Z notation was introduced as an object-oriented extension to the Z notation [CS90]. It was developed at the Department of Computer Science at the University of Queensland with financial support of the Australian Overseas Telecommunication Corporation(AOTC). The first basic semantical approaches to Object-Z were presented by D. Duke and R. Duke in [DD90]. A first full version was published in 1991 by R. Duke, King, Rose and Smith in [DKRS91] and defined in more detail in Smith's PhD thesis [Smi92]. Certain language constructs changed later on and the semantical and logical foundations for Object-Z were introduced by Smith [Smi94], [Smi95b], [Smi95a], and Griffith [Gri95], [Gri96]. Additionally, the basis for reasoning about and formally verifying Object-Z specifications has been laid by Smith in [Smi95d] and [Smi95c], and extended together with Winter in [WS03]. The language version as it is presented in [Smi00] and [DR00] can be considered as the current version. That is why this thesis is based on the language definition as described in these two books.

Wendy Johnston described a command line tool for type checking Object-Z specifications called Wizard in a technical report in 1996 [Joh96]. Unlike Z where several tools for creating and checking of specifications exist, Object-Z has essentially one main source for tool support: the Community Z Tools[2], a sourceforge project providing a collection of modules that offer different kinds of functionality. For Object-Z, there is a module "corejava" providing java classes for annotated syntax trees, a parser, a type checker and IDE support via a customized Eclipse bundle or Eclipse and jEdit plugins.

2.3 Perfect Developer and the Language Perfect

Perfect Developer was previewed in 1999 as Escher Tool and commercially released in July 2002 [CC04, chapter 8] by Escher Technologies. The goal in developing Perfect Developer was to combine modern component-based, object-oriented software development with the benefits of formal methods and at the same time making developer productivity at least as efficient as with conventional approaches using informal techniques, relying only on testing.

[1] Object-Z Homepage, `http://staff.itee.uq.edu.au/smith/objectz/objectz.html`
[2] `http://czt.sourceforge.net`

Perfect Developer is based on the concepts of formal specification and refinement. That means that first the requirements for the software are defined in a formal way and then refinement can either be done automatically for most of the specifications or provided by the user by specific refinement expressions. Perfect Developer can then verify whether the specification is correct with respect to the generated verification conditions (section 2.3.3) and whether the user-defined refinement satisfies the conditions implied by the specification. As a last step Perfect Developer supports automated code generation in a number of target languages such as Java, C++, and C#.

2.3.1 Value Semantics vs. Reference Semantics

Most modern object-oriented programming languages like Java, implement *reference semantics* for object handling. That is, a variable does not hold the entire object directly, but rather a reference to that object. The object itself is stored somewhere else, which may improve performance, e.g. if references are assigned to other variables, but also gives rise to problems caused by numerous references to the same object called *aliasing*. Each reference to an object may modify it. Controlling or keeping track of such changes gets more and more difficult with the increasing number of references to an object.

Another approach for handling objects is the use of *value semantics*. Here, the object is not considered to have its own object identity, but this identity is rather built up of the values of its containing member variables. Objects are directly connected to a variable and whenever an object is duplicated, this copy is different from the original. Changes to the original are not reflected in the new object.

Perfect adopts the concept of value semantics as the default when declaring variables. Crocker and Carlton describe in [CC04, section 2.4] why value semantics are sufficient and often even more natural and desirable although parameter copying leads to additional overhead. However, if the need for object references arises, Perfect also supports reference semantics.

2.3.2 Verified Design by Contract

The Perfect language together with its verification engine integrated in Perfect Developer bases on the principle of *Verified Design by Contract*. This term has been introduced by David Crocker and discussed in detail in [Cro04]. It bases on *design by contract*, a principle that goes back to Floyd-Hoare Logic [Hoa69], but the term has been introduced by Bertrand Meyer of Interactive Software Engineering in [Mey97].

The basics of Design by Contract can be summarized as follows. We have a call to a method S and a postcondition R, which holds after executing S. However, to execute S the precondition P has to hold. Given P, R, and S, one can be sure that R holds after execution of S only if the following statements are true:

- If P holds initially and S is executed, then this will always result in a state where postcondition R is satisfied.

- Each time immediately before S shall be executed, precondition P holds.

For the method S itself this means that two contracts have to be fulfilled. First the server (where the method is defined) guarantees that whenever the precondition P is satisfied and the method is called, then the call will result in a state satisfying R. Second, all callers (clients) guarantee that they only call the method in a state where P holds and therefore can be sure that R holds after execution of S.

The principle of design-by-contract can also be applied to contexts where objects are dynamically bound, e.g. for inheritance hierarchies. In this case it must be allowed that objects of subclasses can be used whenever an object of a superclass is expected. This implies that in subclasses of these hierarchies preconditions may be loosened, whereas postconditions may be strengthened, i.e. parameter values and their types may be widened and return types and their values may only be constrained additionally.

Verified design by contract in Perfect adapts this approach by creating verification rules to check for validity of such contracts. In Perfect postconditions are considered complete, i.e. they specify what variables are changed by a method and how they are modified. Postconditions as described above for design by contract are implemented in Perfect by post assertions which state all conditions that a client may assume after a call to some certain method. Postassertions must state a logical implication of the postcondition and in contrast to postconditions they are inherited to derived classes.

2.3.3 Verification Conditions

To verify the correctness of the specification a total of over 50 different verification conditions are created. A comprehensive list of them can be found on the Escher Technologies website[3]. A small selection of them is presented here to give the reader an impression of the verification capabilities of Perfect Developer.

Class invariant satisfied: Checks for the postcondition of each modifying schema or constructor that the class invariants remain satisfied.

Precondition of *op* satisfied: Checks for any call to a function, operator, selector or schema with a precondition, if the precondition is satisfied.

Expressions modified by schema are independent: Checks for each call to a schema modifying more than one object that these objects are independent.

Objects modified in parallel are independent: Checks for parallel postconditions that the objects modified are independent.

Operand of 'over' has at least one element Checks whether the collection on which *op* **over** is performed is not empty.

[3]http://www.eschertech.com/product_documentation/Verification Conditions Generated by Escher Verification Studio.pdf

At least one guard is true: Checks for conditional expressions and postconditions with no empty guard if there is at least one option that may be executed, because the guard is true.

Assertion satisfies inherited assertion: Checks for methods declared with **define** or **redefine** and an assertion whether the new assertion implies the original assertion.

Inherited precondition satisfies new precondition: Checks for methods declared with **define** or **redefine** and a precondition, that the original precondition implies the new precondition.

Type constraint satisfied: Checks for parameters, return values, variables and objects modified by a postcondition whether the type constraints are satisfied.

After having tried to verify all created verification conditions, the Perfect verification engine outputs information about which conditions could not be proven correct and gives some hints where the cause of the problem lies. Examples for verifier output will be provided in Chapter 7. Syntax and semantic of Perfect will be presented in more detail in Chapter 4.

2.4 Related Work

Several people have already begun mapping Object-Z specifications into other notations. Paige and Brook describe in [PB04] an approach to integrate BON and Object-Z to provide a means of taking a specification one step further into the actual software development process. The papers of Rafsanjani and Colwill [RC92], Johnston and Rose [JR93], and Fukagawa, Hikita and Yamazaki [FHY94] target a similar issue. All of them developed ways or guidelines of converting Object-Z specifications to the C++ programming language, so that executable code can be easily created from a given specification. Soon-Kyeong Kim and David Carrington have concentrated on finding a formal mapping between UML class diagrams and Object-Z [KC99], [KC00], and [KC02].

Another idea of using Object-Z specifications in the software development process is to derive test cases from the specification. MacDonald, Murray, and Strooper published their first work on creating object-oriented test oracles in 1997 [MMS97] and elaborated their work during the following years in [MS98], [CMM$^+$98], [MCMS99], and [MSH03] together with Carrington, MacColl, and Hoffman. Ashraaf and Nadeem [AN06] also proposed a technique for automated test case generation from Object-Z specifications.

Regarding Perfect, there have also been some efforts to provide conversions from Object-Z. Brian Stevens described how to implement Object-Z specifications in Perfect in [Ste06], but this approach is rather high level and demonstrates how to map the Object-Z structures to Perfect without the in-depth inspection of each language construct necessary to create an implementation for an automated conversion.

Tim Kimber presented the most thorough discussion on how to map Object-Z to Perfect in [Kim07]. In his work, he aims at constructing a tool for automatically translating an Object-Z specification given in the so-called OZ notation of Object-Z. The OZ notation has been introduced by Kimber to enable the user to easily edit an Object-Z specification within an ASCII text file, where the good-looking and clearly arranged, but for specification rather unhandy, Object-Z box notations are substituted by curly brackets like they are used in C, C++, or Java. The mapping is described in much detail, but leaves some language functionality unmapped like generics, inheritance, or distributed operators. Furthermore, the tool that has been implemented based on the mapping instructions has neither been made publicly available, nor could be retrieved on request. However, his work is a good starting point to provide a comprehensive mapping of the Object-Z language features.

2.5 Tools for Language Transformations

As the aim of this work is to provide a tool for automated translation from Object-Z to Perfect, publicly, it is necessary to decide for an implementation language. To make the tool available for a broad audience, both for using the tool as it is and for extending its foundation, the decision should be for a widely used programming language that supports at least Windows and Linux to enable the use for Perfect developer users on both platforms. Therefore, and due to the author's experience with it, Java[4] has been chosen as implementation language.

Apart from the programming language, another important aspect of the translator are a lexical analyzer (lexer) and a parser for language recognition and syntactic analysis of the source language. As the recognition part as such is not in the focus of this work, the parser generator ANTLR was chosen to be used for this purpose. ANTLR[5] has been developed by Terence Parr since 1989. The project started as a parser-generator for LL(1) parsers and evolved during years and over four major versions into a tool that generates parsers with arbitrary look-ahead, so called LL(*) parsers as they process their input from left to right and construct the leftmost derivation of a particular grammar rule. Parr [Par07] includes a thorough discussion on this topic in block three of 'The Definitive ANTLR Reference'. Together with Fisher [PF11] he also provides a foundation on LL(*) parser generators.

Language recognition can be performed in multiple levels by first analyzing the input with a lexer, resulting in input tokens. They are processed by grammar rules constructing an abstract syntax tree (AST). This can then in return be transformed further into templates or some other user-defined internal representation of the input. ANTLR supports various code generation target languages[6] such as Java, C++, C#, and JavaScript. In version

[4] `http://www.oracle.com/technetwork/java/index.html`

[5] `http://www.antlr.org`

[6] ANTLR Runtime Libraries and Code Generation Targets, `https://github.com/antlr/antlr4/blob/master/doc/targets.md`

4 of ANTLR [Par13], the concept of how the abstract syntax tree can be analyzed has been changed to enable a separation between grammar and application code. This is especially helpful, if several applications should be built on top of the same grammar. Terence Parr introduced the listener and visitor patterns to react to the situation when the traversal passes a node in case of the listener or to even control the traversal flow in case of the visitor.

Another project, that can be smoothly integrated with ANTLR, is the StringTemplate project[7]. The initial goal of StringTemplate was to separate business logic and displaying data on webpages by enforcing a strict separation of model and view by means of template engines. A single template is essentially a character string with a certain amount of specially marked tokens that may later on be replaced by variables or even sets of variables. For example, one could use a placeholder for the username on a website template. This is then substituted by the username of the person who is actually initiating the web request or the logged in user.

Templates may be either defined directly in the implementation code or read from a so called *StringTemplateGroup*-file, which gathers all the necessary templates used in a certain application. This enables separation of the program functions from the actual presentation layer. Such StringTemplates can be used to manage the static or repeating parts within the mapping from Object-Z to Perfect like the basic structure of a class or an operation.

After the discussion of the context of this work the following two chapters will present the language constructs of Object-Z (chapter 3) and Perfect (chapter 4) in more detail.

[7] `http://www.stringtemplate.org`

CHAPTER 3

Object-Z

To enable readers to follow further findings in this work, one has to understand the language Object-Z. This chapter presents descriptions of the language constructs added to Z to support object-orientation in Object-Z. However, as the focus of this work is on the object-oriented aspects of the work, the necessary descriptions for those parts of the language that have already been introduced in Z and thoroughly discussed in Spivey's Z reference manual [Spi89], will only be presented in Chapter 5 in combination with the mappings to Perfect.

3.1 The Class Construct

A class is a named box and consists of several graphical elements like named and unnamed boxes. Figure 3.1 shows the general structure of an Object-Z class.

$\mathit{ClassName}[\mathit{FormalParameters}]$
$[\mathit{VisibilityList}]$
$[\mathit{InheritedClasses}]$
$[\mathit{LocalDefinitions}]$
$[\mathit{StateSchema}]$
$[\mathit{InitialStateSchema}]$
$[\mathit{Operations}]$

Figure 3.1: Structure of an Object-Z class

The class is surrounded by a border which limits the scope of the class. In the line at the top of the frame the name of the class is given first, optionally, a list of formal parameters can be specified in square brackets. All other components of the class, visibility list, inherited classes, local definitions, state and initial state schema, as well as operations are situated within the borders of the class frame. Neither of these structures need to be included in a class definition.

$Queue[Item]$

$\upharpoonright(count, INIT, Join, Leave)$

$c : \mathbb{N}$

$c \in 100, 200, 300$

This is a comment.

$items : \mathrm{seq}\, Item$
$count : \mathbb{N}$

$count \leq c$

$INIT$

$items = \langle\, \rangle$
$count = 0$

$Join$

$\Delta(items, count)$
$item? : Item$

$items' = items \frown \langle item? \rangle$
$count' = count + 1$

$Leave$

$\Delta(items)$
$item! : Item$

$items' = \langle item! \rangle \frown items'$

Figure 3.2: Example of an Object-Z class construct: Queue

To illustrate the inner structure of these components a specification of a generic *Queue* is given in Figure 3.2, this specification is adopted from the queue example in [Smi00]. The following sections of this chapter provide a description of the building blocks of an Object-Z specification to give a deeper insight into this notation.

3.2 Genericity

In the Queue class example in Figure 3.2 the formal parameter *Item* is provided to the class definition. This generic parameter, also called formal or template, is needed to provide template functionality for the class. That means, a class *Queue* is defined once without knowledge of the actual parameter type, and the element type of *Item* is only fixed at instantiation time. When a new object is declared, the actual class names for the formal template parameters are handed over.

Instantiation examples are $natQueue : Queue[\mathbb{N}]$ for queues of natural numbers and $messageQueue : Queue[Message]$ given that a type *Message* has already been declared. As all the operations of the class are defined using the generic parameter *Item*, all of them can be used for elements with the actual type of the parameter, like $\mathbb{N}$ or *Message* in the objects declared earlier.

3.3 Visibility List

In Figure 3.2 the first line inside the class box shows the visibility list of the specified class. As indicated by the name, the visibility list determines which features of a class are visible to the environment. In this example the visibility list defines that only the state variable *count*, the *INIT*-schema, and the two operations *Join* and *Leave* can be accessed from other class specifications. Leaving out the visibility list states that all features are visible to the environment.

Visibility in Object-Z comprises read and write access to a feature of a class. As visibility lists are not inherited, the inheriting class has to define its own interface. The list of features in an inheriting class may contain features directly defined in that inheriting class as well as any of the features of the super classes because all features are inherited.

3.4 Inheritance

Object-Z supports single and multiple inheritance. The name of each class which shall be inherited is specified in the class section directly after the visibility list.

SomeInheritingClass

...

$SuperClass1[newVarName/varName, newOperationName/operationName]$

$SuperClass2[\mathbb{N}]$

...

Figure 3.3: Schematical example of inheritance in Object-Z

Figure 3.3 schematically shows how inheritance is declared in Object-Z classes. *SomeInheritingClass* inherits all the features (member variables, constants and operations) of the two classes *SuperClass1* and *SuperClass2*. If both classes happen to contain equally named attributes, these features have to be type compatible for the specification to be consistent. If an operation with the same name is included in a subclass the operation schema of the superclass and the operation schema of the subclass are conjoined. Auxiliary variables also have to be type compatible.

If features are not type compatible or should be regarded as two separate features, it is possible to rename the feature of the superclass like this is done for *varName* and *operationName* of *SuperClass1* in Figure 3.3. If an inherited class has been defined generically, actual parameters must be provided like the parameter $\mathbb{N}$ for class *SuperClass2*. If the inherited class is also declared with a formal parameter, this parameter could also be provided to the inheriting class.

3.5 Polymorphism

Object-oriented polymorphism allows an object typed as class C to belong to other classes as well, and therefore the object may also behave as if it belonged to these other classes.

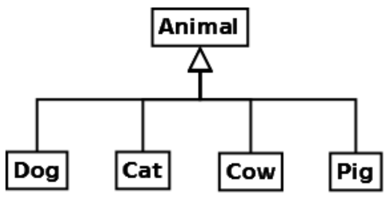


Figure 3.4: Animal hierarchy (simplified based on the animal hierarchy in [Hor00, p. 248])

The animal hierarchy in Figure 3.4 shows the class *Animal* and its four subclasses *Dog*, *Cat*, *Cow*, and *Pig*. If polymorphism is available, an object of one of the four subclasses may as well act as *Animal* object. Class *Animal* might declare a function *makeSound* that gives the characteristic sound that an animal makes, but each of the subclasses may return a different sound depending on the type of animal.

In Object-Z, the term $a : \downarrow Animal$ declares a polymorphic variable. For the hierarchy above, variable a may reference objects of the classes Animal, Dog, Cat, Cow, and Pig. a may also call all the features provided by class Animal using the expression $a.feature$.

However, as already described in the visibility section 3.3, features may be hidden from the environment by not including them in the visibility list. If the operation *makeSound* was only included in the visibility lists of classes *Animal*, *Dog*, and *Cat*, a call of the operation could be dynamically invalid. In this case e.g., a *Cow* object referenced by *a* does not have the *makeSound* operation and this feature could consequently not be applied.

According to [DR00, p.30 f] a call to a feature is well-formed only if:

- Each attribute, operation or the initial state schema in the visibility list of a superclass A is also included in the visibility lists of each subclass of A.
- For visible operations, the communication parameter must be the same, i.e. identically named and typed, in class A and in all subclasses.

Object-Z provides an even more generalized view of polymorphism. Using the union operator $\cup$ several classes may be united to form a new type.

$$Pet == Cat \cup Dog$$

This expression unites the classes Cat and Dog from the animal hierarchy to form the new type *Pet*. For this construct, attributes *a* and operations *op* are said to be in the *polymorphic core* of the class union ([DR00, p.118] if

- attribute *a* is defined in each of the defining classes with the same name and type.
- operation *op* exists in each of the defining classes and the communication parameters are the same in each class, i.e. name and type are identical.

For all features, i.e. attributes and operations, in the polymorphic core the expression *var.feature* for *var* of the union type can always be semantically interpreted. In contrast, for such an expression to be statically well-formed only one of the classes in the class union need to have *feature* in its visibility list.

3.6 Renaming and Hiding

As input and output parameters of operations are equated to equally named and typed variables of the environment, it may sometimes be advantageous to have the ability to rename a variable to overcome naming clashes, for example. In the inheritance example in Figure 3.3 above, SuperClass1 is used together with a renaming statement that renames a variable and an operation.

The general syntax of this renaming construct is as follows.

$$[newName/oldName]$$

This may be used whenever class descriptors or operation expressions are used. The first are used for the definition of inherited classes or whenever a variable is declared to have some class type. The second appears in operation expressions (see sections 3.10.2 and 3.10.3), where they are primarily used to enable or disable equating of variables with the same base names. The construct means that all occurrences of *oldName* in the class or operation expression are substituted by *newName*. A comma separated list of such renaming pairs may be provided in one renaming expression.

The *hiding* construct is only available for operation expressions.

$$operationExpression \setminus (nameList)$$

It means that all the identifiers in the comma separated list *nameList* have to be removed from the set of identifiers available to the operationExpression. This construct may be used to hide some particular input or output variables, for example to achieve that parameters are not equated or that the operation built by an operation promotion does not have input or output variables anymore.

3.7 Local Definitions

In the part following the visibility list of the specification in Figure 3.2 an *axiomatic definition* is introduced. This construct is marked by a vertical line on the left side of the specification text. In the part above the horizontal line local constants can be introduced. Predicates written below the line specify restrictions that hold for these constants. In the given example the predicate states, constant c must have one of the values 100, 200, or 300. This shows that the value of a constant does not need to be globally fixed, but it remains constant for one particular object of the class, once the instance has been created and the constant has therefore been initialized.

Besides simple constants also constant functions may be defined in an axiomatic definition. Constant functions may be regarded as functions, that are totally defined by the predicates in the axiomatic definition and can never ever be altered later on, i.e. the value that is mapped to each element of the function domain is fixed on object creation time.

Axiomatic definitions, *given type definitions*, *abbreviation definitions* and *free type definitions* form the set of local definitions. The characteristics of all of these are summarized below. All kinds of local definitions are also normal Z constructs and therefore can be used outside the class construct as well. For these local definitions the scope, in which the introduced identifiers are known, is the whole specification starting from the point of declaration.

Given/Basic type definitions introduce a number of new types. Defined within a class this type can be used throughout the scope of the class definition starting from the point of definition. The type identifiers are separated by ',' and enclosed in square brackets like in the following example.

$$[BasicType1, BasicType2]$$

Abbreviation definitions introduce new identifiers that can be used instead of some arbitrary expression and are written as:

$$Identifier == Expression$$

The newly introduced *Identifier* may be used wihtin the class in which it is declared.

Free type definitions introduce types with a fixed set of values that are allowed for this type. On the right hand side a list of branches is given, that describe the values of this free type.

$$Colours ::= Blue \mid White \mid Yellow \mid Green \mid Red \mid Black$$

The new type can be used starting from the declaration until the end of the class definition.

Axiomatic definitions introduce constants and constant functions. The properties of these constants and functions are defined by the predicates given in the predicates section in the part below the horizontal line.

Declarations
Predicates

Unlike all other local definitions, identifiers declared in axiomatic definitions may also be included in the visibility list of a class to make them available to the environment.

3.8 State Schema

The state schema introduces instance variables, also called *primary variables* which form the state of the object. State schemas are always marked by an unnamed box and can be separated by a horizontal line into two sections, the declaration and the predicate part, in the same way as axiomatic definitions. The first part introduces the names and types of the state variables the second part states restrictions that always have to hold for these variables. Variables may have primitive types like integer, natural, or real, collection types like set, sequence, or bag, user-defined types introduced by free and basic type definitions, classes, or functional and relational types.

The section of variable declarations may once again be separated into a part for primary and one for secondary variables, separated by a Δ symbol. The values of *secondary variables* are defined by a fixed relationship between some of the primary variables or constants. This also means that the value of secondary variables is normally not changed directly, but rather changes whenever one of the defining primary variables changes its value.

The part after the horizontal line, the *predicate* section, contains one or more predicates that define constraints for the state variables. These invariants always have to hold, even if the value of a state variable is changed by an operation. The postcondition of operations can be considered to implicitly contain the class invariants such that the invariant is satisfied before and even after execution of the operation.

Queue

c is a constant.

$items : \mathrm{seq}\, Item$
$count : \mathbb{N}$

$count \leq c$

Figure 3.5: State schema of class Queue in Object-Z

In Figure 3.5 the snippet of the specification of class Queue declaring the state schema is given. Two primary variables *items* and *count* are declared. The second variable *count* is restricted to have a smaller value than the constant *c*. Predicates in the predicate section may be built as simple as in this example, but they may also use disjunctions, implications, negations, or equivalences as connectives. The restrictions may even be expressed by existential or universal quantification expressions.

3.9 Initial State Schema

In Figure 3.6 the initial state schema is a box named with the keyword *INIT* in its top border line. Its specification text may refer to all already introduced, locally defined identifiers including constants and state variables. It describes the initialization condition, the state which must hold for an object of the class on creation time. The predicates given by the INIT schema may also hold later on, when the object has already changed several times. Therefore, the statement *object.INIT* says that *object* satisfies all the constraints specified by the initial state schema.

Queue

INIT

$items = \langle\, \rangle$
$count = 0$

Figure 3.6: Initial state schema of the class Queue

In the example in Figure 3.6 two predicates are provided to define the initial state of an object of class *Queue*. First, variable *items* is specified to be an empty sequence and second, variable *count* must have value 0. Whenever both of these two conditions hold, a Queue object satisfies the initialization condition.

3.10 Operations

In Object-Z there are three kinds of operations:

- Operation schemas
- Operation promotions
- Operation expressions

A description of each of these is given in the following subsections. All three describe a relationship of the state variables before and after the operation has been applied. That is, they state in a declarative manner the conditions that the state variables have to satisfy, such that the operation can be applied and which conditions hold afterwards. However, the way how to get to this state is not specified like in imperative programming languages.

3.10.1 Operation Schemas

The most basic form is the operation schema, a named box that may be split by a horizontal line into two parts like state schemas. The building blocks of operation schemas are described below.

Deltalist : Lists all the primary variables that may be changed by this operation schema. If present, it is the first part of the operation schema. The Δ symbol is followed by a list of identifiers in parentheses.

Declarationlist : In the declaration part auxiliary variables can be specified. By convention input variables are marked with a question mark, output variables end with an exclamation mark. Both of these form the means of communication with the environment. Auxiliary variables may also be undecorated, but these variables are only used within an operation schema.

Predicatelist : The predicate section contains restrictions that must hold for the operation to be applicable, i.e. preconditions, and it states the conditions that must be true after the application of the operation schema, i.e. postconditions that describe the state transitions for objects of a class.

Changing primary state variables that are not included in the deltalist results in an inconsistent specification. Values of secondary variables may change without including them into the deltalist of an operation schema.

In the Queue example in Figure 3.7, *Join* and *Leave* are operation schemas. *Join* may change the variables *items* and *count* and has an input parameter *item?* of the generic parameter type *Item*. The two predicates describe the transitions of the state variables.

Queue

Join

$\Delta(items, count)$
$item? : Item$

$items' = items \frown \langle item? \rangle$
$count' = count + 1$

Leave

$\Delta(items)$
$item! : Item$

$items = \langle item! \rangle \frown items'$

Figure 3.7: Example of operation schemas: Join and Leave

A primed variable like *items'* denotes the state of the variable after the operation schema. *Leave* has only one variable in its deltalist. Instead of an input variable this operation declares an output variable *item!*. The predicate of *Leave* only contains a postcondition (indicated by the primed and the output variable) that states that *items* before the operation must be equal to taking the output parameter *item!* concatenated with the state variable *items* after the operation.

As already indicated in section 3.8, operation schemas implicitly also satisfy all the class invariants as preconditions of the operation and can therefore restrict the states in which an operation is applicable. Additionally, the class invariants modified so that all occurrences of primary variables are primed have to hold after the operation.

For the example, in Figure 3.7, this means that the following predicates also have to be satisfied for the operation schemas *Leave* and *Join*.

$$count \leq c$$
$$count' \leq c$$

This does in fact add the restriction to operation *Join* that *count* has to hold a value strictly smaller than c before the operation so that the second predicate may be satisfied after the operation.

3.10.2 Operation Promotion

If an operation of a class should only propagate the exact behavior of an operation of one of its state variables, then *Operation Promotion* is used. To illustrate operation promotions the Book and SmallLibrary examples shown in Figures 3.8 and 3.9 were created for this work. The two classes model a simple and small library that administrates the handling of only two books.

First the basic type *String* is introduced. Class *Book* allows a person to *lend*, *return* or *review* a book. Only a limited amount of recensions is saved per book and if the number of saved entries exceeds the limit, the oldest one is thrown away. The latest recension can be accessed by the schema *latestRecension*. Additionally, the reader history is stored and can be accessed through *readerList*. A book can only be returned and reviewed by the current lender. Reviews can only be added, while the book is lent. The list of authors may be set or retrieved from a book.

The operations *lendBook1*, *lendBook2*, *returnBook1*, and *returnBook2* propagate the operations *lend* and *return* of the books respectively. As both operations are also in the visibility list of class *SmallLibrary*, these operations are accessible for the environment. As neither *book1* nor *book2* are listed as visible, access to the objects or any of their features would be prohibited.

Book

$\upharpoonright(lend, return, review, latestRecension, authorList, readerList, setAuthorList)$

$[String]$

$$\begin{array}{|l}
maxHistoryEntries : \mathbb{N} \\
\hline
maxHistoryEntries \in \{10, 20, 50, 100\}
\end{array}$$

$$\begin{array}{|l}
title : String \\
authors, recensionHistory, readerHistory : \operatorname{seq} String \\
lent, reviewed, mayBeReviewed : \mathbb{B} \\
\Delta \\
totalLendingCount : \mathbb{N} \\
\hline
\#recensionHistory \leq \#readerHistory \\
\#recensionHistory \leq maxHistoryEntries \\
totalLendingCount = \#readerHistory \\
authors \neq \langle\, \rangle
\end{array}$$

INIT

$$\begin{array}{|l}
recensionHistory = \langle\, \rangle \\
readerHistory = \langle\, \rangle \\
\neg\, lent \wedge \neg\, reviewed \wedge \neg\, mayBeReviewed
\end{array}$$

latestRecension

$$\begin{array}{|l}
recension! : String \\
\hline
reviewed \\
recension! = last\ recensionHistory
\end{array}$$

setAuthorList

$$\begin{array}{|l}
\Delta(authors) \\
authors? : \operatorname{seq} String \\
\hline
authors' = authors?
\end{array}$$

authorList

$authors! : \operatorname{seq} String$

$authors! = authors$

readerList

$readers! : \operatorname{seq} String$

$readers! = readerHistory$

return

$\Delta(lent, mayBeReviewed)$
$reader? : String$

$lent$
$reader? = last\ readerHistory$
$\neg\ mayBeReviewed' \land \neg\ lent'$

lend

$\Delta(readerHistory, lent, reviewed, mayBeReviewed)$
$reader? : String$

$\neg\ lent$
$mayBeReviewed' \land lent' \land \neg\ reviewed'$
$readerHistory' = readerHistory \frown reader?$

review

$\Delta(reviewed, recensionHistory, mayBeReviewed)$
$recension?, reader? : String$

$\neg\ reviewed \land mayBeReviewed$
$reader? = last\ readerHistory$
$reviewed' \land \neg\ mayBeReviewed'$
$((\#recensionHistory = maxHistoryEntries \land$
$recensionHistory' = tail\ recensionHistory \frown \langle recension? \rangle) \lor$
$(\#recensionHistory < maxHistoryEntries \land$
$recensionHistory' = recensionHistory \frown \langle recension? \rangle))$

Figure 3.8: Example of class with operation schemas: Book

Input and output variables of the propagated operations are used as communication variables for the new operation. In this case only the single input variable *reader?* is available as an input variable to all the four operations. Although changes are performed to the inner state of *book1* or *book2*, there are no changes to the *SmallLibrary* object, in particular the references *book1* or *book2* are not altered. This is the reason why there is

SmallLibrary

$\upharpoonright(postRecension, lendBook1, lendBook2, returnBook1, returnBook2, ...)$

$book1, book2 : Book$
$recensionsWall : Book \nrightarrow String$
Δ
$books : \mathbb{P}\, Book$

$book1 \neq book2$
$books = book1 \cup book2$

INIT

$recensionsWall = \varnothing$

postRecension

$\Delta(recensionsWall)$
$book? : Book$
$recension? : String$

$recensionsWall' = recensionsWall \oplus \{book? \mapsto recension?\}$

$lendBook1 \mathrel{\widehat=} book1.lend$
$lendBook2 \mathrel{\widehat=} book2.lend$
$returnBook1 \mathrel{\widehat=} book1.return$
$returnBook2 \mathrel{\widehat=} book2.return$

Figure 3.9: Example of operation promotions: SmallLibrary

no need for a deltalist in such operations. Operation promotions do not change state variables of objects of the class in which the operation promotion is defined.

3.10.3 Operation Expressions

Operation expressions or operation operators as they are called by Smith [Smi00], introduce the ability to combine several operations together to form a new indivisible operation. These operation operators originate from the schema operators that are available in Z to build more complicated operations out of simple ones like schema composition and schema disjunction [Lig01, chapter 6], [Spi89, chapter 3.8].

The following six kinds of operation expressions exist in Object-Z:

Conjunction: This operator models that two operations that are performed simultaneously. Communication between the two operations is not possible. The operation schemas of the two constituting operation schemas are conjoined, which means that

SmallLibrary

$$
\begin{array}{l}
lendBoth \mathrel{\widehat{=}} book1.lend \land book2.lend \\
returnBoth \mathrel{\widehat{=}} book1.return \land book2.return \\
transferAuthors1 \mathrel{\widehat{=}} book1.authorList \parallel book2.setAuthorList \\
transferAuthors2 \mathrel{\widehat{=}} book1.authorList \parallel_{!} book2.setAuthorList \\
lendAny \mathrel{\widehat{=}} book1.lend \mathbin{[\!]} book2.lend \\
returnAny \mathrel{\widehat{=}} book1.return \mathbin{[\!]} book2.return \\
lendAndReaderList1 \mathrel{\widehat{=}} book1.lend \fatsemi book1.readerList \\
reviewAndThenReturnBook2 \mathrel{\widehat{=}} book2.review \fatsemi book2.return \\
lendAnyAlt \mathrel{\widehat{=}} [b? : books] \bullet b?.lend
\end{array}
$$

Figure 3.10: Extension of the SmallLibrary class

equally named input or output variables are equated and the conditions defined in the predicate sections of the schemas are conjoined as well.

In Figure 3.10, the two operations *lendBoth* and *returnBoth* make use of the conjunction operator. Both of the two combined operations, *lend* and *return*, have the same input parameter *reader?*. These variables are equated when operation conjunction is performed. If variables shall not be equated the only way to overcome this is to hide or rename one of the parameters.

Parallel Composition: This operator models two operations occurring in parallel. The predicates of the two operation schemas are conjoined and inter-object communication is possible, that is, output variables of one operation and input variables of the other are considered equal if the base names of these variables are the same. All equated variables are then hidden from the environment. This operator is commutative, but not associative.

In the *SmallLibrary* example the operation *transferAuthors1* calls the operations *authorList* and *setAuthorList* of Book. The first has an output variable *authors!* and the latter an equally named input variable *authors?*. The value of *authors!* is used as input parameter to *authors?*, which copies the author of one book to the other.

Associative Parallel Composition: This is essentially the same as parallel composition except for the variable hiding. Equated output variables are not hidden and can therefore be accessed by subsequent operations, which makes this operator associative and commutative.

The operation *transferAuthors2* is the same as *transferAuthors1* from the point of object state changes, but the output parameter *authors!* is not hidden from the environment.

Nondeterministic Choice: This operator is used to model angelic choice between two operations. This means, in any case at most one of the combined operations may be performed and which one is chosen depends on which operation is applicable. If both operations are applicable, either can be chosen and it is not further specified which one it is. The operator is commutative and associative and no inter-object communication is available. The argument operations are required to have the same auxiliary variables, i.e. they must be signature compatible.

In Figure 3.10 operations *LendAny* and *ReturnAny* use the nondeterministic choice operation expression. In the first case, both *lend* operations have one input parameter *reader?* and are therefore signature compatible. Lending a book can only be performed if the the book is not yet lent. This means that only the book that is currently not lent will be chosen. If both books are available, one of the two books is chosen arbitrarily.

Sequential Composition: This operation is the only one that defines an execution order for the two combined operations. Inter-object communication is possible from the first to the second operation, that is, output variables of the first can be equated with input variables of the second operation. The intended semantics of the predicates is that first the state transitions of the first operation are applied and then the transitions of the second operation. The sequential composition is only applicable if the preconditions of the first operation are satisfied in the beginning and the intermediate state reached after the state changes from the first transition, satisfies the preconditions of the second operation. The operation is considered atomic, that is it can only be applied as a whole or not at all.

The operation *lend* has one input parameter *reader?* and the operation *readerList* in Figure 3.9 has one single output parameter *readers!*, in which the names of the readers up to that time are returned. So the composed operation *lendAndReaderList1* needs one input parameter *reader?* as an input for operation *lend* and provides one output parameter *readers!* to the environment. If the whole operation is applicable, first *lend* is performed, which leads to a state change and immediately afterwards *readerList* is called.

The operation *reviewAndReturnBook2* is made up of the operations *book2.review* and *book2.return*. Therefore the composite operation needs two input variables *recension?* and *reader?*, but delivers no output variables. The predicates of the two operations are conjoined so that first the new recension is added as a review and then the book is returned.

Scope Enrichment Operator: has the following structure:

$$ScopeEnrichOp \mathrel{\widehat=} [Declarations \mid Predicates] \bullet OperationExpression$$

It declares some auxiliary variables and may restrict them then by *Predicates*. The scope enrichment operator enables all the declared variable of the first argument operation to be accessible in the second argument operation. There, the variables are used in the *OperationExpression*. The declared variables are comparable to the declaration of auxiliary variables in operation schemas, which are available in the local scope of the operation schema. In the case of the scope enrichment, these variables enrich the scope or environment to the right of the $\bullet$. The [*Declarations* | *Predicate*]-part may also be substituted by any other kind of *OperationExpression*. In this case the scope of all the declared variables is also extended to the right hand side.

In operation *lendAnyAlt* in Figure 3.10, the input variable *b?* is set to one of the two books and then applies the promotion of operation *lend* to this input parameter.

3.10.4 Distributed operators

For some of the operation operators described in the last section, there is also a more general form, the so called *distributed operators*. The following three kinds of operators are provided by Object-Z for this kind of operation:

- Distributed Conjunction $\bigwedge$
- Distributed Nondeterminitistic Choice $[\!]$
- Distributed Sequential Composition $\fatsemi$

As one can see in Figure 3.11, the syntax is essentially the same for the three of them. *DOp* has to be replaced by the corresponding operator symbol.

```
__ SomeClass ______________________________________
| DOp Declaration[ | Predicate] • OperationExpression
|__________________________________________________
```

Figure 3.11: Distributed operators in Object-Z

The declaration defines auxiliary variables, that can be restricted by the optional *Predicate*. The *OperationExpression* is then performed for each object defined by the *Declaration* part. The behavior depends on the chosen operator:

distributed conjunction is similar to the normal conjunction described earlier except that there are more argument operators

distributed nondeterministic choice chooses one of the operations from an arbitrary number of options

SomeClass

$Op3 \mathrel{\widehat{=}} \fatsemi\ b : s \mid b.value < 10 \bullet b.Op2$

Figure 3.12: Example of a distributed sequential composition in Object-Z

distributed sequential composition executes the resulting operation expressions in a sequence. In contrast to the binary operation, the sequential order is not predefined for the distributed version of this operation. The operands may appear in any order that is applicable.

Figure 3.12 shows an example for distributed sequential composition. In the declaration part it selects a subset of set *s* consisting of only those objects *b* with feature *value* smaller than 10. If only two objects *x* and *y* satisfy this condition, Op3 would be equivalent to:

$$Op3 \mathrel{\widehat{=}} (x.Op2 \fatsemi y.Op2) \mathbin{[\!]} (y.Op2 \fatsemi x.Op2)$$

As the distributed sequential operator only says that it is applicable if at least one ordering of the variables can be found, all options must be taken into consideration. If the declarations and predicate part results in a set with three members *x*, *y*, *z*, the Op3 would be equivalent to:

$$\begin{aligned} Op3 \mathrel{\widehat{=}}\ & (x.Op2 \fatsemi y.Op2 \fatsemi z.Op2) \mathbin{[\!]} (x.Op2 \fatsemi z.Op2 \fatsemi y.Op2) \mathbin{[\!]} \\ & (y.Op2 \fatsemi x.Op2 \fatsemi z.Op2) \mathbin{[\!]} (y.Op2 \fatsemi z.Op2 \fatsemi x.Op2) \mathbin{[\!]} \\ & (z.Op2 \fatsemi x.Op2 \fatsemi y.Op2) \mathbin{[\!]} (z.Op2 \fatsemi y.Op2 \fatsemi x.Op2) \end{aligned}$$

Apparently, the complexity is rising with the number of elements that satisfy the conditions in the declaration and predicate section.

This concludes the chapter introducing the Object-Z language. Next, Perfect, the second specification language presented in this work, is described in detail.

CHAPTER 4

Perfect

This section provides an introduction to the main Perfect language constructs and is based on the language information about Perfect available in the online tutorials of the Escher website[1], the introduction to Perfect by Carter and Monahan [CM05], and the Perfect Developer language reference manual [Esc].

4.1 Symbols

Some of the most basic symbols of the Perfect specification are given in Table 4.1, so that more complex constructs can be easier understood.

Table 4.1: Symbols in Perfect

Symbol	**Description**
`//`	Starts a one-line comment terminated at the end of the line
`;`	Declaration and statement separator, statements are executed sequentially.
`,`	Statement separator, statements are executed in parallel.
`self`	References the current object
`!`	Indicates a change in the value of a variable, parameter or `self`
`'`	The prime refers to the final value of a variable.
`?`	Can be used to state that this part of the specification is not yet decided

There are three kinds of brackets in Perfect to group expressions:

[1]`http://www.eschertech.com/tutorial/tutorials_overview.php`

Round Brackets embrace parameter lists or define the evaluation order in expressions.

Square Brackets enclose the guards of the alternatives in conditional expressions.

Curly Brackets are only used to enclose the parameter list of a constructor call.

4.2 Predefined Types and Class Library

Perfect provides various predefined classes in its class library. Some of the classes that will be necessary for the further work, are named and described here, while a comprehensive list can be found in the Perfect language reference manual [Esc, Appendix A].

bool Represents a boolean object, that may have one of the two truth values (`true` or `false`). The logical operators and (`&`), or (`|`), implication (==>, <==), equivalence (<==>) and negation (`~`) are available for this type.

char: Represents a single character literal. A predecessor (<) and successor (>) operation are available as well as various functions to determine whether the character is a letter, digit, or a control character.

string: Represents a sequence of characters. For strings all sequence operations are available.

int: Represents integer numbers. The class provides the well-known binary arithmetic operators addition (+), subtraction (-), multiplication (`*`), integer division (/), remainder (`%`), and exponentiation (`^`). Additionally the unary predecessor, successor, and negation operators are available.

nat: Represents special kinds of integers, i.e. integers with non-negative value. All the operations available for type int are also available for type nat.

real: Represents rational and irrational numbers. Except for the remainder and integer division operators, the same operators are available for real numbers. Additionally, the division of real numbers is available as well as some functions to round the real value to integer values.

set: Represents a collection of objects in which each item may only ever occur once.

seq: Represents an ordered collection of objects. The same item may occur arbitrarily often in a sequence.

bag: Represents a collection of values with no order of items and no restriction on the number of occurrences of one item.

pair: Represents a combination of two ordered items x and y. The items can be retrieved from the pair `p` by accessing `p.x` or `p.y` respectively.

triple: Represents a combination of three ordered items x, y, and z. The items can be retrieved from the triple `t` by accessing `t.x`, `t.y`, or `t.z` respectively.

map of (X → Y): Represents a mapping of items of type X to items of type Y. Various methods exist to access domain, range, or single elements, add or remove mappings.

void: Has only one valid value: `null`.

Various methods are available for collection types, set, sequence, and bag, that provide for collection concatenation (++), intersection (⋆⋆), difference (−), disjunction (##), or counting the number of elements in the collection (#).

4.2.1 Further Operators

Additionally to the operators described within the list above some comparison operators applicable for various types exist. These operators are listed and described in Table 4.2.

Table 4.2: Comparison operators in Perfect

Symbol	**Description**
˜˜	The rank operator yields one of the values `rank below`, `rank same`, `rank below`. It is predefined for enumeration classes in which the declaration order defines the order of the enumeration elements.
=	Equality is defined automatically by the system, but the user may provide a refinement. Whether the two operands are equal depends on the values of all abstract data members.
< <= > >=	The greater/smaller than (or equal) operators are predefined for all classes representing numbers. The system defines them automatically according to the definition of the rank operator, e.g. `a < b <==> a ˜˜ b = rank below`
`in`	The inclusion operator checks whether an element is in a collection
««= =»»	These inclusion and reverse inclusion operators compare two collections whether one is a subset or strict subset of the other set.
`like`	Checks if the operands have exactly the same types at runtime.
`within`	Checks if a value is a member of one of the types forming a united type.

Last, but not least, the sequence creation operator ".." shall be mentioned which is defined for integers and characters. It yields a sequence of elements starting with the element on the left side ranging up to the element on the right side. If the element on the right is smaller, then the resulting sequence contains elements in decreasing order.

4.3 Declarations

To declare a new variable the following statements are used in Perfect:

`a: t`	Declares one variable *a* with type *t*
`a, b, c: t`	Declares three variables *a, b* and *c* all with type *t*

Bound variable declarations may also declare variables as members of collections in quantified and transformation expressions.

`a::collection`	Declares variable *a* to belong to *collection*

4.4 Expressions

This section describes a selected set of expressions in Perfect that are necessary to be understood for discussion of the transformation rules in Chapter 5.

4.4.1 Quantified Expressions

Perfect provides universal and existential quantification expressions over types and collections. The syntax of this construct is given as follows:

```
(forall|exists) declaration :- predicate
```

The *declaration* part introduces some variables and for each of these variables it is checked whether it satisfies the condition in *predicate.* The whole expression yields true if all variables or at least one satisfies the predicate for the universal and the existential quantification, respectively.

4.4.2 Choosing Elements Expressions

Elements of collections can be filtered so that exactly one, several or any of several elements of the collection are chosen. Such constructs are built in the following way.

```
(that|any|those) declaration :- predicate
```

The *declaration* part may only declare one *identifier* in the way described in section 4.3. The values referred to by the identifier are then filtered by the *predicate* resulting in one or more elements. The meaning of `any` is to take any of these resulting values. For `that` must be asserted that the result contains exactly one element or all elements are equal. For these two options it is also possible to leave out the predicate, which is equal to stating `true` as the predicate. In the third option `those` returns all the matching elements.

4.4.3 Transformation Expressions

Transformation expressions can be applied to sequences, sets or bags to transform the elements of these collections. This language construct comes in two syntactical variations:

```
for identifier::collection yield expression
for those identifier::collection :- predicate yield expression
```

The first expression takes all elements of *collection*, referred to as *identifier*, and calculates for each element *expression*. The second takes only those elements of collection that also satisfy the predicate and then transforms these using *expression*. Both variants keep the collection type, which means that for sets all duplicates of transformed elements will be removed from the result set and the order of elements remains the same as for sequences. In contrast, the element type depends on the result type of *expression*.

4.4.4 Conditional Expressions

A conditional expression consists of two or more guarded expressions enclosed in parentheses.

```
([Guard1] : Expression1,
 [Guard2] : Expression2,
 [] : ElseExpression)
```

The guards are evaluated in the order stated and if a guard evaluates to true, the expression on the right side is evaluated and the conditional expression is exited. The remaining guards are not evaluated. An empty guard, which is equivalent to a guard containing *true*, can be added as shown in the last line to provide a branch to be executed if all the others evaluate to false.

If a guarded expression is preceded by the keyword `opaque`, this makes the conditional expression nondeterministic. In this case no default guard may be specified, but on the other hand all guards are evaluated and in case there is more than one that yields true, one is chosen arbitrarily.

4.5 United Types

To state that a function may return objects of some arbitrary class, but also the value `null`, united types are useful.

```
a: string || void
```

The expression above states that variable *a* may be either *string* or *void*, i.e. have the value *null*. This construct is not restricted to combinations with type *void*, and may be used with any pre- or userdefined type.

4.6 Type Conversion

Perfect provides two options to convert types, the operators `is` and `as`. The first operator narrows the type of a variable whereas the second is the converse and widens the type.

```
a is string
42 as int || string
```

This first expression is only valid if *a* has run-time type *string*. In the other example the automatic type *integer* of 42 is expanded to types *integer* or *string*.

Except for this explicit type widening, Perfect also allows for automatic type widening in some cases as described in part three of the Perfect basic tutorial[2] and [Esc, section 5.4.14].

4.7 Enumeration Classes

To define a class that has only a finite number of valid values, enumeration classes are the best way to express that in Perfect. If one wants to define a type *Color* that may take some particular values, this can be written as:

```
class Color ^= enum white, yellow, orange, red, purple, blue, green end;
```

Variables of types defined in that way can be declared in the same way as described in section 4.3. The following example shows how to declare a variable of type *Color* and how to refer to the values of the enumeration type to initialize variable *colorOfCar*.

```
colorOfCar: Color;
colorOfCar = Color red;
```

4.8 Declaring Subtypes

Sometimes only a restricted number of values of an already defined type may be needed for some particular variable. For example, the scores of an examination may range between 0 and 100. A subtype of the integer numbers is advantageous for this purpose. Perfect provides means for declaring subtypes using a `those`-clause similar to that described in section 4.4.2.

```
class ExamScore ^= those x: int :- 0 <= x <= 100;
```

[2]http://www.eschertech.com/tutorial/tutorials_overview.php

This creates the new type *ExamScore* and it may be used as such to declare variables. Such type declarations may introduce new types deduced from any kind of pre- or user defined type. The new type provides all the functionality of the original type, but only for a restricted subset of values of the original type.

4.9 Abstract Classes

In Perfect the most complex types may be introduced by so-called *abstract classes*. Listing 4.1 gives an overview of the structure of such a class in Perfect.

```
class ClassName of FormalParameters ^=
    inherits SuperClassName
    abstract abstractMembers
    internal internalMembers
    confined confinedMembers
    interface interfaceMembers
end;
```

Listing 4.1: Abstract classes in Perfect

In the first line the class name is declared followed optionally by formal parameters used for template classes. These formal parameters may also be followed by expressions defining restrictions on the actual classes with which these formal parameters are instantiated. The next part of the declaration is the optional *inherits*-clause which defines at most one class as the direct superclass of this class.

The remaining part of the class specification is also referred to as the *class body* and specifies the behavior of the class, its possible state transitions and may even provide refinements and implementation details to improve performance while profiting from the validation and verification mechanisms available in Perfect Developer. Constant, variable, constructor, operator, function, selector, schema, property, axiom, and even inner class declarations count amongst the *member declarations* possible in the four different sections of the class body.

Abstract and internal sections are the private part of a class declaration and therefore invisible for other classes. In contrast, the confined and interface sections are the public part of the class declaration that may be visible and accessible for some or all other classes. The order of the sections is mandatory as given in the class example in Listing 4.1. However, a class can only specify one of abstract, confined, or interface part, the other parts are optional then.

Abstract section: The abstract section counts among the two sections appearing in most of the Perfect specifications. As described earlier this part of the specification is private, which means that no other class can directly see or access the members declared in this part. Except for member declarations, the abstract section is also the place for defining class invariants that express relationships between abstract data members. Abstract data members together with class invariants form the model of a class.

The postconditions of all constructors and modifying operations of the class have to respect these relationships. Therefore, Perfect implicitly includes the invariants in all these postconditions, which means that the whole postcondition, i.e. the manually defined postconditions plus the implicit ones, yields false if the first part contradicts the implicitly added invariants.

Internal section: This part describes actual implementations of abstract data members, for example to represent data structures in a more efficient way. The internal section may consist of the same building blocks as abstract sections including class declarations and invariants, furthermore, implementations may be specified in the internal section.

Confined section: This part contains declarations of those class members that are intended to be visible for subclasses. Confined members are only directly accessible within the defining class and all its derived classes. This section may contain functions, selectors, properties, axioms, constructors, or redeclarations of abstract variables or constants.

Interface section: Whereas the confined section is only visible and accessible for a restricted group of classes, member declarations of the interface section are accessible from any other object without restrictions. Interfaces may declare the same types of members that confined sections do. The constructor provided for object initialization is usually provided in this part of the class body. Interfaces are the second kind of Perfect class body sections that are present in almost all Perfect specifications.

4.9.1 Class Member Declarations

This section gives an overview of what kinds of class members can be specified in the four sections of the class body. Each syntactical element will be illustrated by a short example, which will be part of the specification of a class representing currency amounts.

Constant Declarations

A constant in Perfect declares an identifier that gets a value on initialization and will never change afterwards. The second characteristic of constants is that they have the same value for all objects of the same class in which they are defined. Listing 4.2 shows how to declare an identifier to hold a constant value of 99.

```
class CurrencyAmount ^=
abstract
   const CENT_MAXIMUM ^= 99;
interface
   ?
end;
```

Listing 4.2: Constants in Perfect

This is the most commonly used form of constant declarations. The syntax of Perfect also allows for more complex structures in constant declarations, but as these will not be relevant in the further course of this work, they are not discussed here. More details on constant declarations are available in [Esc, chapter 6.2].

Variable Declarations

In Perfect a member variable declaration is started with the keyword `var` followed by a declaration as defined in section 4.3 that specifies name and type of the variable.

```
class CurrencyAmount ^=
abstract
   const CENT_MAXIMUM ^= 99;
   var   amount, cents: nat,
         currency     : string;
interface
   ?
end;
```

Listing 4.3: Variable declaration example

The variable declaration in Listing 4.3 declares three variables. The type of a variable can be either a class name, a united type or a subclass of another type introduced by a `those`-clause.

Invariants

In the presented example the cent values shall be constrained by an upper and lower bound, which is expressed by an invariant in the abstract section.

```
invariant 0 <= cents <= CENT\_MAXIMUM;
```

After adding this invariant, it is no longer possible to set cents to any arbitrary value. Invariants always start with the keyword `invariant` and are followed by a predicate stating relationships between class members. An arbitrary number of invariants can be provided in the abstract section of a Perfect specification.

Constructors

Constructors provide the means to create new objects of a class and at the same time initialize the abstract data members of the object. Constructors always have to set each abstract data member to some value while keeping all invariants satisfied.

The simplest form of constructor is written as follows.

```
build{};
```

This can only be used in a context where no abstract data members have to be initialized. As soon as there are abstract data members, the easiest way of initializing an object is providing a parameter list containing each abstract member name preceded by "!" together with its type. The *CurrencyAmount* class provides such a constructor as shown in Listing 4.4.

```
class CurrencyAmount ^=
    ...
interface
    build{!amount:nat, !cents:nat, !currency:string};
end;
```

Listing 4.4: Constructor example

This initializes the abstract data members to the values provided in the parameter list, where the mapping is performed by means of the names. If one tries to verify this specification in Perfect Developer, a conflict for the initialization of variable *cents* arises. If a value of 100 or higher is provided as parameter to the constructor for this variable, the invariant is no longer satisfied. To overcome this situation a precondition is added to the constructor saying essentially the same as the invariant. However, the precondition may even be more restrictive. The code snipped in Listing 4.5 shows the adoptions that have to be made in the context of this class.

```
class CurrencyAmount ^=
    ...
interface
   build{!amount:nat, !cents:nat, !currency:string}
      pre 0 <= cents <= CENT_MAXIMUM;

   build{!amount:nat, !cents:nat}
      pre 0 <= cents <= CENT_MAXIMUM
      post currency != "EUR";

   build{centInput:nat}
      post ([centInput > CENT_MAXIMUM]:
                amount != centInput / (CENT_MAXIMUM + 1),
                cents  != centInput % (CENT_MAXIMUM + 1),
            []: amount! = 0, cents! = centInput),
                currency != "EUR";
end;
```

Listing 4.5: Constructors with precondition example

In other cases it may be necessary to modify the input parameters or even set a variable to a predefined value without the need of input parameters, then a postcondition is added to the constructor. In the *CurrencyAmount* example it is convenient to provide a constructor with a default value for the currency (i.e. "EUR" in the second constructor of Listing 4.5).

A second form may be to provide the whole amount in cents as an input parameter. This requirement is fulfilled by the third constructor in Listing 4.5, which differentiates between the cases of cents below and including or over `CENT_MAXIMUM`. In the first case the values can be set directly, whereas some calculations are performed for the latter case.

Functions

Functions in Perfect assure to have no side-effects. They do not change the state of an object, i.e. they do not modify values of member variables, but they always return at least one value. Therefore, the function header comes in two forms. Simple functions with only one return value can be written as:

```
function fName (parameterList):returnType
```

Functions with one or several return values have the following notation:

```
function fName (parameterList)retVal1:rType1, rVal2:rType2, rVal3:rType3
```

In both cases, the *parameterList* may be empty, so the whole expression, including parentheses, is omitted. Directly following the function header, a precondition for the function may be specified.

```
pre PredicateList
```

If a precondition is given, the function may only be executed, if the constraints are met on function call time. Perfect Developer creates a verification condition to check that the preconditions are satisfied for each function call.

The function body can also be specified in two different ways.

```
^= Expression
satisfy PredicateList
```

In the first case exactly one expression for each return value is given in the order of appearance. In the *satisfy*-variant the terms `result` or `result.returnValue` reference the return parameters and may be used more than once in the predicates of the *PredicateList*. Redefining abstract member variables as functions in the *interface* or *confined section* provides read-access to them. A temporary variable is declared to save the container object for all parameters of a function. The single return values can be accessed by referring to their names.

Listing 4.6 illustrates the different variants just described. Function *convert2CA* provides one return value with the expression postcondition variant whereas *convert2AmountNCents* also scales the CurrencyAmount but provides amount and cents separately using the satisfy-form. The last function *showRetvalAccess* introduces a temporary variable to show the access of the return values of another function.

```
class CurrencyAmount ^=
interface
   function convert2CA(factor: nat):CurrencyAmount
      pre factor > 0
      ^= CurrencyAmount{((amount * 100 + cents) * factor)};

   function convert2AmountNCents(factor: nat) amountRet: nat, centsRet: nat
      pre factor > 0
      satisfy ([cents * factor <= CENT_MAXIMUM]:
                  result.amountRet = amount * factor &
                  result.centsRet  = cents * factor,
               []:result.amountRet = amount * factor + (cents * factor) / 100 &
                  result.centsRet  = (cents * factor) % 100);

   function showRetvalAccess(factor: nat):bool
   ^= (let retVal ^= convert2AmountNCents(factor);
         (retVal.amountRet > 20 & retVal.centsRet > 20));
end;
```

Listing 4.6: Single- and multi-value return functions example

Selectors

To provide full read and write access to an abstract member variable or constant, this member may be redeclared as selector in the confined or interface section. In this case any other object of any class may read the value of the member and even arbitrarily modify its value. As this would imply that invariants could not provably hold, it is not allowed to express invariants with respect to members declared as selectors.

```
class CurrencyAmount ^=
interface
    selector currency;
end;
```

Listing 4.7: Selector example

In the *CurrencyAmount* example a requirement might be to provide full read and write access for the currency string. So this variable is redeclared as selector in the interface section. Assuming another class has a member variable `currencyAmount` of type `CurrencyAmount`, then this class may refer to the `currency` member by the dot notation, `currencyAmount.currency` for both reading from and writing to the member variable.

Operators

In many cases it is convenient to use well known operators in user defined types instead of function names to express a similar meaning as the predefined operators. Perfect allows to provide userdefined implementations for several unary and binary operators [Esc, section 6.6]. For convenience an operator for addition of amounts shall be specified for class *CurrencyAmount*, as well. Listing 4.8 specifies the +-operator to add *CurrencyAmounts* conveniently. This operator also takes care that the invariant for variable *cents* remains satisfied for the newly created *CurrencyAmount* object by distinguishing two cases in the operator body. The first guard evaluates to true whenever the cents of the two operands yield a value of at least 100. In this case the sum of the amounts must be increased by one and the sum of the cents must be decreased by 100. Alternatively, the cent and amount values of the operands simply have to be summed.

```
class CurrencyAmount ^=
interface
   operator +(other: CurrencyAmount): CurrencyAmount
      pre other.currency = currency
      ^= ([cents + other.cents >= 100]:
             CurrencyAmount{amount + other.amount + 1, cents + other.cents - 100},
          []:CurrencyAmount{amount + other.amount, cents + other.cents});
end;
```

Listing 4.8: Operator for CurrencyAmount

Schemas

Whereas functions and operators described so far may only return new objects of the same class, schemas also allow for changes of the values of member variables. Listing 4.9 shows the syntax of a schema.

```
schema !schemaName (parameterList)
    pre preconditionPredicates
    post postcondition;
```

Listing 4.9: Syntax of schemas in Perfect

The *schemaName* may be prefixed by an exclamation mark (`!`), depending on whether the schema allows changes of member variables or not. Following the schema name, an optional list of parameter name and type pairs may be given enclosed in parentheses. While functions in Perfect may only use input parameters, Perfect allows four kinds of parameters for schemas as listed here:

Input Parameters are declared as a pair of name and type, but without any kind of decoration or keyword identifier. They may only be read by the schema.

Output Parameters are postfixed by an exclamation mark (`!`) and the type is preceded by the keyword `out`. The initial values of such parameters are irrelevant, but final values have to be given for these parameters

Limited Parameters are postfixed by an exclamation mark (`!`) and the type is preceded by the keyword `limited`. According to [Esc, chapter 6.8.1] these parameters are only limited writable, that is, they may not be reassigned. Consequently, for limited variables of union type, the actual type cannot be modified by the schema.

Repeated Parameters are all parameters declared after the keyword `repeated`. This means that several parameters of one type may be given and the possible number of these parameters is not predefined.

If the schema is only applicable under certain circumstances, these conditions may be specified in the precondition predicates given after the keyword `pre`. The postcondition part provides the details on the actual state transitions. Such postconditions can be expressed in various forms and provide different syntactical elements, that are described in the next section.

4.9.2 Postconditions

The most general postcondition is defined by the *change ... satisfy*-form. Following the keyword `change` all the variables have to be listed that will be modified by this schema. The postcondition predicates after the keyword `satisfy` state the relationships of variables before and after applying the schema by expressions using constants, parameters, and state variables. In the predicate all the variables in the change list may be referred to as primed variables. Although this form provides for literally all kinds of postconditions, this structure also has some shortcomings as it is not always possible to automatically deduce an implementation.

The *assignment postcondition* variation assigns an expression on the right hand side to a variable on the left hand side. Although this approach is often an easy and straight forward way of specifying state changes, this syntax is not applicable universally, but only in those cases, where the new value can be explicitly written on the right side.

There is also a shorthand form which involves binary operators. A postcondition that wants to double some value, add it to another variable and then assign it as new value to this second variable could either be stated as follows:

```
post x!=x+2*y
post x!+2*y
```

```
schema !convertAmountChangeSatisfy(factor: int)
    pre factor > 0
    post change amount, cents
         satisfy amount' = amount * factor + (cents * factor) / 100 ,
                 cents'  = cents * factor % 100

schema !convertAmountAssign(factor: int)
    pre factor > 0
    post amount! = amount * factor + (cents * factor) / 100 &
         cents!  = cents * factor % 100

schema !convertAmountThen(factor: int)
    pre factor > 0
    post cents!  = cents * factor then
         amount! = amount * factor + cents / 100 then
         cents!  = cents % 100;

schema !convertAmountAnd(factor: int)
    pre factor > 0
    post amount! = amount * factor + (cents * factor) / 100 &
         cents!  = cents * factor % 100;
```

Listing 4.10: Different kinds of postconditions in Perfect

The *then-postcondition* may be used to express a sequential order on postcondition predicates. Primed subexpressions refer to the immediate after state, i.e. primed variables in the left subpredicate refer to the intermediate state between applying the left and the right subpredicate and unprimed variables in the right subpredicate may refer to variables changed by the left subpredicate.

The *&-postcondition* may be used whenever parallel modifications to variables shall be performed. As a consequence this form can only be applied if the sets of variables to be modified of each individual postcondition are disjoint. Listing 4.10 shows examples for the four schemas making use of all the four postcondition types described above. Each schema provides exactly the same functionality, but expresses the result using different syntactical elements. Listing 4.11 presents the usage of output variables in schemas in conjunction with the application of the "&"- and "then" forms of postconditions.

```
schema !convertAmountAndReturn(factor:int, newAmount! : out CurrencyAmount)
   pre factor > 0
   post (cents!  = cents * factor % 100 &
         amount! = amount * factor + (cents * factor) / 100) then
         newAmount! = CurrencyAmount{amount, cents};
```

Listing 4.11: Output parameters in combined "then" - and "&"-Postcondition

The *bracketed postcondition* given enclosed in parentheses allows introducing temporary variables or constants. Additionally, guards as described in the conditional expressions section 4.4.4 may provide different postconditions if certain boolean expressions specified within the guard evaluate to true. An example is given in Listing 4.12.

```
schema !convertAmout2CentNAmountNReturn(factor: int,newAmount!:out nat,newCents!:out
    nat)
    pre factor > 0
    post (let centsTimesFactor ^= cents * factor;
            [centsTimesFactor <= CENT_MAXIMUM]: newAmount! = amount * factor &
                newCents!  = cents * factor,
            []: newAmount! = amount * factor + (cents * factor) / 100 &
                newCents!  = (cents * factor) % 100);
```

Listing 4.12: The bracketed postcondition

If all elements of a collection shall be modified in one step, *forall-postconditions* may be helpful. The syntax is similar to the bound variable expressions described in section 4.4.1 except that the predicate of the bound variable expression is replaced by a postcondition predicate. A valid example of this kind of postcondition is given in Listing 4.13. The prefix keyword `nonmember` makes function *toLowerCase* accessible on class, not object level.

```
nonmember function toLowerCase(aString: string): string
    ^= (for c::aString yield c.toLowerCase);

schema !convertCurrenciesToLowerCase
    post (forall i::0..<#currencies :- currencies[i]!=toLowerCase(currencies[i]));
```

Listing 4.13: Valid "forall" postcondition

```
schema !convertCurrenciesToLowerCase
    post (var lowerCaseCurrencies: string != "";
          forall i::0..<#currencies :- (currencies[i]! = toLowerCase(currencies[i]),
                lowerCaseCurrencies! = lowerCaseCurrencies ++ currencies[i] ++ ","));
```

Listing 4.14: Invalid "forall" postcondition

As the modifications described by this postcondition type are considered to happen in parallel, the changed objects must be distinct and constructs such as the one presented in Listing 4.14 are not possible. The variable *lowerCaseCurrencies* aims on containing all the currencies in lower case concatenated to one string where each currency is separated from the next by a space. However, as *lowerCaseCurrencies* does not depend on the bound variable *i*, this means that *lowerCaseCurrencies* would have to be modified by several postconditions in parallel. This is a situation that cannot be satisfied.

4.9.3 Calling Schemas

A modifying schema is called using a similar notation as in function calls, but the dot between variable name and function name is replaced by an exclamation mark indicating that the object is modified by the schema.

```
schema !convertAmount(factor : int)
    pre factor > 0
    post currencyAmount!convertAmountAssign(factor);
```

Listing 4.15: Schema calls in Perfect

To conclude this section about class member declarations, a remark for constants, functions, selectors, operators, and schemas is added. All these kinds of members may be declared preceded by the keywords `opaque` or `ghost`. The first keyword always introduces a nondeterministic context that has to be propagated upwards. The latter defines that no code has to be produced for these members. Ghost members are usually helper functions used in the context of invariants, assertions, or preconditions.

4.9.4 Preconditions

Preconditions as already used for operators, functions, selectors, constructors, and schemas define the criteria in which cases such a member may be applicable or not. A precondition is a predicate that may refer to any member variable, constant or input parameter and expresses the relationships that have to be true before such a method may be used. It specifies a part of the contract that the client, i.e. the method caller, has to ensure so that the method can be used. On the other hand it defines the contract for the server, that it delivers the results specified in the postcondition whenever the precondition is satisfied.

4.9.5 Postassertions

The other part of the contract that is concluded between a method and its caller is specified in a so-called *postassertion*. Postassertions define the conditions which are guaranteed to hold after the method call whenever the method is called in a state and with parameter values such that the precondition is satisfied. Postassertions can be given for operators, functions, selectors, constructors, and schemas, and describe the return value or final value of a modified object. In the case of schemas the postcondition has to imply the postassertion, i.e. the conditions defined by the postcondition also imply that the conditions given in the postassertion hold.

```
schema !convertAmount(factor: int)
    pre factor > 0
    post cents!  = cents * factor then
         amount! = amount * factor + cents / 100 then
         cents!  = cents % 100
    assert amount' = amount * factor + (cents * factor) / 100,
           cents'  = cents * factor % 100,
           (factor > 1) ==> amount' >= amount;

function convertAndCutToAmountByFactor(factor:int) amountRet: nat, centsRet: nat
    pre factor > 0
    satisfy (let retVal ^= convertAmoutToCentAndAmount(factor);
                (result.amountRet = retVal.amountRet & result.centsRet  = 0))
    assert result.amountRet >= amount * factor,
           result.centsRet = 0,
           (cents * factor <= CENT_MAXIMUM) ==> result.amountRet = amount * factor,
           (cents * factor > CENT_MAXIMUM)  ==> result.amountRet > amount * factor;
```

Listing 4.16: Postassertion examples

The assertion in the schema example in Listing 4.16 states very much the same as the postcondition, but expresses the relationship between amount and cents and their final states by using only one predicate for each variable. Additionally, the last condition asserts that if the parameter factor is bigger than one, then the final value of amount must be at least as high as the initial value of amount.

As function *convertAndCutToAmountByFactor* shows, assertions may also be used in function definitions. In this case the assertion says something about the return value or object *result* and the function body has to deliver return values that satisfy the postassertions.

4.9.6 Properties

Properties are class member declarations, but are at the same time a special kind of assertion that may also involve more than one operator, function, or schema. That is why they have not been described in previous sections. In the examples in the previous sections some functions and schemas have been introduced that essentially calculate the same values. To allow Perfect to create checks on whether these equalities hold, one may specify properties that express these relationships.

The difference between functions *convert2AmountNCents* and *convertAndCutToAmount-ByFactor* is that the latter cuts the cents, but the amount must be the same if the input factor is the same. The property declaration shown in Listing 4.17 states the relationship between the two functions and at the same time provides this information to the verification engines, which can help in proving verification conditions.

```
property (factor: int)
    pre factor > 0
    assert convert2AmountNCents(factor).amountRet =
                convertAndCutToAmountByFactor(factor).amountRet,
           convert2AmountNCents(factor).centsRet >=
                convertAndCutToAmountByFactor(factor).centsRet;
```

Listing 4.17: Properties in Perfect

4.9.7 Axioms

Sometimes the verification engine of Perfect is unable to prove certain verification conditions. If the specifier knows about predicates that can be considered as true, an axiom can be provided stating these constraints. The verifier will then assume that this axiom is true and can therefore include this assumption in its verification process.

```
function convertAmountWithReal(factor: real): CurrencyAmount
    pre factor > 0.0
    ^= CurrencyAmount{((amount * 100 + cents) * factor).rounddn};

axiom (realValue:real)
    pre realValue >= 0.0
    assert realValue.rounddn >= 0;
```

Listing 4.18: Axioms in Perfect

The constructor of class `CurrencyAmount` needs a non-negative integer parameter value, but the rounding functions provided by class real like *rounddn* do not assert the return value to be positive even if strictly positive numbers are handed over as parameter. Providing an axiom stating such knowledge not yet known may help the verification engine in the verification process. The axiom in Listing 4.18 states that if function *rounddn* is applied to a non-negative real number, then the return value is also non-negative and can be considered as a fact by the Perfect verifier.

4.9.8 More on Genericity

When using template classes, sometimes certain operators or functions need to be available to properly specify the generic functionality. Perfect provides the rank operator and a ghost equality by default for template classes. To complement these, requirements on the formal parameters, such as necessary operators, functions, or schemas available for a template class or restrictions on the class hierarchy, may be specified in an optional part. Functions or operators specified in the `require`-part of a template declaration can be assumed to exist and may be used for members of generic class. Additionally, a verification condition is created for each reference to an object of the template class which instantiates parameter X.

Assuming that there is a hierarchy with *BankCard* as superclass and *DebitCard* and *CreditCard* as subclasses, the example in Listing 4.19 declares a template class *BankCards*, which may hold either a set of BankCard, DebitCard, or CreditCard objects.

```
class BankCards of X
    require X within BankCard
    ^= set of X;
```

Listing 4.19: Restrictions on templates in Perfect

In the second example in Listing 4.20, a template *Queue* is introduced, which may hold elements of any class that defines a function *isSpecialItem* without parameters.

```
class Queue of X
    require X has function isSpecialItem
    ^= ClassBody
end;
```

Listing 4.20: More restrictions on templates in Perfect

4.9.9 More on Inheritance

All confined and interface members are inherited to subclasses and can be accessed by new or overriding members. In the class body members that have already been declared may be `redefined`, but must have the same signature as the overridden member, i.e. have the same parameters and deliver the same result type. Additionally, an overriding schema or function always has to be applicable whenever the overridden member is applicable. The results delivered by the function or the postcondition of the schema must satisfy the specification of the overridden member.

It is also possible to leave the specification details for later and declare interface functions, operators, selectors, schemas, and, as a consequence whole classes, as `deferred`. In this case one cannot instantiate objects of this class, but variables can be declared as `from DeferredSuperClass`. Such variables can then reference objects of any class derived from this deferred class. Deferred classes may actually define some of the functions or schemas etc in the class or at least give a postassertion, while leaving some others for specification in subclasses. If a deferred member is overridden the first time, the keyword `define` must be given as a prefix.

The example in Listing 4.21 defines two classes *Shape* and *Polygon*. The deferred class Shape provides the actual specification of schema *translate* including a postassertion that is inherited in all derived classes. The absence of the precondition of this schema in the derived class shows how preconditions can be loosened in overriding methods of subclasses. The function *perim* only provides a postassertion in class Shape, whereas Polygon provides a definition of this method for which the return value has to satisfy the given assertion.

```
class Point ^=
abstract
    var x, y: int;
interface
    selector x, y;
    build{!x: int, !y: int};
end;

deferred class Shape ^=
abstract
    var startingPosition: Point;

interface
    build{!startingPosition: Point};

    selector startingPosition;

    schema !translate(vX: int, vY: int)
        pre vX > 0, vY > 0
        post startingPosition!= Point{startingPosition.x + vX, startingPosition.y + vY}
        assert startingPosition'.x = startingPosition.x + vX,
               startingPosition'.y = startingPosition.y + vY;

    deferred function perim: int
        assert result > 0;
end;

class Polygon ^=
    inherits Shape
abstract
    var nodes: seq of Point;
    invariant #nodes >= 3;

interface
    build{startingPos: Point, !nodes: seq of Point}
        pre #nodes >= 3
        inherits Shape{startingPos};

    redefine schema !translate(vX: int, vY: int)
        pre vX > 0, vY > 0
        post startingPosition!= Point{startingPosition.x + vX, startingPosition.y + vY}
        assert startingPosition'.x = startingPosition.x + vX,
               startingPosition'.y = startingPosition.y + vY;

    define function perim: int
    ^= (+ over(for i::(nodes.front.dom) yield length(nodes[i], nodes[i+1])));

    nonmember function length(p1:Point, p2:Point): int
    ^= p1.x - p2.x + p1.y - p2.y;
end;
```

Listing 4.21: Inheritance in Perfect

This short section on inheritance concludes the overview of the Perfect language. The next chapters will provide the abstract mapping of Object-Z to Perfect and detailed information on the implementation of this mapping.

CHAPTER 5

Mapping Object-Z to Perfect

This chapter presents the solutions and results found in this work, which were also practically realized as tool as described in Chapter 6. The mappings of most Object-Z constructs to equivalent constructs in Perfect and why these are deemed appropriate in this work are shown here. While some transformations could be easily found and the rules are straight forward, some other, more complex Object-Z constructs needed a more thorough analysis to provided a suitable mapping to Perfect. First the conversion for different kinds of expressions is presented, then the mapping rules for each of the building blocks of an Object-Z class specification are described. This work illustrates many of the mappings, especially the more complex ones, using examples given in Object-Z and Perfect. Whenever the semantics of an Object-Z syntax element have not been discussed in the introductory sections, a short description is given here. Considering all the present state of the art and related work, this chapter provides a mapping of all core Object-Z language syntax and semantics and even for some of the more complicated Z expressions. However, some Z constructs are only partially mapped, because more analysis on the context in which these expressions are used. This work mainly focusses on mapping the object-oriented language constructs of Object-Z, not Z expressions.

5.1 Definitions

During the descriptions of the mappings some keywords will be used repeatedly and it is essential that the intended semantics are clear. Therefore some definitions are provided here for the better understanding.

Base name: The base name of a variable is the name of a variable without decorations. Decorations can be ', ! and ?.

Attribute: State variables and constants together form the attributes of a class.

Feature: The features of a class are all attributes and operations of a class and INIT if the initial state schema exists

Type compatibility: Two type definitions are compatible if they define identical sets. [Smi00, p. 46]

Signature compatibility: Two operation definitions have exactly the same input and output variables with regard to name and type.

5.2 Expressions

Most of the expressions described in the following sections are not only syntax elements of the Object-Z language, but also of the Z language. In the Z reference manual [Spi89] Spivey describes the semantics formally and informally in the mathematical toolkit section. Examples for Z/Object-Z expressions can also be found in Appendix A of [DR00].

5.2.1 Sets

In Object-Z there are several ways of defining sets, creating sets out of others or applying functions to them. Table 5.1 lists these different possibilities providing also the names used for these symbols, their usage in Object-Z, and their mappings to Perfect.

Object-Z distinguishes between two kinds of sets regarding the magnitude. Whereas the *power set* is regarded as potentially infinitely large, the *finite set* always has a finite number of elements. In Perfect sets are always considered as finite, only a pre- or user-defined type like natural numbers may have an infinitely large number of elements. This means that mapping both kinds of sets to `set of X` as presented in Table 5.1 does not reflect the potential infiniteness of power sets in Object-Z in Perfect, but is regarded as most accurate because Perfect does not provide a means of expressing infinite sets.

While the mapping of functions like set union, intersection, difference, size, and minimum or maximum does not need detailed explanation as these language constructs can be directly mapped to equivalent functions provided by the Perfect library, the functions at the bottom of Table 5.1 need further comments on why these mappings are appropriate. As observable in Table 5.1 Perfect provides an operator "`..`" denoting a range as well, but the semantics of this operator is slightly different in Perfect. While Object-Z creates a set of integer values if a is less or equal to b by using the range operator, Perfect produces a sequence of integer elements starting from a ranging up to b no matter which of the values is larger. To overcome this difference, it is necessary to distinguish between two cases and apply the function `.ran` to the result of the range-operator in the mapping to Perfect. This results in a set of exactly the same integer values as in Object-Z.

The last two functions *Generalized Union* and *Generalized Intersection* operate on a set of sets of some type X in Object-Z. The first shall produce a set of all the elements appearing in at least one of the subsets of A, the latter stands for the set of all those elements contained in all the subsets of A. To map these generalized operations, the

Table 5.1: Mapping of sets and operations on sets to Perfect

Name	**Usage in Object-Z**	**Mapping to Perfect**
Power Set	$\mathbb{P}\,X$	`s: set of X`
Non-empty Subset	$\mathbb{P}_1\,X$	`s: set of X` `invariant #s > 0`
Finite Set	$\mathbb{F}\,X$	`s: set of X`
Finite Non-empty Set	$\mathbb{F}_1\,X$	`s: set of X` `invariant #s > 0`
Empty Set	$\varnothing$	`set of X {}`
Cartesian Product	$X \times Y$	`pair of (X, Y)`
Set Membership	$\in \; \notin$	`in ~in`
Subset	$\subseteq$	`<<=`
Proper Subset	$\subset$	`<<`
Set Union	$\cup$	`++`
Set Intersection	$\cap$	`**`
Set Difference	$\setminus$	`--`
Set Size	$\#$	`#`
Minimum	$min\ s$	`s.min`
Maximum	$max\ s$	`s.max`
Range	$a \mathrel{..} b$	`([a<=b]:(a..b).ran,` `[]: set of int {})`
Generalized Union	$\bigcup A$	`flatten(A)` or `([A.empty]: set of X {},` `[]: ++ over(A)`
Generalized Intersection	$\bigcap A$	`([A.empty]: set of X {},` `[]: ** over(A)`

Perfect `over`-expression is helpful. This expression allows a binary operator to be applied to a set, bag, or sequence. In this case the binary operators `++` and `**` may be utilized as described for set union and set intersection in Table 5.1 and they are applied to a set with element type `set of X`. In case set A does not contain elements, the result is simply a new empty set. In addition, Perfect provides for an even more elegant way of mapping generalized union: the pre-defined function `flatten` makes one big set out of a number of sets that have all elements of the same type.

5.2.2 Relations

Relations provide a means of combining two values into one pair. The simplest form of relations is the *binary relation*, further forms and operations that may be applied to relations are shown in Table 5.2.

The *binary relation* described in Table 5.2 is defined as $X \leftrightarrow Y == \mathbb{P}(X \times Y)$ in the

Table 5.2: Mapping relations and their operations to Perfect

Name	Usage in Object-Z	Mapping to Perfect
Binary Relation	$X \leftrightarrow Y$	`set of pair of (X, Y)`
Ternary Relation	$X \leftrightarrow Y \leftrightarrow Z$	`set of triple of (X, Y, Z)`
n-ary Relation	$X \leftrightarrow ... \leftrightarrow Z$	`set of nTuple of (X, ..., Z)`

Relation Usage	a _rel_ b
	(a, b, c) *in rel*
		`pair of (X, Y){a, b} in rel`
		`triple of (X, Y, Z){a, b, c} in rel`

Maplet	$x \mapsto y$	`pair of (X, Y){x, y}`

mathematical tool kit of the Z reference manual [Spi89]. So, the mapping to Perfect is basically just a combination of the mappings of *cartesian product* and *power set*. *Ternary relations* can be easily expressed in Perfect using triples. In Object-Z relations can be created with an arbitrary number of parameter elements. Perfect only provides `pair of` and `triple of` as equivalent expressions. For tuples or n-ary relations with $n \geqslant 4$, it is preferable to define new Perfect classes fourTuple, fiveTuple and so on, in the same ways as for pairs and triples. This makes the whole specification more readable and even simplifies the transformation as n-tuples only need to be prefixed by the appropriate class name, `of` and the list of types.

The following simple example illustrates what the mapping looks like using the instructions stated above. X1 to X6 are the types of the single elements:

(a, b, c, d, e, f)

```
sixTuple of (X1, X2, X3, X4, X5, X6){a, b, c, d, e, f}
```

Relation usage refers to the language construct used to show whether elements are related or an n-tuple is a subset of a relation. The last line in Table 5.2 provides a mapping for the *maplet* construct, a representation of an ordered pair as described in the mathematical toolkit of Z in [Spi89, chapter 4.2].

5.2.3 Functions

Functions are a special kind of relations. In Object-Z, there are several kinds of functions symbols that are used in declarations to define variables of functional types. Perfect uses the `map of (X -> Y)`-class to express such function types, but this construct does not reflect the invariants that are inherently stated by the different function symbols in Object-Z. So they have to be explicitly added to the invariants part of the mapping to Perfect. Table 5.3 provides an overview of the basic rules how functions and function references are mapped to Perfect and the invariants that have to be added for each function type.

In Table 5.3 it is easily observable, that the invariants given for the finite partial function and finite partial injection are equal to their non-finite counterparts. As with sets, maps

Table 5.3: Basic translation rules for functions and mapping different function types to Perfect

Function type	**Usage in Object-Z**	**Invariants to Be Added**
Function definition	$f : X \fun Y$	`f: map of (X -> Y)`
Access of functions	$f(x) = y$	`f[x] = 2 * y`
Partial function	$f : X \pfun Y$	no invariants to add
Total function	$f : X \fun Y$	`invariant forall x:X :- x in f;`

Partial injection	$f : X \pinj Y$	`invariant forall x1::f.dom, x2::f.dom`
		`:- f[x1] = f[x2] ==> x1 = x2;`

Total injection	$f : X \inj Y$	`invariant forall x1::f.dom, x2::f.dom`
		`:- f[x1] = f[x2] ==> x1 = x2;`
		`invariant forall x:X :- x in f;`

Partial surjection	$f : X \psurj Y$	`invariant forall y:Y :- y in f.ran;`

Total surjection	$f : X \surj Y$	`invariant forall y:Y :- y in f.ran;`
		`invariant forall x:X :- x in f;`

Bijection	$f : X \bij Y$	`invariant forall y:Y :- y in f.ran;`
		`invariant forall x:X :- x in f;`
		`invariant forall x1::f.dom, x2::f.dom`
		`:- f[x1] = f[x2] ==> x1 = x2;`

Finite partial function	$f : X \ffun Y$	no invariants to add

Finite partial injection	$f : X \finj Y$	`invariant forall x1::f.dom, x2::f.dom`
		`:- f[x1] = f[x2] ==> x1 = x2;`

in Perfect are implicitly finite and hence, the mapping is not completely equivalent for non-finite functions, but as long as a mapped value is included for each object of interest the result is still sufficient.

5.2.4 Operations on Relations and Functions

For relations and functions, there are several operations available. However, due to the fact, that relations are mapped differently to Perfect than functions, it is necessary to distinguish in all cases between functions and relations. First, the translation of domain and range operations is shown in Table 5.4. While the operations on functions can be mapped by the equally named functions `.dom` and `.ran`, a suitable mapping for relations can be created using a `for`-clause to select only the elements of the left or the right hand side of the relation.

Additionally, mappings for domain and range restriction and anti-restriction as well as overriding will be presented. First, the mappings for domain and range restriction and anti-restriction are presented in Table 5.5. These two operations aim on narrowing the possible pairs of a binary relation R to allowed or forbidden values (sets S or T) on left or right hand side of the pair. The mapping of this operation for functions has to

Table 5.4: Mapping of domain and range operations of functions to Perfect

Description	Usage in Object-Z	Mapping to Perfect
Relations		
Domain	dom *rel*	`for x::rel yield x.x`
Range	ran *rel*	`for x::rel yield x.y`
Functions		
Domain	dom *f*	`f.dom`
Range	ran *f*	`f.ran`

yield a `map of (X -> Y)`, whereas the mapping for relations has to result in a `set of pair of (X, Y)`. To create sets of pairs the `those`-clause can be used in Perfect. Domain and range values of a pair are accessed by selectors `x` and `y`, respectively. In case of functions, first the pairs are extracted from the original map and the set of restricted pairs is taken as input for the resulting map.

Table 5.5: Mapping domain and range restrictions to Perfect

Description	Object-Z	Perfect
Relations		
Domain Restriction	$S \lhd R$	`those x::R :- x.x in S`
Range Restriction	$R \rhd T$	`those x::R :- x.y in T`
Domain Anti-Restriction	$S \ntriangleleft R$	`those x::R :- x.x ~in S`
Range Anti-Restriction	$R \ntriangleright T$	`those x::R :- x.y ~in T`
Functions		
Domain Restriction	$S \lhd R$	`map of (X -> Y)` `{those x::R.pairs :- x.x in S}`
Range Restriction	$R \rhd T$	`map of (X -> Y)` `{those x::R.pairs :- x.y in T}`
Domain Anti-Restriction	$S \ntriangleleft R$	`map of (X -> Y)` `{those x::R.pairs :- x.x ~in S}`
Range Anti-Restriction	$R \ntriangleright T$	`map of (X -> Y)` `{those x::R.pairs :- x.y ~in T}`

The last available operation for relations and functions is the *override*-operation. According to the mathematical toolkit of the Z language [Spi89, p. 102] the override function $S = Q \oplus R$ means that in the resulting relation S, everything in the domain of R is related to what it is related in R, and everything else in the domain of Q is related to what it is related in Q. The mappings for relations and functions are shown in Table 5.6.

The mapping approach for relations is taking all the pairs of Q, in which the first value is not first value of any of the pairs in R, and join these pairs with all the elements of R. In Perfect, this can be expressed using a `those`-clause. In the case of functions, the mapping has to be separated into overriding of functions by other functions (translated to maps in Perfect) or by a relation or set (translated to a set of pairs). In the first case, the mapping is similar to domain and range restriction. The `map of (X->Y)` constructor taking a set of pairs builds a new map from elements in Q for which there is no corresponding element in map R, i.e. a pair with the same left hand side. The resulting map is united with R using the ++-operator. For the second case, the selection of the elements of *qMap* has to be modified to a Perfect transformation expression using the `for`-clause to adapt the different mapping of the Object-Z *dom* function to Perfect for relations as described earlier. Both cases are illustrated in Table 5.6.

Table 5.6: Mapping overriding for relations and functions to Perfect

Name	Object-Z	Perfect
Relations	$qRel \oplus rRel$	`(`**`those`**` q::qRel :- (`**`exists`**` i::rRel :- q.x = i.x))++ rRel`
Functions	$qMap \oplus rMap$	**`map of`**` (X->Y){ `**`for those`**` q::qMap.dom :- q ~`**`in`**` rMap.dom `**`yield pair of`**` (X, Y){q, qMap[q]}} ++ rMap`
	$qMap \oplus rRel$	**`map of`**` (X->Y) `**`for those`**` q::qMap.dom :- q ~`**`in`**` (`**`for`**` r::rRel `**`yield`**` r.x) `**`yield pair of`**` (X, Y){q, qMap[q]} ++ rRel}`

5.2.5 Sequences

Sequences are basically functions from the natural numbers to some other type X. Table 5.7 describes ways how to declare different kinds of sequences and lists functions on sequences together with their mappings to Perfect. It must be noted that Object-Z indices start at value 1, whereas Perfect starts at index 0. This difference has to be reflected in a modification of the mapping whenever elements of the sequence are accessed by their index numbers and the index number is not retrieved from the sequence itself, but rather user input or program input like a set of values.

Additionally to operations listed in Table 5.7, operations like *ran* and *dom* on binary relations as described in sections 5.2.2 and 5.2.3 are also applicable to sequences. Perfect operators `ran` and `dom` are also available for the sequence type. The Object-Z operator # denoting set size or number of elements may as well be applied to sequences and is mapped to the # operator in Perfect that returns the number of elements in the sequence.

Extraction on a sequence denoted by $U \upharpoonleft s$ with a set U of positive natural numbers and some sequence s means that only those sequence elements of s shall be adopted to the resulting sequence that have an index that is given in the set of positive natural numbers in U and that are at the same time valid indices in the sequence s. To accomplish a mapping with the same meaning in Perfect, first the set U is transformed into a non-decreasing sequence and all elements that exceed the number of valid indices in s

Table 5.7: Mapping sequences and operations for sequences to Perfect

Description	**Usage in Object-Z**	**Mapping to Perfect**
Finite Sequence	$s : \text{seq}\,X$	`s: seq of X`
Non-empty Finite Sequence	$s : \text{seq}_1\,X$	`s: seq of X` `invariant #s > 0`
Injective Sequence	$a : \text{iseq}\,X$	`a: seq of X` `invariant forall x :: a :- (x # a)= 1`
Empty Sequence	$\langle\,\rangle$	`seq of X{}`
Sequence Size	$\#s$	`#s`
Concatenation	$s1 \frown s2$	`s1 ++ s2`
Reverse	*rev s*	`s.rev`
Head of Sequence	*head s*	`s.head`
Last of Sequence	*last s*	`s.last`
Front of Sequence	*front s*	`s.front`
Tail of Sequence	*tail s*	`s.tail`
Extraction	$U \upharpoonleft s$	`for those i :: u.permndec :- i < #s yield s[i-1]`
Filtering	$s \upharpoonright V$	`those elem::s :- elem in V`
Prefix Relation	$s1 \text{ prefix } s$	`s.begins(s1)`
Suffix Relation	$s1 \text{ suffix } s$	`s.ends(s1)`
Segment Relation	$s1 \text{ in } s$	`s1 <<= s`
Distributed Concatenation	$\frown/\,q$	`flatten(q)`

are filtered out. Second, for each of these index elements the element of sequence s at this index is put into a new sequence, which results in the expected sequence.

Filtering is the counterpart to extraction. Applying filtering by some set V to sequence s with elements of the same type, denoted by $s \upharpoonright V$, yields a sequence of elements, where all elements are removed from the sequence that are not in V. The mapping to Perfect is similar as the one for range restriction, only those elements that are in the sequence and satisfy the condition to be also contained in set V are added to the resulting sequence.

The last function in Table 5.7, *distributed concatenation* creates one flattened sequence from the sequence of sequences of some type X. This operation strongly resembles the generalized union of sets and in fact it can be mapped to Perfect in the same way. The global function `flatten` is also defined for `seq of seq of X` in Perfect, so this function can be directly applied to q to yield the expected sequence of elements of type X.

5.2.6 Bags

Whereas sets may only ever contain equal items once, bags may include any number of the same item. Therefore, a special multiplicity operation, counting the occurrences of an item in the bag, is available for this kind of collection. Table 5.8 shows the mappings of bags and operations on these bags.

Table 5.8: Mapping bags and operations on bags to Perfect

Description	**Usage in Object-Z**	**Mapping to Perfect**
Bag	bag X	`bag of X`
Empty Bag	$[\![\ ]\!]$	`bag of X{}`
Multiplicity	$a \sharp x$	`a # x`
Bag Scaling	$n \otimes x$	`x.rep(n)`
Bag Membership	$a ⋿ x$	`a in x`
Sub-bag Relation	$x1 \sqsubseteq x2$	`x1 <<= x2`
Bag Union	$x1 \uplus x2$	`x1 ++ x2`
Bag Difference	$x1 \uminus x2$	`x1 -- x2`
Bag of Elements of a Sequence	*items s*	`s.ranb`

5.2.7 Collection Constructors

It is possible to explicitly construct a collection by listing their members explicitly for all three Object-Z collection types. Table 5.9 sums up the mappings for collection construction. The number of expressions building up the collection is arbitrary, i.e. one or more.

Table 5.9: Mapping collection construction expressions to Perfect

Description	**Usage in Object-Z**	**Mapping to Perfect**
Set	$\{e1, e2, e3, ...\}$	`set of X{e1, e2, e3, ...}`
Sequence	$\langle e1, e2, e3, ...\rangle$	`seq of X{e1, e2, e3, ...}`
Bag	$[\![e1, e2, e3, ...]\!]$	`bag of X{e1, e2, e3, ...}`

5.2.8 Set Abstraction

Another way to construct sets is making use of the Object-Z set abstraction construct:
$\{\mathit{declarations} \mid \mathit{predicates} \bullet \mathit{expression}\}$
The *declarations* part defines variables, the *predicates* part, if existent, may restrict the defined variables and the remaining *expression* part may transform values of declared variables. The result is the set of all these restricted and transformed variables. Some examples for this syntactic structure are given in appendix A of [DR00, p. 202].

Similar Perfect language constructs that could be used for a mapping are the `those` and the `for those` clauses depending on whether *expression* is provided or not. The first construct has the limitation that declarations with a predefined or user defined type do not produce a set of values, but a new type, so such expressions are only supported in a context where a type is expected, like the declaration of subtypes (section 4.8) or as type in variable declarations. In other contexts `those`-clauses may only be used in a way such that they deliver finite collections. In this usage context only the variation with bound variable declarations can be applied such that the mapping produces a set as result. Therefore, the type of the declaration either has to be a collection or a type restricted to a discrete number of values.

Finiteness can be achieved for natural numbers or integers, whenever the *predicates* restrict the set by a lower and upper bound. The implicit lower bound of natural numbers is $n \geqslant 0$. In such cases the set of values can be defined using the range operation $lowerBound \mathrel{..} upperBound$ introduced in section 5.2.1. A simple approach for finding a lower or upper bound for the type expressions is to take comparison expressions like $100 > varName$ with *varName* as only operand on one side into consideration. One can easily deduce an upper bound from such expressions.

For other pre- or user-defined classes, no transformation rules will be presented in this work because a much deeper analysis of the context would be necessary to provide a general mapping, which is out of scope of this work. Table 5.10 sums this up for set abstractions without transformation expressions.

Table 5.10: Mapping set abstraction without transformations to Perfect

Declaration Type	**Mapping to Perfect**
Collection Expressions	
$\{v : coll \mid pred\}$	
Sets	`**those** v::coll :- pred`
Bags or Sequences	`(**those** v::coll :- pred).ran`
Type Expressions	
$\{v : type \mid pred\}$	`**those** v: type :- pred` But limited usage
Natural numbers	`**those** v::(0 .. upperBound).ran :- pred` if *uppperBound* can be found in *pred*
Integer numbers	`**those** v::(lowerBound .. upperBound).ran :- pred` if *lowerBound* and *upperBound* can be found

As described in section 4.4.3 transformation expressions in Perfect only work with a bound variable declaration, which means that they can only be used when the type of the declared variable in the set abstraction is a collection expression, collection typed variables, or an upper and lower bound for an integer typed variable can be found just like for *set abstraction without transformations.* A mapping to Perfect for all other kinds of set abstraction is more sophisticated and a more in-depth analysis has to be made to be able to perform an automatic mapping. Therefore, no rules are presented here to transform such Object-Z expressions to Perfect. Table 5.11 sums up the mapping rules for set abstraction with transformations.

Table 5.11: Mapping set abstraction with transformations to Perfect

Declaration Type	**Mapping to Perfect**
Collection Expressions	
$\{v : coll \mid pred \bullet expr\}$	
Sets	`for those v::coll :- pred yield expr`
Bags or Sequences	`(for those v::coll :- pred yield expr).ran`
Type Expressions	
$\{v : type \mid pred \bullet expr\}$	
Natural & Integer Numbers	`for those v::(lowerBound..upperBound).ran :- pred` if *lowerBound* and *upperBound* can be found in *pred*
In General	No mapping, in depth case by case analysis necessary

According to the syntax definition it is also possible to declare more than one variable in the declaration section. However, neither Duke and Rose [DR00] nor Smith [Smi00] give examples of such set constructions. One can deduce from the semantics described in [DR00] that variables of different types would all be combined to a tuple of values. As Perfect cannot handle sets of different, non compatible types and the focus of this work is on Object-Z while only providing a basic set of transformation rules for the various expressions, finding a mapping covering all aspects and usage variations of set abstraction is left for future work.

5.2.9 Summation

A construct similar to set abstraction is *summation* with the following syntax:
Σ *schemaText* $\bullet$ *featureCall*
It can be used to sum a list of values, i.e. as in set abstraction *schemaText* introduces a set of values or objects and for each of these an object feature is accessed or a function is applied (*featureCall*). The resulting values are then added so that the summation finally delivers one result. In general, the operation is only applicable for features and functions with a type that has a +-operator defined.

Table 5.12: Mapping summation functions to Perfect

Usage in Object-Z	**Mapping to Perfect**
Σ *schemaText* $\bullet$ *featureCall*	`+ over(` *schemaText* $\bullet$ *featureCall* `)` Mapping von *schemaText* $\bullet$ *featureCall* wie bei set abstraction.

The mapping for the *schemaText* $\bullet$ *featureCall* part is the same as in Table 5.11 with *expression* substituted by *featureCall*. Table 5.12 illustrates that summation is achieved using the `over`-construct of Perfect in combination with the +-operator.

5.2.10 Numbers

Expressions may also be formed out of relations or operations on numerical values and variables. Mappings of arithmetic operators and relations are shown in Table 5.13.

Table 5.13: Mapping operations for numbers to Perfect

Name	**Symbol in Object-Z**	**Mapping to Perfect**
Binary	$+ - * /$ div mod	`+ - * / / %`
Unary	$-$	`-`
Comparison	$< \leqslant > \geqslant$	`< <= > >=`
Successor	$succ(n)$	`>n`

Boolean valued expressions are conjoined using logical connectives, not comparison operators because in Object-Z the latter are not applicable for the boolean values *true* and *false*. To read more about the mapping of logical operators, refer to section 5.2.16.

5.2.11 Genericity

Genericity in Object-Z classes has been introduced in section 3.2. Table 5.14 shows a summary how to map these generic parameters to Perfect and how a generic instantiation looks like in both languages. *Formal* and *actual parameters* are mapped to Perfect in the same way.

Table 5.14: Generics in Object-Z and Perfect

Type	**Usage in Object-Z**	**Mapping to Perfect**
Formal parameters	$Queue[Item]$	`Queue` **`of`** `Item`
Actual parameters	$ClassName[int, nat]$	`ClassName` **`of`** `(`**`int`**`,`**`nat`**`)`

5.2.12 Object References

In Object-Z an object can reference itself by using the keyword *self*. This allows, for example, to hand over the object itself to another object as a schema input parameter. In Perfect the same keyword is used to reference the object

Table 5.15: Self-Reference in Object-Z and Perfect

Usage in Object-Z	**Mapping to Perfect**
self	**`self`**

5.2.13 Equality in Object-Z and Perfect

Object-Z is a language that uses reference semantics. Therefore, each object is considered to have its own identity and two objects are considered equal only if they are really identical with respect to this identity. The inner state of the object is not examined when two objects c1 and c2 are checked for equality.

In contrast, Perfect uses another approach and considers all abstract data variables of the two objects to be compared for equality [Esc, chapter 5.3.3]. However, primitive types like natural numbers, or integers can be directly compared for equality. Collections make use of the equality operators of their elements. So collections of primitive types are compared whether both contain elements with the same values, but if the elements are arbitrary objects, comparison is based on the inner object status, not object identity.

A summary of how to map the equality operator to Perfect is given in Table 5.16

Table 5.16: Equality in Object-Z and Perfect

Description	**Usage in Object-Z**	**Mapping to Perfect**
Numbers or literals	$x =$ 'l'	`x = 'l'`
Two variables of primitive type	$x = y$	`x = y`
Two collections of primitive types	$x = y$	`x = y`
All other types	$x = y$	No fully correct mapping possible, `x = y` appropriate for objects that have a unique identifier in their abstract data.

One solution to overcome this problem is to explicitly use the `ref`-type in Perfect. However, Crocker and Carlton state that reference data types cause much overhead, are often even unnatural and due to the aliasing, cause severe problems in verifying the specification [CC04, 2.4 Object Identity].

In general the following statements about the practical influence can be made if the mapping for equality is provided for all kinds of types:

- If two objects are tested for equality and the answer is no, the user can be sure, that the objects are not identical.
- Only if two objects are considered equal, they may be identical, as well.
- If the system is designed in a way that a state variable contains a unique identifier for each object, equality in Perfect expresses that two objects are identical as otherwise the identifier was not unique.
- Systems in which it is possible that two different objects have the same inner state may deliver incorrect results with regard to identity and equality.

The operator $\neq$ is the negation of the equality operator and therefore all the problems discussed for equality still hold (with opposite meaning). However, to take advantage of the Perfect Developer verification engine, both operators are translated in this work as if both languages used the same kind of semantics.

5.2.14 Object Containment

Object-Z also uses the notion of *object containment*. As described above, Perfect primarily uses value semantics, which also implies that an object used as a state variable in another class, is always contained by the outer object. This means that using object containment in Object-Z does not raise the need for a special mapping, but rather specifies contained objects as if they were used in a value semantics language like Perfect.

In contrast, using state variables without object containment in Object-Z, is still mapped as if these variables were contained objects because of the value semantics in Perfect. If Object-Z refers to one object A by some other objects B and C, and then A is changed by a schema call, this means that the referenced object A for B and C stays the same when using reference semantics, but the inner state of A is different and these changes are visible for B and C. Mapping this situation directly to Perfect yields objects B and C containing each a copy of A. This means that whenever A changes, this is not reflected in the copies contained in B and C. To overcome this problem, it would be necessary to keep track of all copies of such an object and if a change occurs in one copy, this change is also propagated to the other ones.

If the specification is directly done in Perfect, the specifier is aware of the value semantics and can model the system appropriately with this in mind. For the first prototype keeping track of all objects and adding appropriate modifications will not be included in the mapping.

5.2.15 Predefined Types

Each of the predefined types available in Object-Z has a compatible type in Perfect, so that the mapping can be done directly. A new class `pNat` is introduced for the mapping of strictly positive integers if this type is used in the Object-Z specification. Table 5.17 lists these mappings.

Table 5.17: Mapping predefined types to Perfect

Type name	**Symbol in Object-Z**	**Symbol in Perfect**
boolean	$\mathbb{B}$	**`bool`**
integer	$\mathbb{Z}$	**`int`**
natural number	$\mathbb{N}$	**`nat`**

strictly positive integer	$\mathbb{N}_1$	`pNat` and add
		`class` `pNat ^=`
		`those` `n:`**`nat`** `:- n > 0` **`end;`**

real	$\mathbb{R}$	**`real`**
character	*Char*	**`char`**

5.2.16 Logical Operators and Conjunctives

Object-Z provides a total of seven logical operators and conjunctives, four binary, one unary, and two quantification operators to express constraints on variables within boolean expressions.

- Equivalence
- Implication
- Disjunction
- Conjunction
- Negation
- Universal Quantification ($\forall$)
- Existential Quantification ($\exists$ / $\exists_1$)

The first five operators can be directly translated to Perfect by substituting the operator with the Perfect equivalent and applying necessary transformations to the operands on the left and right sides according to the rules provided in other sections of this chapter. Table 5.18 lists the binary and unary logical operators with their Perfect equivalents.

Table 5.18: Mapping logical operators to Perfect

Name	**Symbol in Object-Z**	**Symbol in Perfect**
boolean literals	*true false*	**`true false`**
equivalence	$\Leftrightarrow$	`<==>`
implication	$\Rightarrow$	`==>`
disjunction	$\vee$	`|`
conjunction	$\wedge$	`&`
negation	$\neg$	`~`

Universal and Existential Quantification

The remaining quantification expressions cannot be mapped that straightforward. Before a thorough discussion is presented for each construct in subsequent sections, some common topics are explained in advance.

A quantified predicate has the form $Q\ schemaText \bullet predicate$ where Q is one of the three quantification operators $\forall$, $\exists$ or $\exists_1$, *schemaText* consists of a declaration- and an optional predicate section (*declarations* | *predicates*) that introduces new variables restricted by the predicates and *predicate* is the constraint that has to be satisfied for all, at least or exactly one value respectively.

Bound Variable Declarations

The variables declared in the *schemaText* section are also referred to as bound variables, as they are declared in the scope of a quantifier and cannot be freely set to any arbitrary value. These variables may be declared to have some predefined type like int, bool, real or nat, or some user-defined class. Another alternative is declaring them as one of the collection types. Object-Z treats these options syntactically equally, but Perfect makes use of two syntax variations, either `:` or `::`. The mappings for both types are shown in Table 5.19.

Table 5.19: Bound variable declarations in Object-Z and Perfect

Description	Usage in Object-Z	Mapping to Perfect

Pre- or user-defined types	$\forall x : \mathbb{N}, y : \mathbb{N} \bullet y = x * x \Rightarrow y > x$	`forall x:nat, y:nat`
		`  :- y = x*x ==> y > x`

Collection types	$\exists x : \{1, 2, 3\} \bullet x * x < 16$	`exists x::set of nat{1,2,3}`
		`  :- x*x < 16`

Mapping SchemaText without Predicate

As described in the previous paragraphs, the *schemaText* part has a *declaration* and an optional *predicate* part. First the focus is put on the alternative without the predicate. In this simpler form of quantification it does not make a big difference, how many variables are declared in the schema text. The two following examples show how universal and existential quantification are mapped to Perfect.

$$\forall c : \{2, 3, 4, 5, 6, 7, 20, 30, 40, 56, 100\} \bullet c \bmod 2 = 0$$

The variable c is declared to be a member of the given set of integer numbers. The predicate checks if the remainder of c divided by 2 is 0, i.e. c is an even number. The mapping for this expression to Perfect is straightforward. Listing 5.1 shows how this Object-Z quantification is translated to Perfect.

```
forall c::set of int {2,3,4,5,6,7,20,30,40,56,100} :- c % 2 = 0;
```

Listing 5.1: Simple “forall” example in Perfect

The second example presents existential quantification and declares two variables.

$$\exists a : \mathit{setA}, b : \mathit{setB} \bullet a < 20 \land b < a$$

This example checks whether there exists one combination of an element a of *setA* and an element b of *setB*, where a is smaller than 20 and b is smaller than a. Both *setA* and *setB* are collections, so the Perfect mapping contains `::` as separator between variable name and type. The equivalent construct in Perfect is given in Listing 5.2.

```
exists a::setA, b::setB :- a < 20 & b < a
```

Listing 5.2: Simple “exists” example without constraint in Perfect

Table 5.20 summarizes the mapping of quantification expressions without predicates in the schema text for both existential and universal quantification. This corresponds to the mapping described by Tim Kimber in [Kim07] for quantified expressions.

Table 5.20: Mapping of quantified expressions without schema text predicate

Usage in Object-Z	**Mapping to Perfect**
$\forall$ *declarations* ● *predicate*	`forall declarations :- predicate`
$\exists$ *declarations* ● *predicate*	`exists declarations :- predicate`

The `declarations` used in Table 5.20 are mapped to Perfect according to the rules described in Table 5.19. The `predicate` is either a logical expression or some other expression that yields a boolean value.

Mapping SchemaText with Predicate

In the previous part only *schemaTexts* without predicates have been taken into consideration. Although Tim Kimber does not provide a mapping for this construct in [Kim07], a mapping is possible under some certain circumstances. In the simplest case, the quantification in Object-Z makes use of only one declared variable and contains an arbitrary number of $\wedge$-conjoined constraints as predicates that only depend on this variable, some constants or variables declared outside this construct. To map such an expression to Perfect, the
texttttthose-clause introduced in section 4.4.2 is helpful to describe a collection of values restricted by constraints.

Extending the first example from the previous section by the constraint $c > 10$ in the *predicate*-section of the *schemaText* of the specification restricts the values of the set from which values are chosen by those that satisfy the additional constraint. Listing 5.3 shows the mapping to Perfect using a `those`-clause.

```
forall c::(those i:: set of int{2,3,4,5,6,7,20,30,40,56,100} :- i > 10) :- c % 2 = 0
```

Listing 5.3: Simple “forall” example with constraint in Perfect

More complicated quantifications can be expressed in Object-Z by declaring more than one variable and including all these variables in the predicates of the schema text. Consider an example where all pairs of *players* shall be tested if the values of their feature *remain* are not equal. Obviously, it is desirable to exclude comparing an element with itself, as this would naturally result in the predicate to evaluate to false:

$$\forall p1, p2 : players \mid p1 \neq p2 \bullet p1.remain \neq p2.remain$$

In Perfect there is no such construct of first choosing all values of a set and then restricting the pairs to the interesting ones, but one may alter the collection from which the values are taken in the first place by using a
texttttthose-clause instead of the collection. If one analyzes the example more thoroughly, one will see that it is sufficient that the first variable may take any value of *players* whereas the second may only be set to all the other values of *players*, but not the one chosen as $p1$. This results in the mapping to Perfect given in Listing 5.4.

```
forall p1::players, p2::(those p3::players :- p3 ~= p1)
       :- p1.remain ~= p2.remain
```

Listing 5.4: Players example in Perfect

In general it can be said that for *schemaTexts* in quantifications with two variables, the predicates in the schema can be separated into three groups, predicates restricting only the first, predicates restricting only the second variable, and predicates restricting both. If there are predicates of the first group, then a `those`-clause also has to be introduced for the first variable. If only predicates of group two and three exist, both kinds of restrictions are added to the `those`-clause of the second variable.

Table 5.21: Mapping of quantified expressions with predicates in schema text

Usage in Object-Z	**Mapping to Perfect**
$\forall v1 : type1 \mid pred1 \bullet pred$	No mapping
$\forall v1 : coll1 \mid pred1 \bullet pred$	`forall v1::(those tempVar::coll1 :- pred1):- pred`
$\forall v1 : coll1, v2 : coll2, ... \mid preds \bullet pred$	`forall` `v1::(those t1::coll1 :- preds(v1)),` `v2::(those t2::coll2 :- preds(v2)), ... :- pred` with an appropriate variable ordering <v1, v2, ...>

Table 5.21 shows a summary of how quantified expressions with predicates in the *schemaText* can be translated to Perfect. If a bound variable has user- or pre-defined type, it can only be mapped, if there are no restricting predicates for this variable, because the `those`-clause would have to use this type as well, but the usage of this construct is not allowed in the context of a quantification expression. So the one-variable-case is not applicable for such variables, but they may still appear within a list of bound variables. Additional mapping coverage for such types might be possible in some ways as well, but this analysis is left for an in-depth discussion in future work.

For collection types the mapping is performed using a `those`-clause which restricts the original collection by the predicates of the *schemaText*. If the declaration list contains more than one variable, it is necessary to find an appropriate ordering of the variables, such that newly introduced `those`-clauses only make use of the currently defined variable and variables that have already been defined before. First the *schemaText* predicate has to be split into sub-predicates. A predicate can be split into two sub-predicates, if these two sub-predicates are conjoined by $\wedge$. Each sub-predicate depends on one or more of the declared variables. The term *preds*($var1$) refers to exactly those predicates of the original *schemaText* predicate that are relevant for the declaration of variable $var1$.

An ordering, if any exists, can be calculated by the ordering algorithm created for this work and presented in Appendix A that uses a graph built from predicate and variable nodes, where each variable is connected to a predicate whenever this variable appears in the predicate. The graph may contain both, variables with user- and predefined type or collection type.

Unique Quantification

Whereas the normal existential quantification using $\exists$ can be handled by the Perfect counterpart `exists` and the mappings described above, the second operator $\exists_1$ must be treated specially. In the Z reference manual [Spi89] Spivey describes this operator as *Unique quantifier* and an equivalence using existential and universal quantifiers is given:

$$(\exists_1\, x : A \bullet ...x...) \Leftrightarrow (\exists\, x : A \bullet ...x... \land (\forall\, y : A \mid ...y... \bullet y = x))$$

Therefore the mapping to Perfect combines the mappings of existential and universal quantification. To illustrate the mapping, Table 5.22 shows a minimum example using one declaration without constraining predicates. A solution for more variables and even with *predicates* in the *schemaText* can be found by properly applying the mappings for existential and universal quantification which could also mean that no mapping is possible.

Table 5.22: Mapping unique quantification to Perfect

Usage in Object-Z	**Mapping to Perfect**

$\exists_1\, x : aSet \bullet pred$	`exists x::aSet :- pred &`
	`  (forall t1::(those t2::aSet :- pred[t2/x]) :- t1 = x)`

5.2.17 Variable Declarations

Declarations of variables can be directly mapped to Perfect constructs. Table 5.23 shows the mappings found in this work for all three syntactic variations of variable declarations.

Table 5.23: Mapping variable declarations to Perfect

Description	**Usage in Object-Z**	**Mapping to Perfect**
Single-declaration	$x : T$	`var x: T`
Multi-declaration	$x, y : T$	`var x, y: T`
Polymorphic declaration	$x : \downarrow T$	`var x: from T`

5.3 The Class Construct

The structure of an Object-Z class has been described in section 3.1. Figure 5.1 and Listing 5.5 show how the basic class construct with inheritance, and formal parameters, but without definitions, state and operation schemas can be mapped to Perfect.

```
 ClassName[X, Y] ___________________________________
|
| ...
| InheritedClass1[X]
| ...
|____________________________________________________
```

Figure 5.1: An Object-Z class skeleton with inheritance and generics

```
class ClassName of (X, Y) ^=
    inherits InheritedClass1 of X
    ...
end;
```

Listing 5.5: Mapping of class with inheritance and generics to Perfect

Perfect supports only single-inheritance. Therefore, Object-Z specifications that specify more than one inherited class will not be taken into consideration in this mapping. One inherited class can be included in the Perfect-`inherits` part. Figure 5.1 and Listing 5.5 also illustrate the definition of a generic class and the usage of the formal parameter in both languages.

5.4 Local Definitions

As described in section 3.7 there are four types of local definitions. All of them may either appear within a class definition or on global level. The scope of a local definition starts at the point of the definition and ends at the end of the class in which it is defined.

5.4.1 Given Types

In order to translate *given types* to Perfect in this work, it is necessary to add a new class definition as an inner class to the abstract part of the Perfect specification. The resulting mapping is shown in Table 5.24.

This enables using *BasicType*1 within the defining class as a type. The definition also provides a build-function so that objects of that type can be instantiated. As a basic type definition does not provide more information about the structure of this type, the class does not have any other features and the build-function does not take parameters. One advantage of this translation is that the specification in Perfect can be easily extended by variables, functions or schemas.

Table 5.24: Mapping given types to Perfect

Usage in Object-Z	Mapping to Perfect
$[BasicType1]$	`class BasicType1 ^=` `interface` `build{};` `end;` one class definition per given type in the abstract section

5.4.2 Free Type Definitions

Free type definitions are mapped to Perfect enumeration class definitions. Table 5.25 shows the resulting mappings for definition and usage of free types.

Table 5.25: Mapping free type definition to Perfect

Description	Usage in Object-Z	Mapping to Perfect
Definition	$Color ::= blue \mid green \mid yellow \mid red \mid black \mid white$	`class Color ^= enum blue, green, yellow, red, black, white end;`
Access	$b.color = blue$	`b.color = Color blue`

5.4.3 Axiomatic Definitions

Although there are constants in Perfect, the Object-Z semantics are different. In Object-Z a constant only has to maintain its value for each object after it has been initially set whereas in Perfect constants are named place holders for globally fixed values. Therefore, the presented mapping uses variables to simulate the constant behavior expected by Object-Z.

$$\begin{array}{|l}
limit : \mathbb{N} \\
\hline
limit \in \{10, 20, 30\}
\end{array}$$

In this simple example a constant *limit* is declared that might have one value of the elements of the set given in the predicates section. The steps that have to be taken to map axiomatic definitions to Perfect are:

- A variable with the name and type of the Object-Z constant is declared in the `abstract` section of the Perfect class definition
- The predicate is mapped according to expression mapping rules and is added as an invariant to the `abstract` section, as well.

- The constructor has an input parameter to initialize the value of the constant. In the precondition of the constructor the predicates that constrain the constant are included, so that it may only be set to allowed values.

Listing 5.6 shows these rules applied to the example above.

```
abstract
  var limit: nat;
  invariant limit in set of nat {10, 20, 30};
interface
  build{!limit:nat}
    pre limit in set of nat {10, 20, 30};
```

Listing 5.6: Axiomatic definition mapped to Perfect

5.4.4 Abbreviation Definitions

Perfect does not provide a construct that expresses the same as Object-Z abbreviation definitions, but for one type of expressions a mapping to Perfect can be achieved in a relatively easy way using the `those`-construct. Although other expressions might be mappable as well, a mapping will not be presented for them, as each expression has to be regarded separately. Often abbreviation definitions are used as place holder for a set or range of values like in the following examples.

$$\begin{aligned}&\mathit{BingoNumber} == 1..75\\&\mathit{SmallSquareNumbers} == \{1, 2, 4, 9, 16, 25, 36, 49\}\end{aligned}$$

These can be mapped to Perfect as follows:

```
class BingoNumber ^= those x:nat :- x in (1..75).ran;
class SmallSquareNumbers ^=
   those x:nat :- x in set of nat {1, 2, 4, 9, 16, 25, 36, 49};
```

That means for an abbreviation definition *Identifier* == *Expression* where *Expression* is a collection typed expression a new class named *Identifier* is created with a `those`-clause as class body. There, a temporary variable `x` is introduced with the element type of the collection as type and with the constraint that `x` is element of *Expression*.

5.4.5 Definitions on Global Level

If given, free type, axiomatic, or abbreviation definitions are declared on global level, outside of any class, these definitions are regarded as globally visible throughout the whole specification. The reason for this is, that in Z there is no way of expressing visibility which means that all identifiers have to be globally visible.

Some modifications for the transformation rules above have to be applied so that the semantics are adapted:

Abbreviation, free, and given type definitions are expressed as class or enumeration class definitions in Perfect, so they can be simply placed outside of all other classes on global level.

Axiomatic definitions on local level are translated to declarations and predicates within a class. Variable declarations may not be simply put outside a class definition in Perfect. Therefore it is necessary to introduce a class that simulates globally accessible axiomatic definitions. This is done by adding a class `Global` which contains all global axiomatic definitions. To make all constants and functions globally visible, the transformation rules for mapping visibility described in section 5.7 have to be applied.

5.5 State Schema

In section 3.8 the state schema with its two kinds of variables and its predicates that restrict their possible values has been presented. The described mapping is based on the state schema shown in Figure 5.2.

StateSchemaMapping

$stateVar1 : type1$
Δ
$secVar1 : typeS1$

$predicate1$
$predicates(secVar1)$

Figure 5.2: State schema in Object-Z

5.5.1 Primary Variables

Several steps have to be performed to map primary variables to Perfect appropriately. For each Object-Z variable *stateVar*:

- A variable declaration is added to the abstract section
- For proper initialization, the class constructor takes an input parameter for each variable *stateVar* with the same name.
- Each predicate that only contains primary variables, constants, or visible features of other classes, but no secondary variables, is added as invariant to the abstract section and as precondition to the constructor.

- If the type of a variable implies additional invariants according to the mapping rules presented in section 5.2, these predicates are also added as invariants and preconditions just like explicitly stated predicates.

Listing 5.7 shows how the primary variables of the schematic Object-Z class given at the beginning of this section are mapped to Perfect using these rules.

```
class StateSchemaMapping ^=
abstract
  var stateVar1: type1;
  invariant predicate1;
interface
  build{!stateVar1: type1}
    pre predicate1;
end;
```

Listing 5.7: Primary state schema mapped to Perfect

5.5.2 Secondary Variables

Secondary variables are in fact functions that represent a fixed relation between primary state variables, constants and literals with exactly one value at any time. Therefore the mapping makes use of functions in Perfect, in which constraints given in the predicates part of the state schema form the function body. Depending on visibility the secondary variable function is put either in the confined or interface part of the specification.

The function has the same name and type as in the Object-Z specification and both are mapped according to the rules of section 5.2. Additionally, all predicates of the state schema that have this secondary variable in their specification are appropriately mapped to Perfect and then added as function body. If it is necessary to split predicates into smaller parts, this is possible where the outmost operator is a conjunction. Listing 5.8 shows the results for the Object-Z class assuming that the secondary variable is globally visible.

```
class StateSchemaMapping ^=
interface
  function secVar1: typeS1
  ^= predicates(secVar1);
end;
```

Listing 5.8: Secondary state schema mapped to Perfect

5.6 Inital State Schema

According to the description of the initial state schema given in section 3.9 the mapping of this construct has to cover two purposes. First, objects have to satisfy the predicates listed in this schema on object creation time. Second, at any time the initial state schema has to tell whether the inner object state satisfies the conditions given. To achieve the first goal, the predicates have to be added appropriately as preconditions to the class constructor in Perfect. A function returning a boolean value is added to the specification, that states whether the initialization conditions are currently satisfied by the object or not, for achieving the second goal. Listing 5.9 shows the mappings for the initial state schema in Figure 5.9.

INITMapping

INIT

*predicate*1
*predicate*2

Figure 5.3: INIT schema in Object-Z

It has to be said that normally the names of state variables and constants remain the same when mapped to Perfect. However, if modifications are necessary in any way, the same modifications have to be applied to each appearance of such an identifier, i.e. appearances in *predicate*1 or *predicate*2. Note that the predicates refer to the input parameters of `build` in the constructor and to abstract data members of the Perfect class in the `INIT`-function.

```
class INITMapping ^=
abstract
  ...
interface
  build{!state1: t1, !state2: t2, ..., !const1: t3,...}
    pre predicate1 & predicate2;

  function INIT: bool
  ^= predicate1 & predicate2;
end;
```

Listing 5.9: Initial schema mapped to Perfect

5.7 Visibility

Visibility of a feature in Object-Z is expressed by the visibility list. According to Smith [Smi00, p.46] the following kinds of features may appear in this list:

- constants
- state variables
- the initial state schema
- operations

Absence of the visibility list indicates visibility for all the features within the class. The list above does not include basic, free type, or abbreviation definitions, which means that such local definitions will never appear visible from outside the class. In contrast, Perfect determines visibility and accessibility depending on the section of the specification in which a definition appears. The mapping of features in the visibility list is performed as stated in Table 5.26

Table 5.26: Mapping of features in visibility list

Feature Type	**Access**	**Mapping to Perfect**	**Section**
Constant	read-only	`function constName;`	interface

State variable	read
	write
		`function var1 ;`
		`schema !set_var1(p_var1: type1)`
		`  pre invariants(var1)[p_var1/var1]`
		`  post var1!=p_var1;`
		Change all write accesses to `var1` outside the class definition to a call to `set_var1`.
			interface
			interface

INIT function		`function INIT: bool`
		`^= predicatesInINITFunction;`
			interface

Operations		no special mapping rule	interface

Write access can be enabled by adding a `set`-schema in the interface section that assures that all the class invariants referring to this variable, denoted by `invariants(varName)` are satisfied if `varName` takes the new value. Additionally, all occurrences of the form $varName' = expression$ have to be transformed to read `!set_varName(expression)`. If a primed variable is modified from outside the class, but does not appear alone on one side, no mapping can be provided.

If features are not listed in the visibility list, it is nevertheless important to make them visible for inheriting classes to model the visibility properties of Object-Z. To accomplish this, the same mapping rules as for visible features have to be applied, but the mapping output has to be put in the *confined* instead of the *interface section.*

5.8 Operations

The different kinds of operations in Object-Z and their building blocks have been described in section 3.10. It is necessary to take a closer look at their purpose for the mapping to Perfect. Several high-level helper functions are defined here to describe the mappings of operations in the following.

opName(): The name of the operation.

inVars(), inVarDecls(): The set of all input variables (postfix "?" in Object-Z) and their corresponding variable declarations respectively.

outVars(), outVarDecls(): The set of all output variables (postfix “!” in Object-Z) and their corresponding variable declarations.

auxVars(), auxVarDecls(): The set of all other auxiliary variables defined within an operation schema, that are neither input nor output variables and their corresponding variable declarations.

deltalist(): The set of primary variable identifiers listed in the deltalist part of the operation definition.

communicationVars(): The set of variables that act as communication from one operation to another.

commonOutVars(): The set of output variables that are common between two operations.

preconditions(): The set of all predicates of the predicate list that make up the precondition of this operation schema. Preconditions contain unprimed state or input variables, but may never include primed state, auxiliary, or output variables.

postconditions(): The set of all predicates of the predicate list that contain primed state or output variables.

usedStateVars(): The set of all the state variables used by this operation in unprimed form.

If any of the helper functions are used in a mapping, the elements form comma-separated lists unless stated differently. As variable names in Perfect may only consist of alphanumerical characters and underlines, input, and output variables always have to be transformed in the mapping to Perfect. Table 5.27 shows the applied rules.

Table 5.27: Mapping of input and output variables of operations

Usage in Object-Z	**Mapping to Perfect**
$inputVar?$	`inputVar_in`
$outputVar!$	`outputVar_out`

5.8.1 Operation Schemas

The following sections present the transformation rules for different kinds of operation schemas to Perfect. The characteristics, the mapping rules, and a simple example are provided for each type in this chapter.

No-Change Operation

The *No-Change* or *Boolean Operation* refers to operations that do not change the state of an object nor output any values. However, these operations provide information about the current state of the object, i.e. whether all preconditions of the operation hold.

isBelowBound

$bound? : \mathbb{N}$

$balance < bound?$

$bound? \geq 0$

```
function isBelowBound (bound_in:nat): bool
  ^= balance < bound_in & bound_in >= 0;
```

Listing 5.10: Operation isBelowBound mapped to Perfect

In the case when the precondition of an operation is not satisfied in Object-Z, the operation is considered not to be applicable according to [Smi00, p. 55]. In contrast, in Perfect a function may only be called if the preconditions are satisfied otherwise the validator produces an error message. Boolean functions are typically used in operation expressions where they indicate whether the whole operation is applicable. These semantics have to be simulated by the mapping. Therefore, boolean operations are mapped to boolean-valued functions in Perfect in this work. The *isBelowBound* example operation in Listing 5.10 illustrates the mapping from Object-Z to Perfect.

Table 5.28 sums up characteristics of no-change operations and their mappings. Expressions in italic refer to the mapped values of results of the helper functions presented earlier. If *op.inVarDecls*() is an empty set, the surrounding parentheses are left out. If *op.preconditions*() returns more than one element, the single items are conjoined using the `&`-operator of Perfect.

Table 5.28: Mapping of no-change operations

Characteristics	deltalist() = ∅ outVars() = ∅ postconditions() = ∅

Mapping to Perfect	**`function`** `opName(`*inVarDecls()*`)`**`: bool`**
	`^=` *preconditions()*`;`

No-Change Operation with Output

The next, more complex type of operation schemas are *no-change operations with output* or *function operations*. The difference to the previous type is the presence of one or more output variables and predicates that make use of them. In this work these operations are mapped to Perfect functions with one or more output values as this inhibits changing the state of the object. The operation *fundsAvail* is a simple function operation that uses a state variable *balance* and a constant *limit*.

fundsAvail

$funds! : \mathbb{N}$

$funds! = balance + limit$

```
function fundsAvail_prec
^= (exists tempVar:nat :- tempVar = balance + limit);
function fundsAvail: nat
    pre  fundsAvail_prec
    ^=  balance + limit;
function fundsAvailAlt funds_out: nat
    pre  fundsAvail_prec
    satisfy result.funds_out = balance + limit;
```

Listing 5.11: Function fundsAvail in Perfect

Listing 5.11 shows two possible mappings to Perfect with two approaches for mapping output variable declarations and postconditions. This work favors the second approach, because it is more flexible when it comes to the number of output parameters and also makes it possible that several predicates set constraints on a single output variable. The precondition introduces an existential quantification for all output variables to mimic the intention that functions in Object-Z are not applicable if preconditions or the implicit precondition that the postconditions can be satisfied do not hold. The operation characteristics and mapping rules are summarized in Table 5.29.

Table 5.29: Mapping of functions to Perfect

Characteristics	deltalist() = ∅ outVars() ≠ ∅ postconditions() ≠ ∅

Mapping to Perfect	`function opName(`*inVarDecls()*`)` *outVarDecls()*
	  `pre` *preconditions()*
	  `satisfy` *postconditions()*`;`
	Adding an existential quantification for output variables to preconditions and replacing each output variable *outVar* by `result.outVar` in postcondition

Output Reference	`opName(`*inVars()*`).outVar`

Change Operations

In contrast to the two previously presented types of operations, change operations may modify the inner state of an object, i.e. change the value of state variables. Only primary state variables listed in the deltalist may be changed by these operations. Changes of the inner state of objects are expressed using modifying schemas in Perfect, which is illustrated in the following example in Listing 5.12.

withdrawAvail

$\Delta(balance)$

$amount! : \mathbb{N}$

$amount! = balance + limit$

$balance' = -limit;$

```
schema !withdrawAvail(amount_out!: out nat)
    pre exists tempVar1:nat, tempVar2:int
        :- tempVar1 = balance + limit & tempVar2 = -limit
    post change balance, amount_out
         satisfy amount_out' = balance + limit,
                 balance' = -limit;
```

Listing 5.12: Operation withdrawAvail in Perfect

In the `post`-clause the variables to be changed by the schema are listed after keyword `change`. These are all primary variables specified in the deltalist of the Object-Z operation schema and, additionally, all output variables. The `satisfy`-part, lists the postconditions of the operation. If the Object-Z operation specification includes auxiliary variables, these are declared as temporary variables at the beginning of the postcondition and are valid only throughout this part of the Perfect specification. The precondition consists of the translation of all preconditions of the Object-Z specification and an existential quantification stating that values for output parameters and changed primary variables exist that satisfy the postcondition. A summary of the characteristics of change operations and the mapping to Perfect is given in Table 5.30

5.8.2 Operation Expressions

As introduced in section 3.10.3, operation expressions use one or more operation schemas to build more complex operations. Depending on the type of operation schema described in section 5.8.1, the mapping delivers either a function with or without output or a schema in Perfect. Whenever a change operation is involved, the result will have the structure of a change operation with the change-satisfy postcondition replaced by a postcondition list. Here, output variables are always postfixed by “ ! ”. Otherwise, if there are output variables in the resulting operation, then the structure of function operations

Table 5.30: Mapping change operations to Perfect

Characteristics	op.deltalist() $\neq \varnothing$ op.postconditions() $\neq \varnothing$

Mapping to Perfect	`schema !opName(`*inVarDecls(), outVarDecls()*`)`
	  `pre` *preconditions()*
	  `post change` *deltalist(), outVars()*
	      `satisfy` *postconditions()*`;`
	Adding an existential quantification for output variables and changed primary variables to preconditions

Auxiliary Variables	**`satisfy(var`** `auxVar:type;` *postconditions()* `);` for each auxiliary variable *auxVar*
Output Declaration	`outVar!:`**`out`** `type`

is used. In all other cases, the final operation is built like a boolean operation which never has postconditions.

In Listing 5.12 the mapped preconditions have been directly added to the precondition section of the schema. However, this approach has some shortcomings if it comes to operation expressions which may call functions or schemas from other classes. Sometimes, not all features used in a precondition are visible globally but in fact, it is sufficient to provide a means of stating whether the precondition of a mapped operation schema holds or not. Therefore, this precondition check is extracted to a boolean-valued *precondition function*, which has the same visibility as the mapped operation. Listing 5.13 shows the prototype of such a function.

```
function op_prec(op.inVarDecls()):bool
^= op.preconditions();
```

Listing 5.13: Precondition functions

Operation Promotion

The operation promotion propagates a call of a schema of this or another class. Therefore, the operation type (function or schema) remains the same as in the wrapped operation. Input and output variables are propagated to the calling operation.

In the introductory section 3.10.2 class *Book* has been specified with operation *return* that takes one input parameter *lender*? and modifies two state variables *lent* and *mayBeReviewed*. In Figure 3.9 operation *returnBook*1 propagates operation *return* called on *book*1 to the specification of *SmallLibrary*. In Perfect, the operation *returnBook*1 is mapped to a schema because *return* of class *Book* has a non-empty deltalist. Listing 5.14 shows the mapping of both operations.

```
class Book ^=
interface
    function return_prec(lender_in: string):bool
         ^= lent <==> true & lender = readerHistory.last &
            (exists lent_temp:bool, mayBeReviewed_temp:bool :-
                (mayBeReviewed_temp <==> false & lent_temp <==> false));
    schema !return(lender_in: string)
        pre  return_prec(lender_in)
        post change lent, mayBeReviewed
             satisfy lent' <==> false & mayBeReviewed' <==> false;
end;
class SmallLibrary ^=
interface
    function returnBook1_prec (lender_in:String): bool
        ^= book1.return_prec(lender_in);
    schema !returnBook1 (lender_in:String)
        pre returnBook1_prec(lender_in)
        post book1!return(lender_in);
end;
```

Listing 5.14: Operation promotion in Perfect

The mapping rules for promoting operation *op* of object *o* as operation *opPromo* are summarized in Table 5.31. Accesses to helper functions of *opPromo* are used without the operation name to simplify the mapping description. Preconditions of this operation type are only a propagation of the preconditions of the inner operation, i.e. calling the precondition function *op_prec*. In function operations output parameters are promoted to the outer operation by assigning the result of accessing each output variable `outVar` of the function call to `result.outVar` in the `satisfy`-part. Change operations only call the inner operation with all the input and output parameters. Object and operation name are separated by an exclamation mark instead of the dot to indicate a change in the calling object. If no caller is explicitly given, *self* is assumed resulting in the term `!op1` as caller of the inner operation.

Table 5.31: Mapping of operation promotions to Perfect

	opPromo $\hat{=}$ o.op

Characteristics	inVars() = op.inVars()
	outVars() = op.outVars()

Operation Type	same as *op*
Bool Function Body	**pre** `o.op(`*inVars()*`);`
Function Body	**pre** `o.op_prec(`*inVars()*`);` **satisfy result**`.outVar = o.op(`*inVars()*`).outVar` for all output variables *outVar*
Change Operation Body	**pre** `o.op_prec(`*inVars()*`);` **post** `o!op(`*inVars(), outVars()*`);`

The following sections describe the mappings of the binary operation expressions that have been introduced in section 3.10.3. The type of the resulting operation can be deduced in the same way for all the following operations. If either operation is a change operation, the result will also be a change operation translated into a Perfect schema. If none is a change operation and there are output variables in the resulting operation, the combined operation is mapped to a Perfect function. In the remaining cases the result is a boolean function.

Operation Conjunction

Operation conjunctions in Object-Z combine modifying and non-modifying schemas in a parallel manner as if the variables and predicates of the two schemas were just merged into a new one. This also means that variables with the same name in both schemas have to be equal in the resulting conjunction. The following situations must be considered in more detail:

- Input or output parameters with equal names
- Calling operations on the same object in modifying schemas

In the following, we assume dealing with conjunction operation $conjOp$, which is built from the two operations $op1$ and $op2$ defined in the same or some other class.

$$conjOp \mathrel{\widehat=} a.op1 \wedge b.op2$$

Basic Mapping Rules Before going into more detail, the general case shall be considered and the basic mapping rules are formulated. In conjunction operations the set $inVars()$ is formed by combining the two sets $op1.inVars()$ and $op2.inVars()$ where duplicates are removed, which is consistent with the intended meaning of variables with the same name. The set of $outVars()$ is built similarly. The resulting precondition is created by conjoining the preconditions of both operations.

In contrast, one has to differentiate for postconditions regarding operation types. If the resulting operation is a no-change operation with output, each output variable is propagated to $conjOp$ as presented for operation promotion. In change operations, output variables may have different sources, either function or change operations. In the latter case each output variable is simply used as output variable in the parameter list of the operation call. Output variables originating from function operations have to be explicitly assigned to the output variable. Table 5.32 illustrates the basic mapping for conjunction operations.

Input or Output Parameters with Same Names Input variables can be easily used by two operation calls within one operation in Perfect. However, if $op1$ and $op2$, both use the same output variable $outVar$, a solution to correctly reflect these equated variables has to be found. First, it has to be said that Object-Z implicitly only allows the composite operation to be applicable if there is an object that satisfies all the conditions

Table 5.32: Mapping of conjunction operations to Perfect

	conjOp $\hat{=}$ a.op1 $\wedge$ b.op2
Characteristics	inVars() = op1.inVars() ∪ op2.inVars(), outVars() = op1.outVars() ∪ op2.outVars(), commonOutVars() = ∅ a ≠ b
Preconditions	`a.op1_prec(`*op1.inVars()*`) & b.op2(`*op2.inVars()*`);` where *op*1 is a function or schema and *op*2 is a boolean function

Postconditions	**Functions**
	`result``.outVar1 = a.op1(`*op1.inVars()*`).outVar1 &`
	`result``.outVar2 = b.op2(`*op2.inVars()*`).outVar2 & ...`
	for all output variables *outVar*1 of *op*1 and *outVar*2 of *op*2
	Change Operations
	`outVar! = a.op1(`*op1.inVars()*`).outVar`
	for each output variable *outVar* if op1 is a function
	`a!op1(`*op1.inVars(), op1.outVars()*`)`
	for each change operation
	all postconditions combined by '`,`'

to act as output variable of both operations. Therefore, a precondition has to be added stating that there exists one particular value that may fulfill this condition. If the operation is a change operation an additional postcondition function `op_post` has to be added to provide the value of output variables without doing the actual change to the calling object.

Regarding the postcondition, again, there is a difference between function and change operations. In the combination of two functions or a function and a change operation, one of the promotions of a common output variable is simply omitted in the mapping. This is necessary, because an object cannot be modified twice in parallel in Perfect. In the combination of two change operations, this problem can be solved by introducing a temporary variable which substitutes one of the two appearances of the output variables. This modification in combination with the precondition ensures, that the mapping still delivers a correct result in this work.

Table 5.33 provides an overview on the modifications necessary to handle the mapping of common output variables between any kind of combination of functions and schemas.

Calling Operations on the Same Object in Modifying Schemas In the previous cases, the caller objects have been considered different. This section presents how to map calls on the same object properly. As both operations may modify an arbitrary set of state variables the mapping has to make sure that references to unprimed state variables always refer to the not yet changed value.

A second difficulty in the mapping of these operations is that calling modifying schemas

Table 5.33: Modifications with common output variables

	conjOp $\widehat{=}$ a.op1 $\wedge$ b.op2
Characteristics	commonOutVars() $\neq \varnothing$
Operation Type	same as without common output

Preconditions	additional to the precondition without common output:
	`exists tempVar:type :-`
	`  tempVar = a.op1(`*op1.inVars()*`).outVar1 &`
	`  tempVar = b.op2(`*op2.inVars()*`).outVar1`
	one temporary variable declaration for each common output variable.

Postconditions	**Functions**
	Only include one occurrence of output variable promotion.
	Functions and Change Operations
	Do not include output variable promotion of function operation.
	Two change operations:
	Include temporary variable declaration for each common
	`var tempVar:type;`
	`  a!op1(`*op1.inVars(), op1.outVars()*`),`
	`  b!op2(`*op2.inVars(), op2.outVars()*`);`

in parallel on the same object is not possible in Perfect. To provide a mapping that simulates a very similar behavior in this work, the individual postconditions may be combined using the keyword `then` to call them in a sequence. An ordering has to be found in which the second operation does not refer to state variables changed by the first operation. In Object-Z the order of operations in the operation conjunction is not significant so that operands may as well be exchanged.

To determine a pairwise order between two operations *op*1 and *op*2, one makes use of the helper function *usedStateVars*(). The rules are presented in Table 5.34. The pairwise order can be used as input for a topological sorting algorithm as presented by Knuth in [Knu97] to find an ordering for more than two operations. If any order of operations can be found, the only adoption to be made is applying the found order and replacing the separating comma by the term `then`.

Parallel Composition

This next composite operation is very similar to the conjunction operation, but also allows for variable communication from output variables of one to input variables of the other operation. However, if two operations are conjoined using the parallel or associative parallel composition operator, but neither schema provides output variables nor output-input communication pairs, the mapping is done according to the rules presented for conjunction operations.

As with conjunction operations there are certain conditions that have to be taken into

Table 5.34: Finding a pairwise order for conjunction operations

```
isBefore(op1, op2)
if ($op1.deltalist() \cap op2.deltalist() \neq \varnothing$):
 error;
else if ($op1.deltalist() \cap op2.usedStateVars() = \varnothing$):
 return true;
else if ($op2.deltalist() \cap op1.usedStateVars() = \varnothing$):
 return false;
else
 error;
```

special consideration:

- Input or output parameters with equal names
- Operations with communication variables
- Calling operations on the same object in a modifying schema

The complete mapping of parallel composition operations builds up on some basic characteristics that are presented in Table 5.35.

Table 5.35: Basic characteristics of parallel compositions

parOp $\widehat{=}$ a.op1 $\parallel$ b.op2
communicationVars() = (op1.outVars() $\cap$ op2.inVars()) $\cup$ (op2.outVars() $\cap$ op1.inVars())
inVars() = (op1.inVars() $\cup$ op2.inVars()) $\setminus$ communicationVars()
outVars() = (op1.outVars() $\cup$ op2.outVars()) $\setminus$ communicationVars()
commonOutVars() = (op1.outVars() $\cap$ op2.outVars()) $\setminus$ communicationVars()

Preconditions and postconditions are basically formed in the same way as described for conjunction operations. It has to be noted that in parallel composition operations all the output variables that have been present in one of the constituent operations may disappear. This happens in case each output variable functions as a communication variable. The mapping rules for common input and output variables given in section 5.8.2 can be directly used for parallel composition. The only speciality to be noted is that before considering which input or output variables are equated, all the communication variables have to be eliminated, because they are no longer part of the external interface of the composite operation.

Mapping Communication Variables According to the Object-Z language definition the operations combined by parallel composition happen at the same time and the operation is commutative. Consequently, communication may happen from the left to the right operation, vice-versa or even in both directions in parallel. In Figure 3.10 operation *transferAuthors*1 has been presented to illustrate the use of this kind of operation expression. The left-sided operation *authorList* has an output variable *authors*!, and the operation on the right has an input variable *authors*?. As both communication variables have the common base name *authors*, the two are used for internal communication. In the mapping the same semantics have to be reflected. The operation signatures of *authorList* and for *setAuthorList* in Perfect are:

```
function authorList authors_out: seq of string
schema !setAuthorList(authors_in: seq of string).
```

Therefore, the resulting operation has no more input or output variables to the environment, but only one internal communication variable. This means that the value of the output variable has to be used as input parameter to the change operation.

Operation *setAuthorList* uses the temporary communication variable *tempVar*1 in primed form to state that the input parameter is the final value of *tempVar*, i.e. after having assigned the output of the other operation. However, the dependency between the operations implied by the communication variable is not yet reflected in the mapping done in this work. The only way to introduce a new variable in a precondition is adding a quantification. In this case an existential quantification states that there exists one value or object that is at the same time output of the left operation and input of the right operation. The final mapping of the operation achieved in this work is shown in Listing 5.15.

```
function transferAuthors1_prec: bool
^= book1.authorList_prec &
    (exists tempVar1: seq of string :-
        tempVar1 = book1.authorList.authors_out &
        book2.setAuthorList_prec(tempVar1));
schema !transferAuthors1
pre transferAuthors1_prec
post (var tempVar1: seq of string;
      tempVar1! = book1.authorList.authors_out,
      book2!setAuthorList(tempVar1'));
```

Listing 5.15: Communication variables in parallel composition

In general, the concept of mapping preconditions for this construct is to introduce a new temporary variable for each communication variable in an existential quantification expression and state that this temporary variable is equal to the communication output variable. Postconditions introduce temporary variables to hold the values of the communication output variables and transfer them to the other operation, as well. Once again a distinction has to be made between function and change operations. Table 5.36 sums up the mapping rules for parallel compositions with communication variables.

If a parallel composition operation contains both, common output variables and communication variables, then the resulting existential quantifications or variable declaration

Table 5.36: Mapping communication variables in parallel compositions to Perfect

	parOp $\widehat{=}$ a.op1 $\parallel$ b.op2

Modified Preconditions	∀ commVar:communicationVars() •
	exists *tempVarDecl(commVar)* :-
	tempVarL1 = a.op1_post(*op1.inVars()*, tempVarR1).commVar1
	tempVarR1 = b.op2(*op2.inVars()*, tempVarL1).commVar2
	a.op1_prec(*op1.inVars()*, tempVarR1)&
	b.op2_prec(*op2.inVars()*, tempVarL1)

Modified Postconditions	**Functions** Similar to preconditions, value of communication variable is assigned to temporary variable and then used instead of communication input variable. **exists** *tempVarDecl(commVar)* :- tempVarL1 = a.op1(*op1.inVars()*, tempVarR1).commVar1 tempVarR1 = b.op2(*op2.inVars(*), tempVarL1).commVar2 **result**.outVarL1 = a.op1(*op1.inVars()*, tempVarR1).outVarL1 **result**.outVarR1 = b.op1(*op2.inVars()*, tempVarL1).outVarR1 **Functions and Change Operations** **var** *tempVarDecl(commVar)*; a!op1(*op1.inVars()*, tempVarR1', *op1.outVars()*, tempVarL1!), tempVarR1! = b.op2(*op2.inVars()*, tempVarL1').commVar2, outVarR1! = b.op2(*op2.inVars()*, tempVarL1').outVarR1 **Change Operations** **var** commTempVarDecls(); a!op1(*op1.inVars()*, tempVarR1', *op1.outVars()*, tempVarL1!), b!op2(*op2.inVars()*, tempVarL1', *op2.outVars()*, tempVarR1!)

parts in schemas have to be merged into one such construct because temporary variables for communication variables have to be accessible by predicates for common output variables as well.

Calling Operations on One Object When both operations of the parallel composition are change operations and called from the same object, the same problem with concurrent modifications of one object arises as already described for conjunction operations. The solution for parallel composition is almost the same as for conjunction operations, but in combination with communication variables the mapping has to be modified. The problem arises because of the primed variables in the postconditions of schemas when communication variables are mapped.

```
var commTempVarDecls();
    a!op1(op1.inVars(), op1.outVars(), tempVarL1!) then
    a!op2(op2.inVars(), tempVarL1, op2.outVars())
```

Listing 5.16: Parallel communication with one caller object

Here, `op1` is first executed and `tempVarL1` has a distinct value at this moment. Next `op2` is executed and needs to access the final value of `tempVarL1`, but as this variable has not yet been modified in the part after `then`, `tempVarL1'` does not have a distinct value at that moment. To overcome this problem, the prime is simply removed from all the communication input variables. Mapping to sequential order is only possible if the second operation does not need to access the initial value of features of `a` that are modified by the first operation. Another shortcoming of this mapping is that two-way communication can no longer be translated, because the final value of all the input variables of $op1$ must be fixed before $op2$ may be applied.

Associative Parallel Composition

A variation of parallel composition is the associative form. In Object-Z, the only difference is that output variables remain visible to the environment. This means that only a few changes have to be applied to the mapping of parallel composition.

Input and output variable sets as well as the set of common output variables do no longer exclude the communication variables. Predicates are formed in exactly the same way as for parallel composition, although the result is not the same for both, as associative parallel compositions might have more common output variables, that produce their own predicates in the preconditions.

The main difference in the mapping of postconditions is that they have to reflect the fact that output variables involved in communication are propagated and not hidden. In the mapping, this is done by explicitly setting the reference to the communication output variable in the postcondition to the value of the introduced temporary variable. A summary on the necessary modifications on variable sets and postconditions for associative parallel composition is provided in Table 5.37.

Table 5.37: Differences of associative compared to parallel composition

	assParOp $\widehat{=}$ a.op1 $\parallel_!$ b.op2

Characteristics	inVars() = op1.inVars() ∪ op2.inVars()
	outVars() = op1.outVars() ∪ op2.outVars()
	commonOutVars() = op1.outVars() ∩ op2.outVars()

Postconditions	∀ commVar:communicationVars() •
	where *tempVar* is used as substitute for *commVar*
	add a term depending on operation type
	Function Operation
	`**result**.commVar = tempVar`
	Change Operation
	`commVar! = tempVar'`

Nondeterministic Choice

In comparison to the previous composition operations, nondeterministic choice has one major difference: Both operations must have the same operation signature, i.e. the same input and output variables. Combinations of change and no-change operations are nevertheless possible.

In choice operations only one of the two operations will be executed and the chosen one depends on which one is applicable. One can directly deduce the preconditions of nondeterministic choice operations from this definition. A nondeterministic operation *nonDetChoice* may only be applied if at least one of the two inner operations is applicable. Therefore the preconditions of both operations are combined by a logical or (|). Postconditions on the other hand have to reflect that the postconditions of *op*1 or *op*2 are only satisfied if the preconditions of the respective operation are satisfied. In Perfect, the bracketed form of postconditions is helpful to accomplish the mapping. As the mapping result never provides a default guard, the keyword `opaque` has to be used in the mapped operation to reproduce the nondeterministic semantics. Table 5.38 summarizes the mapping of this operation composition.

Table 5.38: Mapping nondeterministic choice to Perfect

	nonDetChoice $\widehat{=}$ a.op1 [] b.op2
Characteristics	inVars() = op1.inVars() = op2.inVars()
	outVars() = op1.outVars() = op2.outVars()
Preconditions	op1.preconditions() | op2.preconditions()
Postconditions	**Functions**
	`([a.op1.preconditions()]:`
	`  `**`result`**`.outVar1 = a.op1(`*op1.inVars()*`).outVar1 & ...`
	` [b.op2.preconditions()]:`
	`  `**`result`**`.outVar1 = b.op2(`*op2.inVars()*`).outVar1 & ...)`
	Functions and Change Operations
	`([a.op1.preconditions()]:`
	`  outVar1! = a.op1(`*op1.inVars()*`).outVar1 & ...`
	` [b.op2.preconditions()]:`
	`  b!op2(`*op2.inVars()*`, outVar1!)& ...)`
	Change Operations
	`([a.op1.preconditions()]:`
	`  a!op1(`*op1.inVars()*`, `*op1.outVars()*`),`
	` [b.op2.preconditions()]:`
	`  b!op2(`*op2.inVars()*`, `*op2.outVars()*`)`
	Declare the combined operation `opaque` for the nondeterministic behavior.

Sequential Composition

The only operation expression that implies an order on the combined operations is the sequential composition. Similar to parallel composition, communication is possible using output and input variables with the same base name, but this communication may only happen from the first to the second operation. Regarding applicability of a sequential composition it has to be said, that the whole operation may only be used in a situation where the precondition of the first operation is satisfied and the precondition of the second operation is satisfied based on the state reached after executing the first operation. Therefore the mapping of the sequential composition is equal to the mapping of parallel composition with one-way communication from the first to the second operation without common output variables and where left and right operation do not change the same object.

If both operations are called on the same object and the first operation modifies the state of the caller, then the precondition of the second operation might have to access the value of a state variable changed by the first operation. In this case the new precondition cannot be built with the means already introduced. It is necessary to have access to a variable that holds the state of the caller modified by the first operation. This can be achieved by an additional helper function in the class of the caller that returns an object of this class with the state variables set to the values of the caller object modified by the first operation. Table 5.39 sums up the mapping rules created in this work for sequential composition referring to the mapping rules of parallel composition as far as applicable.

Scope/Environment Enrichment

The scope enrichment operation expression aims on providing the variables declared in the operation on the left side to the operation on the right side, which is, except for communication variables, not possible for any of the other operation expression types. Although the operation on the left hand side might be any kind of operation expression, this work will only present a mapping for flat operation schema declarations. Usually, the left operation of scope enrichment selects an object out of an existing collection (like in *scopeEnr*1) of elements by applying a predicate *pred*. This chosen object acts then either as a caller on the right hand side or as an input variable to an operation (like in *scopeEnr*2) that takes an input variable with the same name. These variations are illustrated as follows:

$$\mathit{scopeEnr}1 \widehat{=} [a : \mathit{collection} \mid \mathit{pred}(a)] \bullet a.\mathit{op}1$$
$$\mathit{scopeEnr}2 \widehat{=} [a? : \mathit{collection} \mid \mathit{pred}(a?)] \bullet \mathit{op}2$$

Instead of selecting only one variable, there might be several which can all be used by the predicates. The actual mapping depends on the declaration type of the selectors and on the operation type of the operation expression on the right hand side.

Table 5.39: Mapping sequential composition to Perfect

	seqComp $\widehat{=}$ a.op1 $\fatsemi$ b.op2

Characteristics	communicationVars() = op1.outVars() ∩ op2.inVars()
	inVars() = (op1.inVars() ∪ op2.inVars()) \ communicationVars()
	outVars() = (op1.outVars() ∪ op2.outVars()) \ communicationVars()
	commonOutVars() = (op1.outVars() ∩ op2.outVars()) \ communicationVars()

	Sequential order is predefined: ⟨ op1, op2 ⟩

Preconditions	If $a = b$ and $op1$ is a schema
	Introduce an additional temporary variable with type of a
	that holds the state of a modified by $op1$ provided by helper function
	`op1_constr`
	`exists` *tempVarDecls()* `:-`
	`tempVar1 = a.op1(`*op1.inVars()*`).cOutVar1 & ...`
	`tempVarA = a.op1_constr(`*op1.inVars()*`) &`
	`tempVarA.op2_prec(`*op2.inVars()*`, tempVar1)`
	No preconditions for common output variables.
	Preconditions are equal to parallel composition in all other cases

Postconditions	Postconditions are similar to parallel composition, but in functions only include promotions of common output as call from right hand side, in schemas ’`,`’ is replaced by **`then`**

First, this work takes a closer look on the case in which the selector is an input variable. Here, $a?$ comes from outside and the mapping only has to make sure, that the value satisfies the predicate on the left side and the preconditions of the operation on the right. The operation on the right is mapped according to its own mapping rules described in one of the sections of this chapter while the operation type remains the same.

If a has no suffixes, then it is in fact an auxiliary variable in Object-Z, this means, that it is not set by the environment, but is fixed to a certain value locally and then the right-hand side operation is involved on this object. Again, this operation might be a no-change operation with or without output or a change operation. For the latter case, this also implies, that the inner state of a will be modified by the operation on the right. Therefore the step of setting a and then modifying it must be executed sequentially which is expressed by `then`. The necessary mapping rules for scope enrichment are summarized in Tables 5.40 and 5.41 for scope enrichments with input variables and auxiliary variables as variables with enriched scope, respectively.

This concludes the list of operation expressions and the description of the mapping for distributed operators is the only remaining part of the transformation rules in the chapter about operations.

Table 5.40: Mapping scope enrichment with an input variable to Perfect

	scopeEnr $\widehat{=}$ [a?: collection | pred(a?)] • op2

Characteristics	inVars() = left.inVars() ∪ op2.inVars()
	outVars() = op2.outVars()
	left.outVars() assumed to be empty
	'a?' as input variable
	preconditions() = left.preconditions() ∪ op2.preconditions()

Postconditions	**Functions**
	∀ outVar:outVars •
	`result.outVar = op2(`*op2.inVars()*`, a?).outVar`
	Change Operations
	`op2(`*op2.inVars()*`, a?, `*op2.outVars()*`)`

Table 5.41: Changes of the mapping of scope enrichment with an auxiliary variable

	scopeEnr $\widehat{=}$ [a: collection | pred(a)] • a.op2
Characteristics	'a' as auxiliary variable
Preconditions	Add existential quantification for auxiliary variable: `exists a::collection :- `*pred(a)* `&` *op2.preconditions()*

Postconditions	introduce a temporary variable for the declared auxiliary variable
	Functions
	∀ outVar:outVars() •
	`exists a::collection :- `*pred(a)* `&`
	`result.outVar = a.op2(`*op2.inVars()*`).outVar`
	Change Operations
	`var a:`*collection.type()*`;`
	`(a! = (any tempVar1::collection :- `*pred(tempVar)* `&`
	`a.op2_prec(`*op2.inVars()*`)))then`
	`a!op2(`*op2.inVars(), op2.outVars()*`)`

Distributed Operators

Distributed operators generalize the binary operation expressions from the previous sections to operations performed on an arbitrary number of objects. Usually, these objects are taken from a flat operation schema similar to scope enrichment.

In *distributed conjunctions* the most important issues in the mapping are that all operations happen in parallel and input and output variables with the same name are equated and therefore have to be equal. As the same operation is applied to all of the callers, it only makes sense that one caller does not appear more that once in the list of callers, which also means that we can assume that concurrent modifications on one caller object will never happen. The exact mapping rules depend once again on the operation type of the inner operation *op* as this is shown in Table 5.42. In the mapping of schema

Table 5.42: Mapping of distributed conjunction to Perfect

	distrConj $\widehat{=}$ $\bigwedge$ a:collection • a.op

Characteristics	inVars() = op.inVars()
	outVars() = op.outVars()

Preconditions	**Bool Functions**
	`forall a::collection :- a.op(op.inVars())`
	Functions
	for each output variable a temporary variable is declared.
	`(exists tempVarDecls() :- (forall a::collection :-`
	`  tempVar1 = a.op(inVars()).outVar1 &`
	`  tempVar2 = a.op(inVars()).outVar2 & ... ))&`
	`  (forall a::collection :- a.op_prec(inVars()))`
	Change Operations
	similar to functions, but `a.op` is replaced by `a.op_post` in existential quantification for common output variables.

Postconditions	**Functions**
	for each output variable a temporary variable is declared.
	`exists tempVarDecls() :-`
	`  result.outVar1 = a.op(inVars()).outVar1 &`
	`  result.outVar2 = a.op(inVars()).outVar2 & ... &`
	`  (forall a::collection :-`
	`    tempVar1 = a.op(inVars()).outVar1 &`
	`    tempVar2 = a.op(inVars()).outVar2 & ...))`
	Change operations
	A temporary variable is declared for each output variable, as a map of elements with type of caller to type of output variable. This mapping is only applicable if *collection* is a sequence or is transformed into one.
	`var tempMap1: map of (a.type()-> outVar1.type()); ...`
	` (tempMap1! = map of (a.type()-> outVar1.type()){}, ... )then`
	` (collection[0]!op(inVars(), outVars()),`
	`  forall i::1..#collection :-`
	`    collection[i]!op(inVars(), tempMap1[collection[i]]!,`
	`                    tempMap2[collection[i]]!, ...))`

postconditions the idea is to set the final value of the output parameters in the call of the first operation and for all other operations the value of the output variable is written to a temporary variable. Therefore, a map is introduced that can hold these values when calling all the other operations.

The *distributed nondeterministic choice* selects only one of the operations for execution, either the only one for which the individual precondition is satisfied or one of the operations where its precondition is fulfilled if there are several. In the remaining cases, the whole operation has to be mapped as not applicable. The transformation rules are presented in Table 5.43.

Table 5.43: Mapping of distributed nondeterministic choice to Perfect

	distrChoice $\widehat{=} [\!]$ **a:collection • a.op**

Characteristics	inVars() = op.inVars()
	outVars() = op.outVars()

Preconditions	**Bool Functions**
	`exists a::collection :- a.op(`*inVars()*`)`
	Functions and Change Operations
	`exists a::collection :- a.op_prec(`*inVars()*`)`

Postconditions	**Functions**:
	for each output variable a temporary variable is declared.
	`(exists a::collection :- (a.op_prec(`*inVars()*`) &`
	  `result.outVar1 = a.op(`*inVars()*`).outVar1 &`
	  `result.outVar2 = a.op(`*inVars()*`).outVar2 & ... )`
	Change operations
	A temporary variable is declared for each output variable, as a map of elements with type of caller to type of output variable. This mapping is only applicable if *collection* is a sequence or is transformed into one.
	`var a:`*a.type()*`;`
	  `a! = (any tempVar1::collection :-`
	        `tempVar1.op_prec(`*inVars()*`))then`
	  `a!op(`*inVars(), outVars()*`)`

For the third operation, *distributed sequential composition*, one sequential order of elements of the collection has to be found that is applicable. This means that all permutations of the arbitrary number of objects has to be checked, which grows exponentially with the element count. Therefore, a mapping to a checker that iterates over all permutations seems to be inoperative. The only other way is to express the precondition using a quantification that states that there is a permutation of the calling objects such that the combined operation built from the permutation order yields a feasible result. However, stating these semantics in terms of the Perfect language has been considered too complex for this work because even more helper functions would be necessary and so no mapping will be presented for distributed sequential composition, here.

Inheritance in Operations

Some important remarks regarding inheritance have to be made for the mapping from Object-Z to Perfect. In Object-Z inherited operations always extend the operation of the superclass, i.e. the operation is conjoined with new predicates. Up until now, we have only used postconditions in the mapping to Perfect, not post-assertions. However, when it comes to mapping inheritance, these *post-assertions* are very important, because they are the inherited part of the operation. Therefore, if an operation is declared in a class that inherits from another, then all the operations with the same name in the super classes have to be enriched with such post-assertions. In fact, this means that postcondition and postassertion will have the same predicates, but it is not possible to leave out either part if inheritance shall be mapped.

This concludes the description of transformation rules from Object-Z to Perfect. The subsequent chapters will present more details on the implementation, a case study, that shows the results of the implemented tool and an evaluation of the mapping and implementation.

CHAPTER 6

Implementation

This chapter provides details on how the mapping rules described in the previous chapter have been transformed into an implementation. The structure and building blocks of the implemented tool are presented here.

6.1 Design of the Mapping Tool

The transformation from Object-Z to Perfect is established in several subsequent process steps. Figure 6.1 shows the processing steps that a specification will undergo during transformation. The box named “Object-Z grammar” represents the parser and lexer grammars of Object-Z, which are provided together with the translation tool. The boxes on the right side show the names of the tool components that will be involved into the described transformation steps.

In the first step, *lexical analysis*, the *OZLexer* analyzes the original Object-Z specification according to the provided lexer grammar and splits the input into a sequence of tokens. In the *parsing* step, the *OZParser* generates a parse tree from the token input and the given grammar that represents the specification. Afterwards, several steps to semantically analyze the parse tree are performed by the *OZTranslator*. This component builds an internal representation of the Object-Z specification on top of the parse tree, iteratively enhanced by all necessary information, such as symbols and type information, to create a Perfect specification as output, by traversing the parse tree several times. During the last step, the *TemplateRenderer* combines all involved templates and prints them out to standard output.

6.2 Tool Implementation

As already proposed in section 2.5, *Tools for Language Transformations*, the implementation is based on the programming language Java, in version 8, in combination with

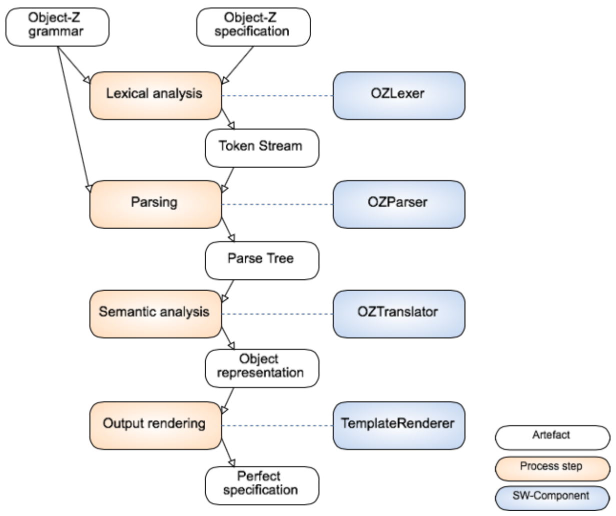


Figure 6.1: Transformation steps within the mapping tool and high-level responsibilities of the components

ANTLR 4 and the StringTemplate templating engine. The following subchapters present more details on the implementation of the components of the mapping tool done in this work.

6.2.1 Input Format – Modified Object-Z Syntax

As shown in Chapter 3, Object-Z uses a graphical syntax that defines the barriers between their logically compound blocks by means of boxes to separate the single blocks. In addition, many non-ASCII characters are used to start special purpose sections. Therefore, LaTeX markup is available to typeset Object-Z specifications properly. Another way to visualize Object-Z specifications is using Unicode encodings to reference all the necessary special characters. Although the grammar of Object-Z is provided in the work of Smith [Smi00], and Duke and Rose [DR00], neither of these works specify the lexical tokens in full detail. Due to these facts a lexer implementation that is based on Unicode syntax or LaTeX markup would have to make some assumptions about the box structures.

Therefore, and furthermore to ease the writing of the specification, this work uses a mapping based on a simplified Object-Z syntax that solely depends only on ASCII characters by replacing the graphical boxes with parenthesized blocks and special characters with keywords. This syntax is adapted from the existing syntax presented by Tim Kimber [Kim07]. Blocks use either curly braces or parentheses and are prefixed by their name. Within Object-Z boxes some constructs use horizontal lines to separate into subsections. These separators are in practice not necessary to correctly parse an Object-Z specification, so they can be simply left out. However, the section of secondary state variables still has to be labelled so that one can distinguish between primary and secondary variables. In the modified syntax, these secondary variables are enclosed by a block with the name "`delta`" within the state schema. An operation block is started with the name of the operation, and within this block the Δ sign has been replaced by the keyword "`delta`" to mark the deltalist of a change operation.

In essence, the syntax has only been adapted as little as necessary, and as far as possible the original Object-Z syntax is maintained. Noticeably, there is no change in the order of language elements within a class or smaller block, so the parser for this parenthesized-block-ASCII-only syntax can be easily substituted by one that accepts LaTeX markup or a unicode character syntax, while still producing the same tree structure, so that the traversal of the tree works fine on both notations. In the following, this syntax is just called *OZ* to distinguish it from the original Object-Z specification language syntax.

The examples given in the case study in Chapter 7 illustrates the modified syntax. The grammar and symbol descriptions used for the translation tool can be found in Appendix B whereas the source code of the whole tool is available at the Object-Z 2 Perfect translation project page[1].

6.2.2 OZLexer and OZParser

Based on the grammar described before, lexer and parser have been generated using ANTLR in version 4.7 with Java as target language. The implementation of the semantical analysis, that is done by *OZTranslator*, bases on the listener interfaces that are automatically created by ANTLR. In this work, the visitor interface classes are not necessary, so there is no need to create them.

After running the `antlr` command on the input grammar file `OZ.g4` six files are generated, two token files (`OZ.tokens` and `OZLexer.tokens`), two classes that implement the lexical, and syntactical analysis (`OZLexer.java` and `OZParser.java`) and two files to provide the listener functionality (the interface `OZListener.java` and the actual implementation `OZBaseListener.java`). The implemented tool uses all these files programatically to trigger the translation process. First, it reads the OZ definition saved in the file at the path provided on startup. This file is analyzed and split into tokens by OZLexer. Then these tokens are parsed by OZParser to produce a parse tree that represents the Object-Z specification.

To illustrate this, an exemplary, slightly simplified parse tree for a sample OZ method (Listing 6.1) is provided in Figure 6.2. The square boxes represent the terminal nodes that correspond to tokens of the specification language. The names of the leave nodes, as they are used by lexer and parser are not included in the diagram for simplicity reasons. The circular nodes are non-terminal nodes, which represent a single rule or production in the grammar of OZ. Reading the tree from top to button, one can see that the outermost parse tree node is *operationSchemaDef* which consists of two terminal nodes containing the texts `withdraw` and `{`. The next parts are the non-terminal nodes

[1]`https://github.com/sylviaswoboda/objectz-2-perfect`

deltalist, *declarationList*, and *predicateList*. The grammar production *operationSchemaDef* ends with a terminal node containing a closing curly brace `}`. The non-terminal nodes are then again split into their building blocks, until all the paths end in terminal nodes.

```
withdraw{
    delta(balance)
    amount?: !N;
    balance' = balance - amount?;
}
```

Listing 6.1: Sample method written in OZ

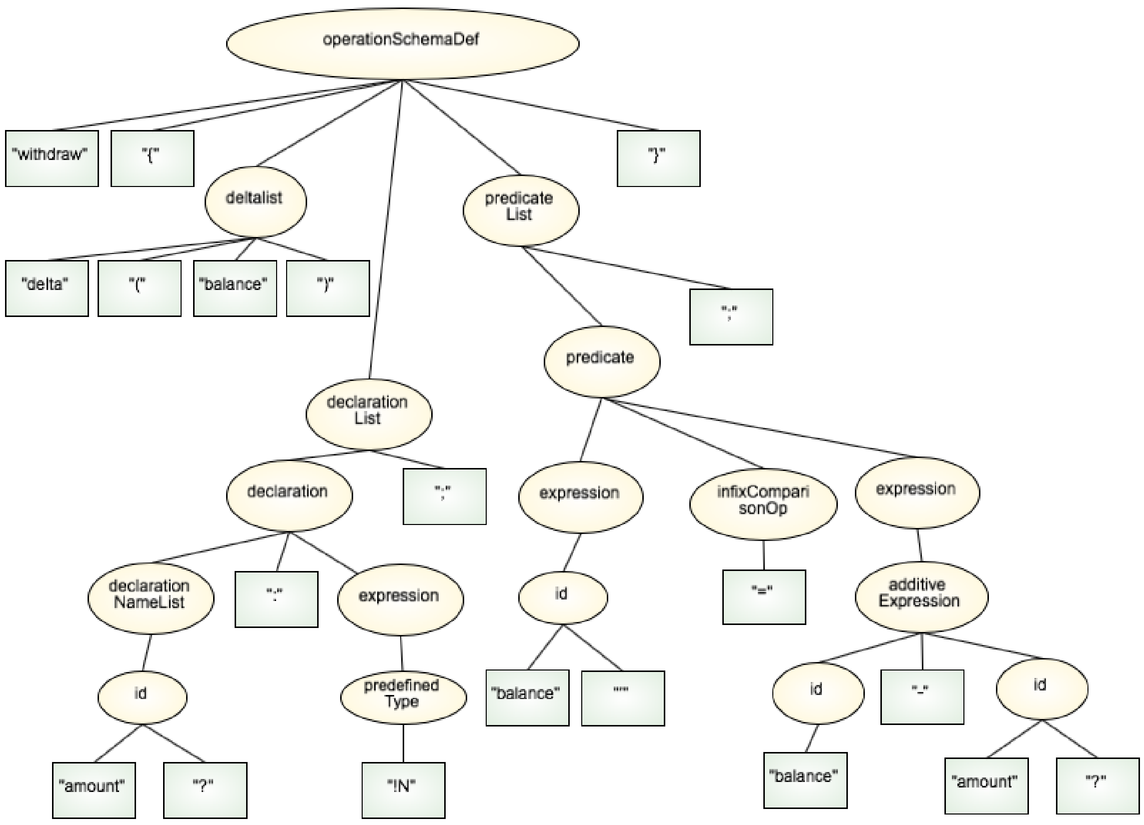


Figure 6.2: Example parse tree of method "withdraw"

The latter two files created by ANTLR, `OZListener.java` and `OZBaseListener.java` have not yet been used so far, but will be necessary when it comes to traversing the parse tree to add further information.

6.2.3 OZTranslator

The purpose of the *OZTranslator* is to gather information about the parse tree and transfer it to the place where it will later be used to produce the output in the Perfect language. The *OZTranslator* breaks these transformation steps down into smaller parts and, therefore, dedicated classes fulfill only a limited purpose each, such as to collect or combine additional semantic information, and, step by step, the single implementations are then chained to provide the translation functionality as a whole. Figure 6.3 shows an overview of the main classes and data structures built by them to do the final translation to Perfect.

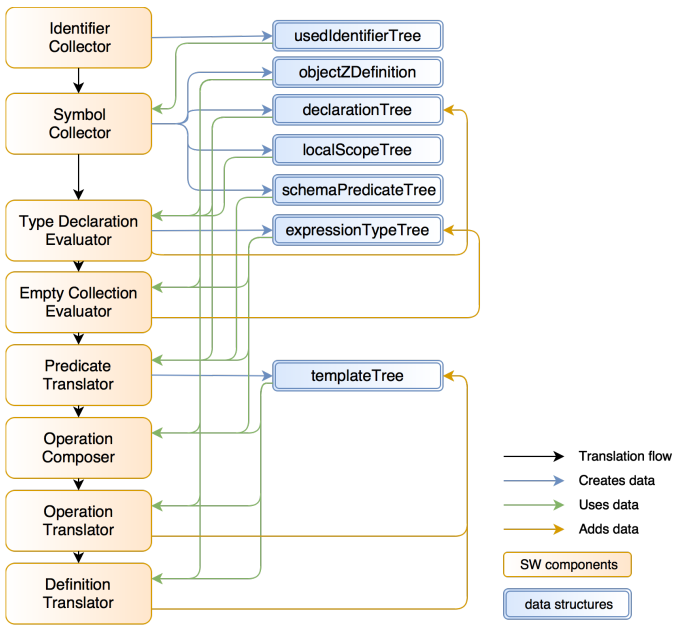


Figure 6.3: Overview of main classes for transformation and the internal data structures

Some, but not all of them are implemented as extension to the *OZBaseListener* class, which provides two methods for each parse tree node. Listing 6.2 shows these methods for parse rule *classDefinition*; `enterClassDefinition` is called by the tree walker before entering the node, and `exitClassDefinition` after exiting the node.

```
@Override
public void enterClassDefinition(ClassDefinitionContext ctx) {
    // actions executed before the rule go here
}
@Override
public void exitClassDefinition(ClassDefinitionContext ctx) {
    // actions executed after the rule go here
}
```

Listing 6.2: Listener methods for parse rule classDefinition

After exiting, all the semantical information from all the visited subnodes has already been collected and so it can be aggregated even further. By implementing interface `OZListener` or extending the default implementation `OZBaseListener`, which enables overriding only selected methods, custom behavior can be provided for walking the tree. When then `ParseTreeWalker` walks through a parse tree, it starts at the root node and then descends down to each child node until it finally returns to the root node again. For the listener implementations, this means, that one can assume that the enter and exit methods of rules in the parse tree provided in Figure 6.2 are called in the order given in Listing 6.3.

```
enterOperationSchemaDef()
enterDeltalist()
exitDeltalist()
enterDeclarationList()
enterDeclaration()
...
exitDeclaration()
exitDeclarationList()
enterPredicateList()
....
exitPredicateList()
exitOperationSchemaDef()
```

Listing 6.3: Call sequence of enter and exit methods while traversing the parse tree

The class `ParseTreeProperty` is another powerful feature of ANTLR that makes it possible to associate data with a parse tree node. For example, it can be used to save information of node *declaration* and this information can be accessed in the exit listener of the parent node *declarationList*. The subsequent paragraphs will provide descriptions of all components of *OZTranslator* shown in Figure 6.3.

IdentifierCollector

The idea of the `IdentifierCollector` is to have the identifiers that are used within a subtree available at any given node. It starts from nodes representing names with and without decoration and aggregates them in every single parent node. The result is saved in the ParseTreeProperty `usedIdentifierTree`. This information is e.g. needed to decide in the next step, whether a predicate is a pre- or a postcondition.

SymbolCollector

The `SymbolCollector` is responsible for declaring each symbol within its scope. Symbols are variables, classes, operations, or local definitions and the possible scopes comprise the whole Object-Z specification, a class, a schema, or a local scope, e.g. in set abstractions or quantification predicates. After this step an `ObjectZDefinition` object is available that already knows about the defined classes and local definitions. The referenced classes in return know about the state variables and the class invariants, operations, the visibility list, whether there is an INIT-function or not, and the names of inherited classes. However, it is important to notice that at this point, structures like variable declarations cannot yet be fully filled as the already collected information is not capable of e.g. determining the type of expressions. One could only say that the type is represented by a certain Object-Z expression, but the type is not yet evaluated.

Beside the `ObjectZDefinition` object, the `SymbolCollector` also outputs three `ParseTreeProperty` objects:

- The `declarationTree` assigns a variable declaration, a `Variable` object that will enable the access to all information of a variable, like its declared type or name, to the identifier node of the variable declaration.
- The `localScopeTree` holds the `LocalScope` objects that are created during this pass of the tree.
- The `schemaPredicateTree` is the container for `ISchemaPredicate` objects that wrap each predicate of the specification to link them back to the parse tree and also provide access to the identifiers used within the predicate.

TypeDeclarationEvaluator

The `TypeDeclarationEvaluator` aims at assigning a type to all expressions within the specification and also to each declared symbol. Simply said, if there is a variable declaration such as $s : \mathbb{P}\,\mathbb{N}$, then the information that s is a variable typed as a set of natural numbers should be available and accessible in the whole context where s is declared. Evaluation of expressions and assignment of the found types to the declared variables happen alternately, which can be easily seen in the previously shown *set* example. First, the evaluator determines the type of $\mathbb{N}$, then $\mathbb{P}\,\mathbb{N}$, then the declaration is recognized and the type of the expression $\mathbb{P}\,\mathbb{N}$ is assigned to the variable s. The next term in the specification could be another variable declaration and the evaluation is done again in the same manner.

The evaluator also determines the actual type if the expression is an identifier of e.g. a class or a basic type, which then evaluates to a user defined type. Predicates evaluate to a boolean type. In addition to the fact that the ParseTreeProperty `declarationTree` now has the necessary type information for declared variables, the ParseTreeProperty `expressionTypeTree` contains the type for all expressions and subparts of expressions in the specification. This information is needed to determine for some expressions the correct translation template based on the involved types.

EmptyCollectionEvaluator

Due to the bottom-up and left-to-right traversal of the tree, it is sometimes not possible to determine the type of a single expression part. For example, in the expression $s = \{\} \cup \{1, 2, 3\}$ it is possible to determine the types of s, $\{1, 2, 3\}$ and even the whole expression, but not the type of $\{\}$. However, the type information is mandatory for some translation rules. In the example, the `TypeDeclarationEvaluator` is only able to recognize $\{\}$ as a set of some type. In order to remain type compatible, the element types of the left and right hand side of the union have to be the same or compatible. Unfortunately, at the time the type of the left expression ($\{\}$) is evaluated, the element type of the right expression ($\{1, 2, 3\}$) is not yet evaluated. In such cases, a second traversal of these empty set, bag, or sequence nodes is helpful to determine the type of element nodes from neighboring nodes. The `EmptyCollectionEvaluator` accomplishes this task resulting in a modification of the affected nodes in the `expressionTypeTree`.

PredicateTranslator

The `PredicateTranslator` assigns a suitable template as mapping to each expression and predicate node. This class makes use of internal data structures `expressionType Tree`, `declarationTree`, and `schemaPredicateTree` for that purpose. The Parse-TreeProperty `expressionTypeTree` is used in this context to determine which mapping should be applied if the mapping depends on the type of the original expression. The `schemaPredicateTree` is used for the proper translation of predicates in quantification expressions. The ParseTreeProperty `templateTree` is initialized by this class and saves all created templates at the corresponding predicate or expression node. The input data structure `declarationTree` enables access to the variables and their types within the tree. In order to be able to build a Perfect declaration only from the information stored within a `Variable` object, the type template information is also handed over to these objects, which means that the information in `declarationTree` is enhanced by this class.

OperationComposer

In the symbol collection phase only the names of referenced operations have been collected for operation expressions, and predicates only reference back to the parse tree. To be able to translate operations to Perfect, it is necessary, to resolve these dependencies. So, the `OperationComposer` looks up the names of the referenced operations in the class of the calling object, retrieves the actual `Operation` object and makes the functions for input and output parameters, preconditions and postconditions available for the composition of the operation. The second duty is to build the preconditions and postconditions of all operations in all classes. This internal representation of an operation is saved directly within the `Operation` object. After this translation step has finished, this information is available for output generation.

OperationTranslator

To maintain the characteristic of Object-Z in referencing the combined operations, it became necessary, to have the information of all referenced operations available before actually doing the translation step. That is why this step is not yet directly incorporated into the previous class `OperationComposer`. The `OperationTranslator` starts from the `ObjectZDefinition` object and iterates over all operations, first those based on operation schema definitions, then those built from operation expressions. In each step, a template of the operation is built, which contains the representation of the operation itself, a function representing its precondition and, if needed, a `post`-function, that makes the value of output variables of schemas available before actually calling the schema and doing the modifications on internal data structures. The resulting StringTemplate with the concrete values of the operation is then saved within the operation and also in the `templateTree` data structure.

DefinitionTranslator

The last step in the translation process chain translates all the remaining symbols to Perfect. The `DefinitionTranslator` translates free type definitions, given types and abbreviation definitions on global level, first. Next, it iterates over all defined classes including their local definitions, and as a last step the single building blocks are combined to one StringTemplate representing the whole Object-Z definition. As in the previous step, the resulting StringTemplates are saved in the `templateTree` and additionally in the internal representations of these language constructs, making the template representation available using the method `getTemplate()`.

6.2.4 TemplateRenderer

Finally, the processed Object-Z specification is rendered as Perfect class definitions. After the Application of all the translation steps, the `getTemplate()` method of the `ObjectZDefinition` object delivers the StringTemplate object representation of the whole specification provided to the tool. It can be rendered into a string by means of the `render()` method, which prints the whole specification translated to Perfect. That is also the functionality available from the `Main`-class created within this project. However, it is also possible to access single classes programmatically and render only parts of the translated Perfect specification code.

6.3 Selected Topic: Types

After the overview of the main components of the translator, this and the following section focus on some aspects of the implementation, that turned out to be quite interesting during the implementation work, which was not anticipated before. The developed internal structures helped to provide the semantics of the specification more explicitly with regard to the needs of the translation to Perfect.

The first topic targets handling types properly, i.e. collecting the types and assigning them to the declared variables and expressions on the one hand and, on the other hand, having the correct information available for the translation. During the implementation of the mapping, it turned out that just looking for the declared type cannot be enough in particular cases. Considering the example in Figure 6.4, there is variable *cset* declared as a set of natural numbers, variable *c* declared as member of *cset* and a quantification predicate declaring variable *b* again as a member of *cset*. The last line defines *SmallNumber* as an abbreviation definition using set abstraction on the right hand side.

$$\begin{array}{l} \mathit{cSet} : \mathbb{P}\,\mathbb{N} \\ c : \mathit{cset} \\ \forall\, b : \mathit{cset} \bullet b < 100 \\ \mathit{SmallNumber} == b : \mathit{cSet} \bullet b < 100 \end{array}$$

Figure 6.4: Using collection typed variables in Object-Z

Object-Z allows variable declarations with a collection variable as type. However, in Perfect this is only possible for bound variable declarations and so this has to be modeled properly during translation. Listing 6.4 shows how these Object-Z declarations should be translated to Perfect. In lines 2 and 3 *c* is declared as a natural number. The invariant states that it also has to belong to *cSet*. In contrast in the quantification expression, the bound variable declaration directly references the name of the set. In the context of the abbreviation definition *SmallNumber*, *b* is again declared with a type and adds the constraint to the predicate section of the `those` clause.

```
1 var cSet: set of nat;
2 var c: nat;
3 invariant c in cSet;
4 forall b::cSet :- b < 100;
5 class SmallNumber ^= those b:nat :- b in cSet & b < 100;
```

Listing 6.4: Translating collection typed variable to Perfect

Obviously, it is not sufficient, to solely know the expression written on the right hand side of a declaration, especially if this is an identifier. To solve this problem, the types are resolved and then saved in a dedicated structure during the translation process. This is accomplished by two classes `ExpressionType` and `Type`. The first represents the whole type information available for any given expression and consists of two `Type` variables, `declaredType` and `effectiveType`. For the example in Figure 6.4, this means that variable *c* has a declared type with `Category.TEMPLATE` and an effective type with `Category.SET` which also references a single `ExpressionType` representing natural numbers as a subtype. The declared type always stands for what was originally written in Object-Z as a StringTemplate, while the effective type is used for type evaluations such as choosing the correct mapping rule for domain and range operations. It is possible to deduce all the declarations and usages in Listing 6.4 from this kind of data structure, but it is still necessary to react to the different possible types within the implementation. Figure 6.5 shows the relations between the two implemented type classes and lists the available values for the enumeration `Category` as an UML class diagram.

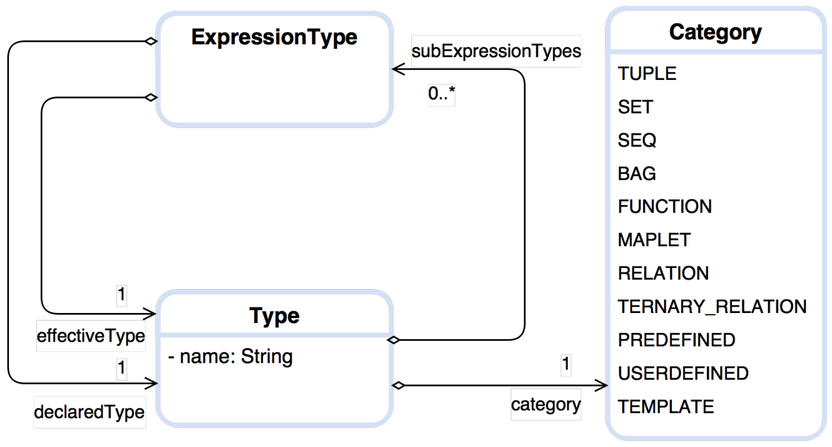


Figure 6.5: Relation between classes ExpressionType and Type as UML class diagram

6.4 Selected Topic: Combining Operation Expressions

Another interesting aspect of the implementation needs to be mentioned in more detail. This work presents a novel approach for an abstraction of the preconditions and postconditions involved in the combination of operation expressions. The goal of this approach was to make it possible that operations of the same types can be combined without nesting and therefore keeping the data structures as flat as possible.

Each composite operation is represented by its own class. Only parallel composition and associative parallel composition can be combined within one class. The classes representing postconditions have also been organized as a hierarchy. Figure 6.6 gives an overview of the classes used to compose operation expression postconditions.

It is eye-catching, that there are six classes extending the abstract class `Combined Postconditions` and only two other classes, `NonDeterministicPostconditions` and `SequentialPostconditions`. The latter two represent the postconditions for the corresponding operation promotions and they fulfill the goal that they can accommodate operation promotions such as $A \mathbin{[\!]} B \mathbin{[\!]} C$ or $A \fatsemi B \fatsemi C$ within only one postcondition object. The list members contain the representation of the sub operation expressions and in the case of sequential composition, there is one entry in the list of `communicationVariables` from each subpart of the operation expression to the next.

On the other side of the tree, there are different kinds of `CombinedPostconditions` that are used to represent postconditions of operation conjunctions and parallel compositions. `EmptyPostconditions` are used in bool functions or whenever there is no postcondition available to prevent combinations with null values. An `OutputPromotion` represents a single promotion of an output parameter of a function. A `ChangeOperation Call` wraps a single promotion of a call to an operation schema. The remaining operations are used to represent more complex situations including common output parameters

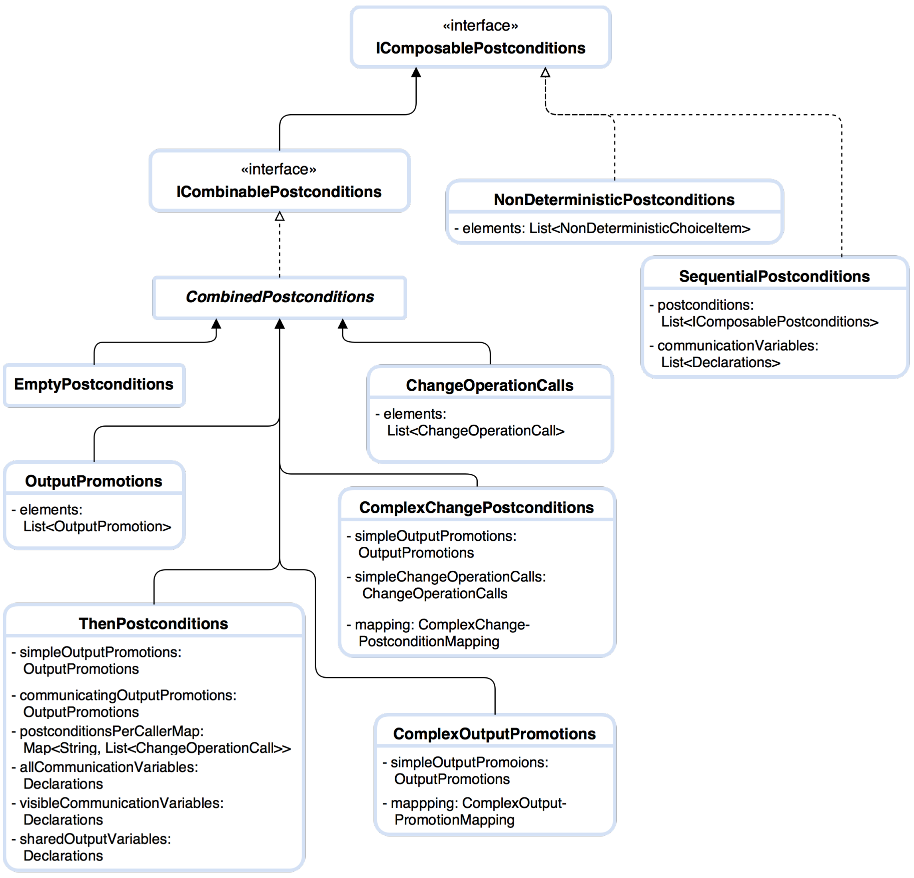

«interface»
IComposablePostconditions
«interface»
ICombinablePostconditions
NonDeterministicPostconditions
- elements: List<NonDeterministicChoiceItem>
SequentialPostconditions
- postconditions: List<IComposablePostconditions>
- communicationVariables: List<Declarations>
CombinedPostconditions
EmptyPostconditions
ChangeOperationCalls
- elements: List<ChangeOperationCall>
OutputPromotions
- elements: List<OutputPromotion>
ComplexChangePostconditions
- simpleOutputPromotions: OutputPromotions
- simpleChangeOperationCalls: ChangeOperationCalls
- mapping: ComplexChange-PostconditionMapping
ThenPostconditions
- simpleOutputPromotions: OutputPromotions
- communicatingOutputPromotions: OutputPromotions
- postconditionsPerCallerMap: Map<String, List<ChangeOperationCall>>
- allCommunicationVariables: Declarations
- visibleCommunicationVariables: Declarations
- sharedOutputVariables: Declarations
ComplexOutputPromotions
- simpleOutputPromoions: OutputPromotions
- mappping: ComplexOutput-PromotionMapping

Figure 6.6: Internal representation of postconditions of operation expressions

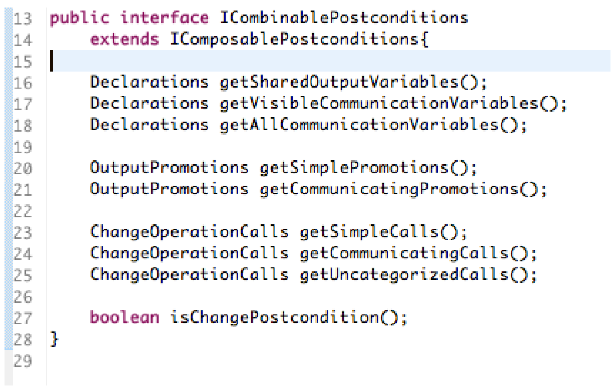
or communication from output to input parameters. `ComplexOutputPromotions` are used for postconditions which contain only operation promotions. Its variable “`mapping`” contains shared output variables and all the information necessary for communication. `ComplexChangePostconditions` are similar to the ones before, except that this class references `ChangeOperationCalls` as well as `OutputPromotions`. The last class, `ThenPostconditions`, represents all those postconditions that need sequentiality in the output, e.g. because they use common callers. The different data structures contain the data to record the dependencies between the called operations. The way how this structure is actually output is only calculated when the template for the operation is created in the `OperationTranslator` step.

`CombinablePostconditions` are conjoined or combined by the dedicated class `CompositePostconditionDataCollector` (see Listing 6.5). It retrieves the necessary data from the involved postconditions using the methods defined in interface `ICombinablePostconditions` which are shown in Figure 6.7.

```
public class PostconditionFactory implements ICompositionFactory {

    public ICombinablePostconditions compose(ICombinablePostconditions left,
       ICombinablePostconditions right, Declarations communicationVariables,
       Declarations sharedOutputVariables, boolean isAssociative){
        CompositePostconditionDataCollector data = new
            CompositePostconditionDataCollector(left, right);
        data.compose(communicationVariables, sharedOutputVariables, isAssociative);
        return new CompositePostconditionFactory(data).createPostcondition();
    }
    public ICombinablePostconditions conjoin(ICombinablePostconditions left,
        ICombinablePostconditions right, Declarations sharedOutputVariables){
        CompositePostconditionDataCollector data = new
            CompositePostconditionDataCollector(left, right);
        data.conjoin(sharedOutputVariables);
        return new CompositePostconditionFactory(data).createPostcondition();
    }
}
```

Listing 6.5: Usage of class CompositePostconditionDataCollector



```
public interface ICombinablePostconditions
    extends IComposablePostconditions{

    Declarations getSharedOutputVariables();
    Declarations getVisibleCommunicationVariables();
    Declarations getAllCommunicationVariables();

    OutputPromotions getSimplePromotions();
    OutputPromotions getCommunicatingPromotions();

    ChangeOperationCalls getSimpleCalls();
    ChangeOperationCalls getCommunicatingCalls();
    ChangeOperationCalls getUncategorizedCalls();

    boolean isChangePostcondition();
}
```

Figure 6.7: Interface ICombinablePostcondition and its methods

This way of building an internal representation of the operation expressions makes it possible to defer decisions how a subexpression should be mapped to Perfect to the latest possible moment, which can make the generated output result easier to read, and the way one can work with the data structure more flexible. For preconditions, a similar structure has been built, but the used concepts are the same, so they will not be described here in more detail.

This ends the chapter about how the tool for translating Object-Z to Perfect has been implemented. The following chapter discusses the results of this work on the basis of a case study.

CHAPTER 7

Results, Conclusion, and Outlook

In this chapter the implemented tool[1] is evaluated based on a case study providing exemplary Object-Z specifications. The tool output and the verification and validation results delivered by Perfect Developer are discussed, especially taking the differences to other mappings into account.

7.1 Case Study: Book and SmallLibrary

To demonstrate the implemented translation tool, this section presents an example that bases on the classes *Book* and *SmallLibrary* provided in Chapter 3 in Figures 3.8 and 3.9. The mapping of the first class focusses on the mapping of the class construct and its different kinds of operations. To keep the case study simple and concise, only a subset of the operations will be included in the mapping. On the other hand, the second class only includes three operation expressions that underline the main features of their translation rules, common output, communication, and sequentiality. The specifications in Listings 7.1 and 7.2 are written in OZ syntax, the input notation for the translator, instead of the usual Object-Z notation and these include only those parts, that will also be discussed in this case study.

```
class Book{
      visible(lend, latestRecension, authorList, setAuthorList)
      const{
            maxHistoryEntries: !N;
            maxHistoryEntries in {10, 20, 50, 100};
      }
      state{
        authors, recensionHistory, readerHistory: seq String;
        lent: !B;
        delta {
```

[1]Object-Z to Perfect Project, https://github.com/sylviaswoboda/objectz-2-perfect

```
            totalLendingCount: !N;
        }
        #recensionHistory <= #readerHistory;
        #recensionHistory <= maxHistoryEntries;
        totalLendingCount = #readerHistory;
        authors ~= [];
      }
    INIT{
        recensionHistory = [];
        readerHistory = [];
        ~lent;
    }
    authorList {
        authors!: seq String;
        authors! = authors;
    }
    setAuthorList {
        delta(authors)
        authors?: seq String;
        authors' = authors?;
    }
    latestRecension{
        recension!: String;
        #recensionHistory > 0;
        recension! = last recensionHistory;
    }
    lend{
        delta(readerHistory, lent)
        reader?:String;
        ~lent;
        lent';
        readerHistory' = readerHistory +^+ [reader?];
    }
}
```

Listing 7.1: Book example in OZ

```
class SmallLibrary {
    visible(postRecension, authorLists, transferAuthorsAndRecension,
        switchAuthorsAndLend)
    state {
        book1, book2, book3: Book;
        recensionsWall: Book <--> String;
        book1 ~= book2;
        book1 ~= book3;
        book2 ~= book3;
    }
    postRecension{
        delta(recensionsWall)
        book?: Book;
        recension?: String;
        recensionsWall' = recensionsWall >O< {book? |-> recension?};
    }

    authorLists ^= book1.authorList && book2.authorList;
    transferAuthorsAndRecension ^= (book1.authorList ||! book2.setAuthorList &&
        book2.latestRecension) 0/9 book3.latestRecension;
    switchAuthorsAndLend ^= ((book1.authorList || book2.setAuthorList) && book2.lend)
        [] ((book2.authorList || book1.setAuthorList) && book2.lend)
}
```

Listing 7.2: SmallLibrary example in OZ

7.1.1 Evaluating the Tool implementation

Applying the translation tool to classes *Book* and *SmallLibrary* maps the OZ input to two Perfect classes. The results of this process are presented and analyzed step by step starting with the abstract section of class *Book* in Listing 7.3. All the primary variables of the state schema of the Object-Z specification and constants of the axiomatic definition are declared here as abstract data members. The secondary variable `totalLendingCount` is not included, just as expected according to the mapping rules. It is notable that each variable is declared in a separate line here, as opposed to the Object-Z specification, which declared several names within one declaration statement. This happens, because internally each variable is represented separately. However, both types of declarations are semantically equivalent. The last four lines in the listing contain the mappings of the invariants presented in the state schema and the axiomatic definition. In the invariants concerning *authors* and *maxHistoryEntries* the types of the collections have been correctly recognized by the translator.

```
class Book ^=
abstract
    var authors: seq of String;
    var recensionHistory: seq of String;
    var readerHistory: seq of String;
    var lent: bool;
    var maxHistoryEntries: nat;
    invariant #recensionHistory <= #readerHistory;
    invariant #recensionHistory <= maxHistoryEntries;
    invariant authors ~= seq of String {};
    invariant maxHistoryEntries in set of nat {10, 20, 50, 100};
```

Listing 7.3: Abstract section of Book example translated to Perfect

The confined section of the mapped example is expected to consist of all remaining members, that have not been included in the visibility list. In the Book example, these are the *INIT* function, the getters and setters for all primary variables and the getter function for the only declared secondary variable. The result is shown in Listing 7.4.

```
confined
    function INIT: bool
    ^= recensionHistory = seq of String {} &
       readerHistory = seq of String {} &
       ~ lent;
    function authors;
    function recensionHistory;
    function readerHistory;
    function lent;
    function maxHistoryEntries;

    schema !set_authors(authors_in:seq of String)
        pre  authors_in ~= seq of String {}
        post authors! = authors_in;

    schema !set_recensionHistory(recensionHistory_in:seq of String)
        pre  #recensionHistory_in <= #readerHistory,
            #recensionHistory_in <= maxHistoryEntries
        post recensionHistory! = recensionHistory_in;

    schema !set_readerHistory(readerHistory_in:seq of String)
        pre  #recensionHistory <= #readerHistory_in
        post readerHistory! = readerHistory_in;

    schema !set_lent(lent_in:bool)
        post lent! = lent_in;

    function totalLendingCount:nat
        satisfy result = #readerHistory;
```

Listing 7.4: Confined section of the Book example translated to Perfect

Listing 7.5 demonstrates the build function that is used to construct new objects of the class. It is always situated within the interface part of the Perfect specification and sets all primary variables and constants to their initial values while maintaining all invariants (lines 1-4 of the precondition section) and conditions specified by the *INIT* function (lines 5-7).

```
interface
    build{!authors:seq of String, !recensionHistory:seq of String, !readerHistory:seq
        of String, !lent:bool, !maxHistoryEntries:nat}
        pre #recensionHistory <= #readerHistory,
            #recensionHistory <= maxHistoryEntries,
            authors ~= seq of String {},
            maxHistoryEntries in set of nat {10, 20, 50, 100},
            recensionHistory = seq of String {},
            readerHistory = seq of String {},
            ~ lent;
```

Listing 7.5: Constructor of Book example translated to Perfect

Listing 7.6 presents the translations of all the operations of class *Book*. For each operation schema, the correct type has been chosen for the mapping, function or schema, depending on whether a deltalist has been included in the operation schema, or not. The mappings of functions and schemas are split into two parts. First, a boolean helper function, defining the preconditions of the actual operation, is declared. Second, the function or schema, that provides the output parameters and state changes, is added. The precondition function is called in the precondition section of the function or schema translation. It contains the predicates of the operation schema that have been classified as preconditions, and additionally, an existential quantification stating that this operation can only be used, when a combination of suitable output parameters and values for primary state variables to be changed can be found that satisfies all the postconditions.

As these are automatically generated conditions with no semantic checks applied, they seem fairly obvious for operations *authorList*, *setAuthorList*, and *latestRecension*, but in operation *lend* the quantification clearly states, that this operation may only be applied, when a new *reader?* can really be added to the *readerHistory*. This is to reflect, that

Object-Z operations that do not satisfy this implicit condition are not applicable at all. For example, the number of readers could be restricted to a maximum by an invariant.

```
    function authorList_prec : bool
        ^= (exists authors_temp:seq of String :- (authors_temp = authors));
    function authorList authors_out:seq of String
        pre authorList_prec
        satisfy result.authors_out = authors;

    function setAuthorList_prec (authors_in:seq of String): bool
        ^= (exists authors_temp:seq of String :- (authors_temp = authors_in));
    schema !setAuthorList (authors_in:seq of String)
        pre setAuthorList_prec(authors_in)
        post change  authors
             satisfy authors' = authors_in;

    function latestRecension_prec : bool
        ^= #recensionHistory > 0 &
           (exists recension_temp:String :- (recension_temp = recensionHistory.last));
    function latestRecension recension_out:String
        pre latestRecension_prec
        satisfy result.recension_out = recensionHistory.last;

    function lend_prec (reader_in:String): bool
        ^= ~ lent &
           (exists readerHistory_temp:seq of String, lent_temp:bool
              :- (lent_temp & readerHistory_temp = readerHistory ++
                  seq of String {reader_in}));
    schema !lend (reader_in:String)
        pre lend_prec(reader_in)
        post change  readerHistory, lent
             satisfy lent' &
                     readerHistory' = readerHistory ++ seq of String {reader_in};
```

Listing 7.6: Interface methods of Book example translated to Perfect

It is already noticeable, that compared to the OZ input, the generated Perfect output is quite long. This is mainly caused by the need to express all the implicit conditions of Object-Z explicitly to make them available to Perfect Developer and, secondly, due to the decision in the presented implementation, to make preconditions available via a precondition function.

The operation `postRecension` in Listing 7.7 demonstrates the translation of expressions with a mapping that distinguishes between types by usage of the *overriding* expression. The translator has correctly chosen the mapping to be applied for relations, as described in Table 5.6.

```
    function postRecension_prec (book_in:Book, recension_in:String): bool
        ^= (exists recensionsWall_temp:set of pair of (Book, String)
            :- (recensionsWall_temp = (those tempVar13::recensionsWall
                  :- (exists tempVar14::set of pair of (Book, String) {pair of (Book,
                      String) {book_in, recension_in}}
                      :- tempVar13.x = tempVar14.x))
                ++ set of pair of (Book, String) {pair of (Book, String) {book_in,
                    recension_in}}));
    schema !postRecension (book_in:Book, recension_in:String)
        pre postRecension_prec(book_in, recension_in)
        post change  recensionsWall
             satisfy recensionsWall' = (those tempVar13::recensionsWall
                        :- (exists tempVar14::set of pair of (Book, String) {pair of (
                            Book, String) {book_in, recension_in}}
                            :- tempVar13.x = tempVar14.x))
                     ++ set of pair of (Book, String) {pair of (Book, String) {book_in,
                          recension_in}};
```

Listing 7.7: Overriding in SmallLibrary example translated to Perfect

Listings 7.8, 7.9, 7.10 show different kinds of operation expressions involving two or more operation calls. In the first one, the *authorLists* operation is included as a representative for operation conjunction. It combines the operation *authorList* called on two *Book* objects, which means that the common output parameter *authors* has to be handled properly here. As described in the mapping, an existential quantification is added in the precondition to make sure that the operation can only be applied, when the values of the output parameter *authors* are equal. In the postcondition, only one promotion of the output parameter is included as expected.

```
    function authorLists_prec : bool
        ^= book1.authorList_prec &
           book2.authorList_prec &
           (exists tempVar25:seq of String
              :- (tempVar25 = book1.authorList.authors_out &
                  tempVar25 = book2.authorList.authors_out));
    function authorLists authors_out:seq of String
        pre authorLists_prec
        satisfy result.authors_out = book1.authorList.authors_out;
```

Listing 7.8: Example for operation conjunction with common output translated to Perfect

Listing 7.9 provides an example for the mapping of associative parallel composition, conjunction and sequential composition. It includes communication by means of variable *authors* and shows, that the value of the output parameter *recension!* is not constrained, which conforms to the semantics, as the operation first sets it to one value and afterwards to another. The first part of the postcondition is the promotion of *recension*, which means, that when applying conjunctions or parallel compositions, the translation tool may change the order in which the postconditions are output while maintaining the semantics of the combined operation.

```
    function transferAuthorsAndRecension_prec : bool
        ^= book1.authorList_prec &
           book2.latestRecension_prec &
           (exists tempVar27:seq of String
              :- (tempVar27 = book1.authorList.authors_out &
                   book2.setAuthorList_prec(tempVar27))) &
           book3.latestRecension_prec;
```

```
schema !transferAuthorsAndRecension (authors_out!:out seq of String,
        recension_out!:out String)
    pre transferAuthorsAndRecension_prec
    post recension_out! = book2.latestRecension.recension_out &
         (var tempVar26:seq of String; (tempVar26! = book1.authorList.authors_out
            & book2!setAuthorList(tempVar26') & authors_out! = tempVar26')) then
         recension_out! = book3.latestRecension.recension_out;
```

Listing 7.9: Example of parallel and sequential compositions translated to Perfect

Finally, a more complex example for the mapping of operation expressions to Perfect is shown in Listing 7.10. It combines nondeterministic choice, parallel composition and operation conjunction involving communication variables and even shows the modification of one object, *book2*, in parallel, in the left hand side of nondeterministic choice. On the right hand side, *book1* and *book2* are modified, so it is not necessary to consider concurrent changes, here.

```
function switchAuthorsAndLend_prec (reader_in:String): bool
    ^= book1.authorList_prec & book2.lend_prec(reader_in) &
       (exists tempVar32:seq of String
           :- (tempVar32 = book1.authorList.authors_out &
               book2.setAuthorList_prec(tempVar32))) |
       book2.authorList_prec & book2.lend_prec(reader_in) &
       (exists tempVar33:seq of String
           :- (tempVar33 = book2.authorList.authors_out &
               book1.setAuthorList_prec(tempVar33)));
opaque schema !switchAuthorsAndLend (reader_in:String)
    pre switchAuthorsAndLend_prec(reader_in)
    post ([book1.authorList_prec & book2.lend_prec(reader_in) &
           (exists tempVar28:seq of String
              :- (tempVar28 = book1.authorList.authors_out &
                  book2.setAuthorList_prec(tempVar28)))]:
              (var tempVar29:seq of String;
                 ((tempVar29! = book1.authorList.authors_out &
                   book2!lend(reader_in)) then book2!setAuthorList(tempVar29))),
          [book2.authorList_prec & book2.lend_prec(reader_in) &
           (exists tempVar30:seq of String
              :- (tempVar30 = book2.authorList.authors_out &
                  book1.setAuthorList_prec(tempVar30)))]:
              book2!lend(reader_in) &
              (var tempVar31:seq of String;
                 (tempVar31! = book2.authorList.authors_out &
                  book1!setAuthorList(tempVar31'))));
```

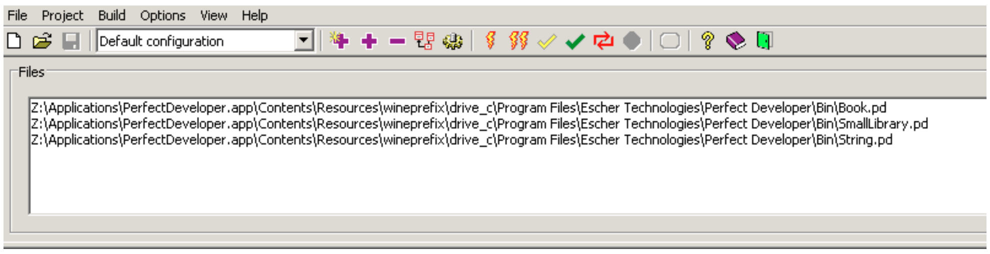
Listing 7.10: Example combining several operation promotions mapped to Perfect

During the translation process, the tool has recognized that one object is modified twice, but the deltalists of the modifying operations do not intersect and *lend*, first, *setAuthorList*, second, is an appropriate ordering, meaning the precondition of *setAuthorList* does not use the member variables that are changed by *lend*. It is interesting to notice that both preconditions are mapped in the same way, combining the precondition functions and adding a quantification expression for the communication variable *authors*, whereas in the postcondition, there is a difference in the mapping. For the left hand side, *book2.setAuthorList* is called using the `then`-postcondition form in Perfect after *book1.authorList* and *book2.lend* have been called in parallel. In contrast, on the right hand side, all three operations can be conjoined using `&` and they have even changed their order completely.

This shows, that the operations are not mapped by recursively applying the binary operations, but by keeping an internal structure that reflects the applied operations and makes it possible to combine more and more of the operations that are considered to be schema conjunctions (operation conjunction, parallel composition, associative parallel composition) while keeping the mapping in a flatter structure. This example also shows, that using precondition functions pays off when preconditions of other operations are referenced several times within the whole specification. It is not necessary to know more about the internals of the precondition function, when calling it from within another precondition or postcondition. The whole specification will be shorter, even more self-descriptive and the relations between operations remain visible to the specifier.

7.1.2 Evaluating the Verification Process in Perfect Developer

After evaluating the implementation, this section discusses how Perfect Developer (version 4.10.2 for windows) processes the produced output. The output of the case study has been added to a Perfect Developer project. Figure 7.1 shows a screenshot of the project files. File *String.pd* contains the mapping of the basic type *String* that has not been discussed in the evaluation earlier as the mapping resulted in a simple class named *String*, exactly as expected from the mapping rules and it was therefore not remarkable enough to be presented in more detail in this work.

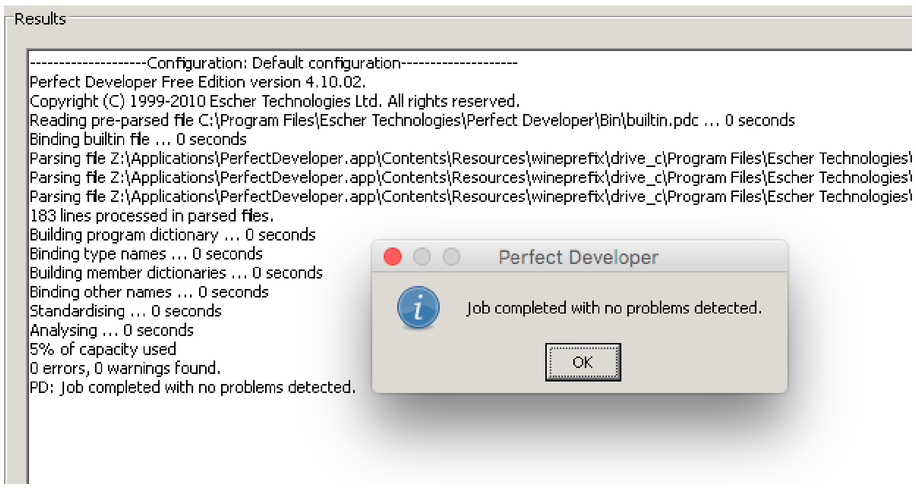


```
Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program Files\Escher Technologies\Perfect Developer\Bin\Book.pd
Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program Files\Escher Technologies\Perfect Developer\Bin\SmallLibrary.pd
Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program Files\Escher Technologies\Perfect Developer\Bin\String.pd
```

Figure 7.1: View of the Book and SmallLibrary project in Perfect Developer

The first step, is to check, whether the code translated from Object-Z has been mapped correctly regarding syntax and typing. The icon below the menu bar with a yellow tick on it, is used to perform this check. Perfect Developer parses and analyses the input and then signals the successful check of the project using a popup, as shown in Figure 7.2.

Second, the Perfect specification is verified by the Perfect Developer verification engine. A total of 38 verification conditions have been created for class *Book* and 43 for class *SmallLibrary*. Perfect Developer indicates that at least one of the verification conditions could not be proved by showing a popup with a warning.



```
--------------------Configuration: Default configuration--------------------
Perfect Developer Free Edition version 4.10.02.
Copyright (C) 1999-2010 Escher Technologies Ltd. All rights reserved.
Reading pre-parsed file C:\Program Files\Escher Technologies\Perfect Developer\Bin\builtin.pdc ... 0 seconds
Binding builtin file ... 0 seconds
Parsing file Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program Files\Escher Technologies\
Parsing file Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program Files\Escher Technologies\
Parsing file Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program Files\Escher Technologies\
183 lines processed in parsed files.
Building program dictionary ... 0 seconds
Binding type names ... 0 seconds
Building member dictionaries ... 0 seconds
Binding other names ... 0 seconds
Standardising ... 0 seconds
Analysing ... 0 seconds
5% of capacity used
0 errors, 0 warnings found.
PD: Job completed with no problems detected.
```

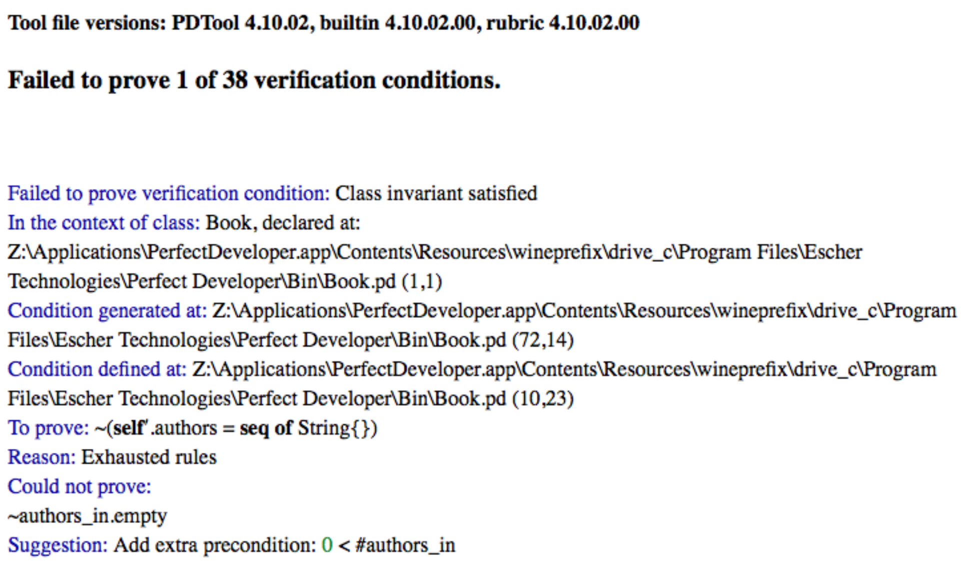
Job completed with no problems detected.

Figure 7.2: Project successfully checked for syntax and types in Perfect Developer

The goals of the generated verification conditions for class *Book* are to check whether type constraints, class invariants, or preconditions of predefined operations are satisfied. Perfect Developer creates two files, *Book_proof.htm* and *Book_unproven.htm* with more details about successful and non-successful verification attempts. Only one condition could not be proven in this example and the provided output is shown in Figure 7.3. It has been generated for operation *setAuthors* to check whether the class invariants remain satisfied while applying this operation. In the "*To prove:*"-line the output indicates that it tries to prove the invariant that variable *authors* could not end up with the value of an empty sequence. The necessary precondition that *authors_in* must not be empty is missing. In fact, Perfect Developer has just found a problem in the specification and even provides a suggestion how to fix it. Therefore, it is necessary to add the suggested precondition to the Object-Z specification, otherwise the specified system could end up in an unwanted state.

For class *SmallLibrary*, Perfect Developer is able to prove 39 out of the 43 generated verification conditions. Additionally to the types of conditions for *Book*, one condition is generated to check whether at least one guard is true for the choice postcondition, and a lot of conditions are generated to prove that the preconditions of the called operations of class *Book* are satisfied in the context of the three combined operations. An overview of the output provided by Perfect Developer of proven and unproven verification conditions for *SmallLibrary* is shown in Figure 7.4. Three verification conditions could not be proven, and one was neither proven nor disproved within the internal time limit.

All of them target the verification of the set of invariants that any two books of *b1*, *b2*, or *b3* must not be equal. In fact, in both operations, *transferAuthorsAndRecension*, *switchAuthorsAndLend* the value of variable *authors* is set to the value of another book. As Perfect uses value semantics and the equality for these classes is achieved when all



```
Tool file versions: PDTool 4.10.02, builtin 4.10.02.00, rubric 4.10.02.00

Failed to prove 1 of 38 verification conditions.


Failed to prove verification condition: Class invariant satisfied
In the context of class: Book, declared at:
Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program Files\Escher
Technologies\Perfect Developer\Bin\Book.pd (1,1)
Condition generated at: Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program
Files\Escher Technologies\Perfect Developer\Bin\Book.pd (72,14)
Condition defined at: Z:\Applications\PerfectDeveloper.app\Contents\Resources\wineprefix\drive_c\Program
Files\Escher Technologies\Perfect Developer\Bin\Book.pd (10,23)
To prove: ~(self'.authors = seq of String{})
Reason: Exhausted rules
Could not prove:
~authors_in.empty
Suggestion: Add extra precondition: 0 < #authors_in
```

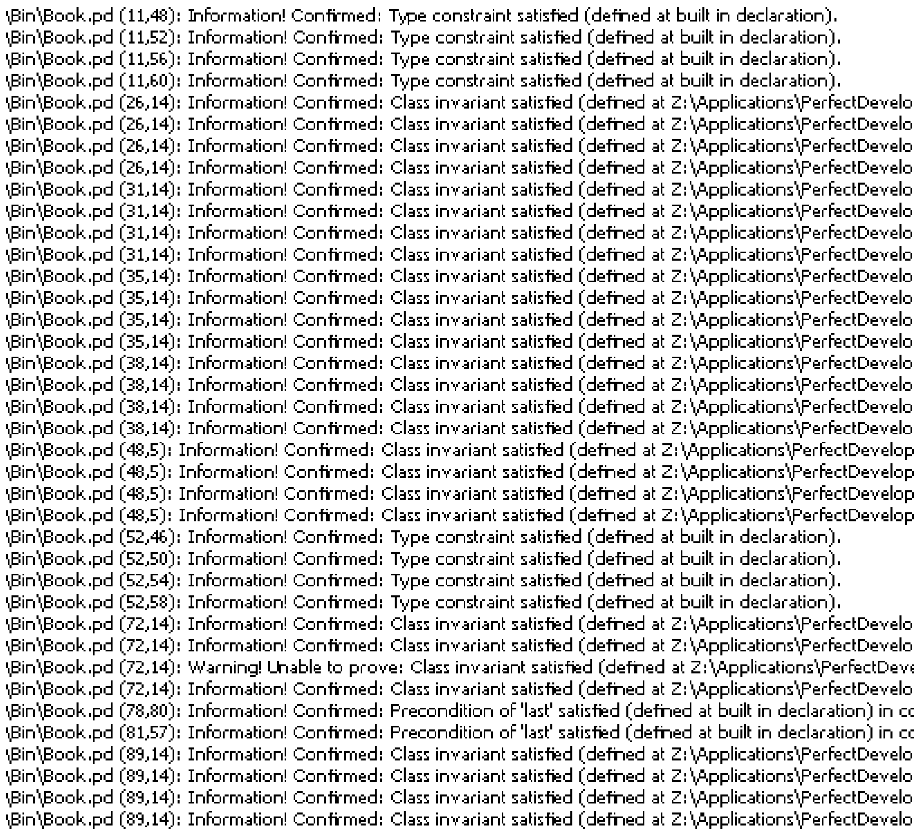
Figure 7.3: Unproven verification condition in example Book

member variables are equal, this also means, that if the value of one member is set to the value of the other object, at least one of the other member variables has to be different. The operations are written in a way that this cannot be asserted from the specification, so the proof fails. In Object-Z reference semantics is assumed, so there would not be the need to make sure objects remain separate. This inaccuracy has already been mentioned when providing the mapping rules for equality. Adding a unique identifier during the translation process, could help for these false-positive-cases, but finding and evaluating an appropriate solution for this problem, is left for future work.

7.2 Conclusion

This work shows how a mapping of Object-Z to Perfect can be achieved for classes with operation schemas and even more complex operation constructs like distributed operations. Also for many of the more expressive constructs of the Z notation such as overriding, set abstraction, or domain and range restrictions, which are likely to be used within pre- or postconditions of operations, mappings could be presented in this work. They cover a large amount of possible usage contexts and even involve different translation rules depending of the types of the involved expressions. The implemented tool, together with a whole lot of unit tests covering the implemented functionality and documenting the mapped features is available from the public repository on GitHub[2].

[2]Object-Z to Perfect Project, `https://github.com/sylviaswoboda/objectz-2-perfect`



```
\Bin\Book.pd (11,48): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
\Bin\Book.pd (11,52): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
\Bin\Book.pd (11,56): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
\Bin\Book.pd (11,60): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
\Bin\Book.pd (26,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (26,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (26,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (26,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (31,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (31,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (31,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (31,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (35,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (35,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (35,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (35,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (38,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (38,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (38,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (38,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (48,5): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelop
\Bin\Book.pd (48,5): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelop
\Bin\Book.pd (48,5): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelop
\Bin\Book.pd (48,5): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelop
\Bin\Book.pd (52,46): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
\Bin\Book.pd (52,50): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
\Bin\Book.pd (52,54): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
\Bin\Book.pd (52,58): Information! Confirmed: Type constraint satisfied (defined at built in declaration).
\Bin\Book.pd (72,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (72,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (72,14): Warning! Unable to prove: Class invariant satisfied (defined at Z:\Applications\PerfectDeve
\Bin\Book.pd (72,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (78,80): Information! Confirmed: Precondition of 'last' satisfied (defined at built in declaration) in cc
\Bin\Book.pd (81,57): Information! Confirmed: Precondition of 'last' satisfied (defined at built in declaration) in cc
\Bin\Book.pd (89,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (89,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (89,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
\Bin\Book.pd (89,14): Information! Confirmed: Class invariant satisfied (defined at Z:\Applications\PerfectDevelo
```

Figure 7.4: Overview of all verification conditions for class SmallLibrary

7.2.1 Limitations of the Implementation

Some aspects that have not yet been covered by the implementation, although a mapping has been presented in this work, are listed here:

- Parallel composition with common communication variables: If three operations are combined by parallel composition ($a.op1 \mid\mid b.op2 \mid\mid c.op3$), then the names of communication variables from op1 to op2 and op2 to op3 must not intersect, otherwise a wrong constraint would be added as precondition.
- Combining any number of operation expressions in any composition order is not yet implemented for all possible variations, especially nondeterministic choice and scope enrichment only provide a basic mapping.
- Distributed operations

- Inheritance: INIT-functions are not yet called by the INIT-function of the derived class. The INIT-function of a derived class is always defined with `redefine` keyword, even if there is no INIT function. Checking whether an identifier is a state variable of a super class is missing.
- Template generation of set abstraction
- Template generation of summation
- Template generation of abbreviation definition
- Declaration of strict naturals is currently always included
- Adding invariants implied by function or set, sequence, and bag definitions are not yet added to the output.

7.2.2 Discussion of Differences to Other Mappings

The findings of this work are finally compared with the proposed transformations presented in the works of Tim Kimber [Kim07] and Brian Stevens [Ste06] pointing out different approaches.

Brian Stevens showed how to exemplarily map an Object-Z class to a Perfect class by informally describing how the mapping shall be achieved manually. In his work, there is no detailed discussion on preconditions in operation promotions or on more complex expressions. So an automated mapping could not have been made, based solely on this work.

Tim Kimber's work, on the other hand, is much more comprehensive, but still leaves some aspects open for further work. This thesis has tackled most of them and has presented alternative mapping approaches for several other language constructs. The following list shows aspects of the Object-Z notation that were missing in Kimber's work and are novel in this work.

- Sets: Additional operations of the Z language have been mapped here, like minimum, maximum, or generalized union and intersection.
- Sequences: Additional operations on sequences in Z such as: reversing, head, last, front, tail of sequence, extraction, filtering, prefix, suffix, segment, and distributed concatenation
- Bags
- Genericity
- Inheritance: This work provides a basic mapping supporting single inheritance without method overriding.

The following paragraphs cover aspects of the mapping presented in this work, compared to the solutions presented in [Kim07].

Sets

The mapping presented in this work, covers the idea of the range operator on sets, $a..b$, that for values a > b the expression evaluates to an empty set, which can be easily mapped using the Perfect choice operator.

Set Abstraction and Summation

Additional rules have been included in this work for use with collections and transformations in set abstraction expressions and summations. However, these translation rules have not yet been implemented.

Bound Variable Declarations

Kimber's work distinguishes for bound variable declarations in quantification expressions, such as $\forall\, \mathit{declarations} \mid \mathit{predicates} \bullet \mathit{predicate}$, between variable declarations of types and collections but leaves the case when *predicates* are provided unmapped. This work extends the mapping of bound variable declarations to support *predicates* restricting the declared variables, and thoroughly discusses how a mapping for several declared variables can be achieved with a variable ordering found by the presented algorithm.

n-ary Relations

Tim Kimber defines the mapping of ternary relations differently in [Kim07], i.e. `set of pair of (pair of (X, Y), Z))`. Using this method an arbitrary number n of n-ary relations can be mapped. However, considering a simple example like a relation $R : \mathit{Name} \leftrightarrow \mathit{Sport} \leftrightarrow \mathit{Place}$ reflecting sentences such as, "Sylvia plays tennis at the playground", the idea is to find a mapping that keeps a flatter structure and improves readability. In Object-Z, the sentence is expressed as $(\mathit{Sylvia}, \mathit{tennis}, \mathit{playground}) \in R$. Listing 7.11 shows both mapping approaches applied to the same example.

```
// Tim Kimber's approach
set of (pair of (pair of (Name, Sport), Place)) \\
pair of (pair of (Name, Sport), Place) {pair of (Name, Sport){Sylvia, tennis},
    playground}

// approach of this work
set of (triple of (Name, Sport, Place))
triple of (Name, Sport, Place){Sylvia, tennis, playground}
```

Listing 7.11: Comparison of the mapping of n-ary relations

Operations on Relations and Functions

Beside covering overriding, domain restriction, and anti-restriction in addition to the domain and range operation, this work also distinguishes between functions and relations in the mapping of these operations and the access of elements of these. This is necessary to provide appropriate translation rules as functions and relations map to different constructs in Perfect, as well.

Free Type Definition

The mapping presented by Tim Kimber [Kim07], `b.color = blue@Color`, has been updated to the notation given in Table 5.25 to support the newer version on how to refer to enum types in Perfect according to the Perfect Developer language manual [Esc]. One further remark has to be made: In Z and Object-Z the free type definition branches can also be expanded by an expression which further describes the value. The mapping of this construct has been left for future work.

Visibilitylist

In Tim Kimber's mapping all features are made visible which does not represent the visibility specified by Object-Z. It is done there to make the features available for the use in preconditions of operations. This mapping uses a completely different approach and encapsulates the semantics of a precondition within a precondition function. So, features can be put into the appropriate areas in the Perfect specification while still making preconditions available within other classes. The same concept applies to the init schema. By wrapping it into a separate function and putting it into confined or interface section, the visibility properties are maintained supporting information hiding.

Operation Schemas

Non-modifying operations schemas, those returning a boolean value and those that only have output variables, have been mapped to functions in this work, while Tim Kimber uses functions only for the first type and non-modifying and modifying schemas for the latter two types.

Regarding the selection of preconditions and postconditions from the predicates provided by the Object-Z specification, this work makes use of a simpler approach, i.e. predicates are only split line by line and every predicate that uses a primed state variable or an output variable is considered a postcondition, the remaining ones are preconditions, but an additional precondition is added to reflect, that values for output and state variables have to be available to fulfill these conditions.

Operation Expressions

The ideas in mapping concepts like common output variables or communication variables have been solved similarly by making use of existential quantifications to assure the expected conditions are satisfied. However, in this work, the presented mappings target on going even further into providing a transformation for more cases. For that reason, the mapping rules and implementations make use of the inherent properties of operation expressions like commutativity of conjunction and parallel composition. Also, the use of helper functions like the precondition or "post" functions that make the values of output variables accessible in preconditions, enable for example the two-way-communication for parallel composition. The advantage of this approach is to encapsulate operations better and also keep the dependencies of operations visible in the mapped operation. However, the shortcoming is that a single operation can become quite long and, if not properly formatted, difficult to read.

7.3 Outlook and Future Work

Although, this work could provide a significant amount of mapped features of the Object-Z language, there are still a lot of possible improvements to the existing work.

- Complete open implementation topics listed in section 7.2.1.
- Improve the feedback in case of language recognition problems during parsing and translation phase.
- Improve error handling: Do not interrupt the translation process in case of errors, but only provide feedback to the user and proceed with the mapping as far as possible.
- Provide and extend translation rules for further language features: such as distributed sequential composition, multi-inheritance, renaming, scope enrichment for arbitrary operations on the left hand side, *let* definitions, λ, μ or other remaining Z expressions.
- Revisit equality in Object-Z and Perfect.
- Replacing the OZ input syntax by Object-Z notation in UTF-8 or LaTeX style, e.g. by instrumenting the core classes provided by the Community Z Tools or only adapt the lexical part of the provided grammar in this work.
- Facilitating the process of creating output files and handing them over to Perfect Developer or building a graphical user interface on top and handing the generated output files over to the Perfect verification engine without switching to another tool. The output is then again fed back to the user via the interface of the translation tool.
- Improve code formatting or add a means of reformatting the generated Perfect output such as code formatting templates of IDEs.

Providing a translation tool from Object-Z to Perfect not only builds the foundation to improve on the existing implementation, but could also be used as a starting point to translate back from Perfect again. The provided translation rules that mainly have to mimic the semantics of language constructs in Object-Z suggest that a mapping from Perfect to Object-Z could even be achieved more easily. Having a graphical notation to represent an implementation and even be able to switch between languages back and forth could be an interesting topic, when it comes to the specification and design phase. In the communication process to find or validate whether the specification represents the model of the real world properly, a visual representation is often more helpful.

APPENDIX A

Ordering Algorithm

In Chapter 5, in the mapping of quantification expressions, the need for an ordering algorithm arose to find an appropriate order of bound variable declarations in such an expression. This chapter presents one way of finding such an ordering and also gives an example to illustrate its function.

Starting from the original sequence of variables and a set of predicates in combination with a graph in which both variables and predicates are represented as nodes and each variable node is related to a predicate node if the variable appears in the named predicate, the algorithm informally described in Listing A.1 either delivers an appropriate ordering or aborts with an error message.

```
Input:
   varSeq   // Variable sequence in order of original appearance
   predSet  // Set of predicates, each predicate uses at least one variable
   G        // Graph representing the dependencies between variables and predicates,
            // V(G) .. vertices, E(G) .. edges, N(v) .. neighbours of v
Output:
   varOrder // list of variable and predicate combinations

1. Is there a variable-node v that has non-collection type?
   1a: Yes. varOrder' = varOrder ++ <v>;
       E(G)' = E(G) \ (v, w) for all w in predSet;
       V(G)' = V(G) \ {v}; Go to 1.
   1b: No, go to 2.
2. Is there a predicate-node p without neighbours?
   2a: Yes, error message and ABORT.
   2b: No, go to 3.
3. Is there a variable-node v without neighbours?
   3a: Yes. varOrder' = varOrder ++ <v>;
       varSeq' = varSeq \ {v}; V(G)' = V(G) \ {v};
       Go to 3.
   3b: No, go to 4.
4. Exists p in predSet with N(p) = {v}?
   4a: Yes, there is exactly one such p. Go to 5
   4b: Yes, there are more such p's.
       Choose p with (p,v) in G & v has smallest index in varSeq.
       Go to 5.
   4c: No, go to 6.
5. varOrder' = varOrder ++ <v>;
   varSeq' = varSeq \ {v}; predSet' = predSet \ {w} with (v, w) in G;
   E(G)' = E(G) \ (v, w) for (v, w) in G; V(G)' = V(G) \ {v};
6. Exists p in predSet with |N(p)| > 1?
   6a: Yes, there is exactly one/more than one such p.
       Choose v in varSeq with
       1. |N(v)| is minimal (for nodes v connected to p/one of the p's)
       2. v has smallest index in varSeq
7. varSeq' = varSeq \ <v>; predSet' = predSet \ {p} with |N(p)| = 0;
   E(G)' = E(G) \ (v, w) for (v, w) in G;
   V(G)' = V(G) \ {v} \ {p} with |N(p)| = 0;
8. (v, {p}) p from step 7; varOrder' = varOrder ++ <v>;
9. Is varSeq = <> ?
    8a: No, go to 4.
    8b: Yes, output varOrder and stop.
```

Listing A.1: Ordering algorithm

The sequence *varSeq* contains all variables declared within this quantification expression in the same order as in the Object-Z specification. The set *predSet* contains all predicates from the schema text of the quantification that depend on at least one of the variables in *varSeq*. The graph G represents the dependencies between variables of *varSeq* and predicates of *predSet*. Edges are only allowed between a variable- and a predicate-node. A connection between variable-node v and predicate-node p exists, if and only if the predicate of p uses variable v.

The output list *varOrder* contains an ordering of the variables together with the predicates that constrain the resulting `those`-clause such that the predicates use only the currently or any previously declared variables.

First all variables declared with non-collection type have to be added to the output list, their outgoing edges are removed from the graph and the variable itself is removed from *varSeq*. After this step, a check is performed if there are any predicates, that have no more dependencies now. This means that these predicates have depended on one of the previously removed nodes for non-collection type variables. As such variables cannot be restricted in Perfect by predicates in a `those`-clause, this results in an error message and abortion. Else, all variables that do not appear in any of the predicates are added to the output list as the next step.

Then the algorithm searches for predicate nodes that have the smallest number of outgoing edges. If there is only one, the adjacent variable-node with the smallest number of neighbors and second the smallest index in *varSeq* is chosen to be the next node in the output list. If there are more such predicate nodes, the next variable node v is chosen according to smaller number of neighbors or smaller index in sequence *varSeq* of all the variables connected to these predicate nodes.

All edges starting in v are removed from the graph, v is removed from *varSeq* and from the graph. All predicates that do not have adjacent edges are now paired with variable v and added to the output list. The predicates are also removed from *predSet* and the graph. Now the graph only contains edges for dependencies of variable nodes that have not yet been added to the output list. If no variables are left to be added, then the output list is returned and the algorithm stops, else the algorithm repeats searching the next predicate with smallest number of outgoing edges until no more variable nodes are left in the graph.

Example: To illustrate the algorithm at work, a small example is presented as follows. Variables $v1$ - $v4$ and $v6$ have collection type, $v5$ has some other type and a few predicates are added in the quantification that use one or more of the declared variables. The notation $p1(...)$ is an abstraction of the concrete predicate and only lists the variables it uses.

$$\begin{array}{l} \forall\, v1 : col1, v2 : col2, v3 : col3, v4 : col4, v5 : type5, v6 : col6 \mid \\ \quad p1(v1, v2, v3) \land p2(v1, v2, v3) \land p3(v1, v2, v3) \land \\ \quad p4(v2, v4) \land p5(v3) \land p6(v6) \bullet \\ \quad p(v1, v2, v3, v4, v5, v6) \end{array}$$

Now the algorithm is applied to the Object-Z example above. First *varSeq*, *predSet* and *graph* have to be initialized.

```
Input:
varSeq:  <v1, v2, v3, v4, v5, v6>
predSet:  {p1, p2, p3, p4, p5, p6}
graph:  <{v1, v2, v3, v4, v5, v6, p1, p2, p3, p4, p5, p6},
          {(p1, v1), (p1, v2), (p1, v3), (p2, v1),
           (p2, v2), (p2, v3), (p3, v1), (p3, v2),
           (p3, v3), (p4, v2), (p4, v4),
           (p5, v3), (p6, v6)}>
```

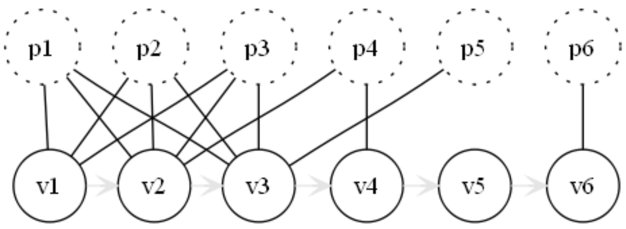
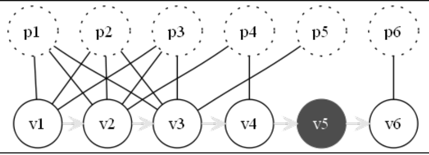
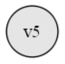
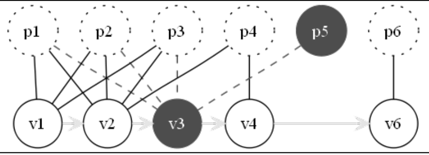
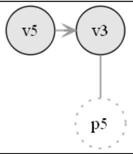
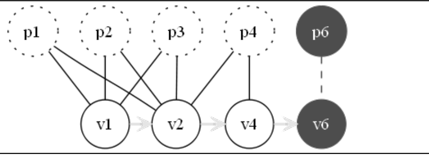
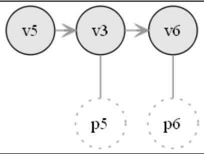
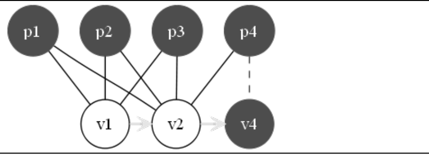
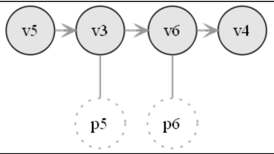
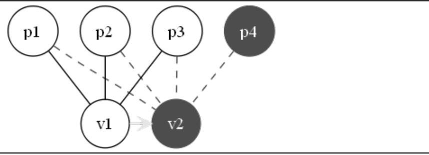
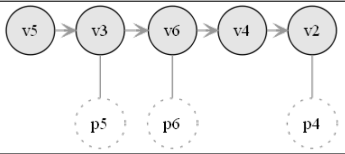
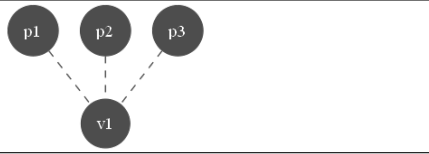
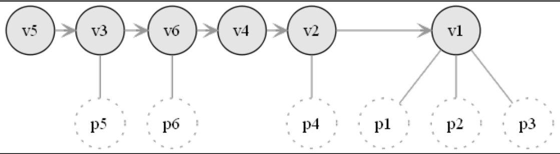


Figure A.1: Visual representation of graph at the beginning

The steps of the algorithm are illustrated in Table A.1. The *Input Graph* column shows the (modified) input graph with relevant nodes for this step highlighted with a dark color and relevant edges stressed by using dashed edges. The light-grey arrows between variable nodes indicate the order in which they appear in the Object-Z specification. In the last column the output variable list together with the predicates that have to be used in the declaration of this variable are shown.

Table A.1: Applied example of the ordering algorithm

Step	**Input Graph**	**Output**
1		
2		
3		
4		
5		
6		

According to *rule 1* of the algorithm the first variable to be added is *v*5, as it has a non-collection type, this is also visualized in the first row of Table A.1. As no other variables of non-collection type exist, the algorithm advances to the second rule. The first graph of step 2 in the table contains no predicate nodes without neighbors, so the algorithm may proceed with the nodes for collection type variables and their predicates in *rules 3 and following*. At this moment the graph does not contain variable edges without neighbors. According to *rule 4*, a predicate with only one outgoing edge has to be chosen next. Nodes *p*5 and *p*6 are eligible. The variable node *v*3 with its dependency to predicate *p*5 is added to *varOrder* as illustrated in the output graph of step 2 in the table. In the same way variable *v*6 is added to the output variable list in the third iteration step.

In step 4 of the table, there are 4 predicates and 3 variables left. All predicates use exactly two variables. So, *rule 6a* is relevant in this situation. Variable $v4$ has only one neighbor $p4$, $v2$ has 4 neighbors and $v1$ has three. Consequently $v4$ has to be chosen as the next variable in the output *varOrder*. After removing $v4$ and its adjacent edges from the graph, there is no predicate node without neighbors (*rule 7*), so no predicate node is paired to $v4$ in the output list.

Now, predicate node $p4$ has only one neighbor left and is found by *rule 4* of the algorithm. Therefore, node $v2$ is the next output node in *varOrder* and has $p4$ as its only depending predicate node. In the last step, predicate nodes $p1$, $p2$, and $p3$ have each one and the same variable node neighbor $v1$. So this node is the last node in the output list and all the three remaining predicates are paired as dependent predicate nodes.

All nodes from *varSeq* and *predSet* have been removed from the graph at this point. So the algorithm stops and outputs the list *varOrder* containing the variable ordering for the mapping to Perfect. For each variable exactly those predicates are paired with the variable, that have to be added in a `those`-clause in the declaration of this variable in the quantification expression in Perfect. The resulting mapping to Perfect of the example above is shown in Listing A.2. Applying expression $exp[newVar/oldVar]$ replaces all occurrences of *oldVar* in *exp* by *newVar*.

```
forall v5:type5,
       v3::(those i::col3 :- p5(v3)[i/v3]),
       v6::(those i::col6 :- p6(v6)[i/v6]),
       v4::col4,
       v2::(those i::col2 :- p4(v2, v4)[i/v2]),
       v1::(those i::col1 :- p1(v1, v2, v3)[i/v1] &
                             p2(v1, v2, v3)[i/v1] &
                             p3(v1, v2, v3)[i/v1])
    :- p(v1, v2, v3, v4, v5, v6)
```

Listing A.2: Ordering algorithm example in Perfect

APPENDIX B

Object-Z Grammar

The simplified ANTLR grammar used to parse the OZ input is provided in Listing B.1. Tables B.1 and B.2 show each Object-Z symbol with its corresponding OZ symbol and token name for the lexer. The full version of the grammar including also alias names for non-terminals used in the listener implementations is available from GitHub[1].

```
program : definition+ ;

definition : localDefinition | classDefinition ;

classDefinition : CLASS ID formalParameters? LCURLY
        visibilityList? inheritedClassList? localDefinitionList?
        state? initialState? operationList? RCURLY ;

visibilityList : VISIBLE LPARA feature (COMMA feature)* RPARA ;
feature : INIT | ID ;
inheritedClassList : INHERITS LCURLY (classDes SEMI)+ RCURLY ;
classDes : ID genActuals? ;
localDefinitionList : localDefinition+ ;

localDefinition : givenTypeDefinition | axiomaticDefinition | abbreviationDefinition
    | freeTypeDefinition ;

abbreviationDefinition : ID formalParameters? ABBRDEF expression ;
givenTypeDefinition : LBRACK ID (COMMA ID)* RBRACK ;
freeTypeDefinition : ID FTDEF ID ('|' ID)* ;
axiomaticDefinition : CONST LCURLY declarationList? predicateList? RCURLY ;
state : STATE LCURLY primary? (DELTA LCURLY secondary RCURLY)? predicateList? RCURLY ;
primary : declarationList ;
secondary : declarationList ;
initialState : INIT LCURLY predicateList RCURLY ;
operationList : (operationSchemaDef | operationExpressionDef)+ ;
operationSchemaDef : ID LCURLY deltalist? declarationList? predicateList? RCURLY ;
deltalist : DELTA LPARA ID (COMMA ID)* RPARA ;
operationExpressionDef : ID ISDEF operationExpression SEMI? ;
```

[1]Object-Z to Perfect Project, https://github.com/sylviaswoboda/objectz-2-perfect

```
operationExpression : DAND schemaText DOT operationExpression
    | DNCH schemaText DOT operationExpression
    | DSEQ schemaText DOT operationExpression
    | opExpression ;

opExpression : opExprAtom
    | opExpression AND opExpression
    | opExpression APAR opExpression
    | opExpression PAR opExpression
    | opExpression NCH opExpression
    | opExpression SEQ_OP opExpression
    | opExpression DOT opExpression ;

opExprAtom : LBRACK (DELTA LPARA ID (COMMA ID)* RPARA)? declarationList? ('|'?
    predicateList)? RBRACK
    | caller? id
    | LPARA operationExpression RPARA ;

caller : (id ATTR_CALL)+ ;
schemaText : schemaDeclarationList ( '|' predicate) ? ;
schemaDeclarationList : declaration (SEMI declaration)* SEMI? ;
declarationList : (declaration SEMI)+ ;
declaration : declarationNameList COLON expression ;
declarationNameList : id (COMMA id)* ;
predicateList : predicate (SEMI predicate)* SEMI? ;

predicate : (FORALL | EXISTS | EXISTS_1) schemaText DOT predicate
    | simplePredicate ;

simplePredicate : simplePredicate CONJ simplePredicate
    | simplePredicate OR simplePredicate
    | simplePredicate IMPL simplePredicate
    | simplePredicate EQUIV simplePredicate
    | predicateAtom ;

predicateAtom : NOT predicateAtom
    | id ATTR_CALL INIT
    | TRUE | FALSE
    | expression underlinedId expression
    | expression | LPARA predicate RPARA ;

expression
    : expression CARTESIAN expression
    | powerSetOp expression
    | prefix expression
    | expression (MULT | DIV | INT_DIV | MOD) expression
    | expression (PLUS | MINUS ) expression
    | expression RANGE expression
    | expression setOp expression
    | expression infixComparisonOp expression
    | expression infixRelationOp expression
    | expression RELATION expression RELATION expression
    | expression RELATION expression
    | (UNION | INTERSECT | CONCATENATE) LPARA expression RPARA
    | CLASS_HIER ID genActuals?
    | p=predefinedType
    | SUM schemaText DOT featureOrFunctionCall
    | op=(MIN | MAX) e=expression
    | id formalParameters
    | id genActuals
    | LCURLY schemaText (DOT expression)? RCURLY
    | LPARA expression (COMMA expression)+ RPARA
    | featureOrFunctionCall
    | LCURLY expression (COMMA expression)* RCURLY
    | LLBRACK expression (COMMA expression)* RRBRACK
    | LBRACK expression (COMMA expression)* RBRACK
    | EMPTYSET | EMPTYBAG | NCH
    | SELF
    | LPARA expression RPARA ;

genActuals : LBRACK expression (COMMA expression)* RBRACK ;
formalParameters : LBRACK ID (COMMA ID)* RBRACK ;
powerSetOp : POWER | POWER1 | FINITE | FINITE1 | SEQ | SEQ1 | ISEQ | BAG ;

infixRelationOp
    : PART_FUNC | TOT_FUNC | PART_INJ | TOT_INJ | PART_SUR | TOT_SUR | BIJEC
    | MAPLET | F_PART_FUNC | F_PART_INJ | DOM_RESTR | RAN_RESTR | DOM_AR | RAN_AR ;

infixComparisonOp : EQUALS | NEQUALS | ELEM | NELEM | SUBSET | STR_SUBSET
    | LT | LTE | GT | GTE | PREFIX | SUFFIX | IN_SEQ | IN_BAG | SUBBAG ;

setOp : UNION | DIFFERENCE | INTERSECT | CONCATENATE | OVERRIDE
    | EXTRACT | FILTER | MULTIPLICITY | SCALING | BAG_UNION | BAG_DIFFERENCE ;

prefix : MINUS | COUNT | RAN | DOM | TAIL | HEAD | REV | LAST | FRONT | ITEMS ;

featureOrFunctionCall : id (ATTR_CALL id)+
    | SUCC LPARA idOrNumber RPARA
    | id LPARA featureOrFunctionCall (COMMA featureOrFunctionCall)* RPARA
    | idOrNumber ;

idOrNumber : id | number ;
underlinedId : UNDERLINE id UNDERLINE;
id : ID DECORATION?;
number : INT | FLOAT;
predefinedType : NAT | PNAT | INTEGER | BOOL | REAL | CHAR ;

DECORATION : PRIME | QUEST | EXCL ;
ID : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_'('a'..'z'|'A'..'Z'|'0'..'9'))* ;

CLASS : 'class';
CONST : 'const';
INHERITS : 'inherits';
INIT : 'INIT';
STATE : 'state';
ATTR_CALL : '.';
UNDERLINE : '_';
SEMI : ';';
COMMA : ',';
COLON : ':';
LCURLY : '{';
RCURLY : '}';
LPARA : '(';
RPARA : ')';
LLBRACK : '|[';
RRBRACK : ']|';
LBRACK  : '[';
RBRACK : ']';

INT : ('1'..'9')('0'..'9')* | '0';
FLOAT : ('0'..'9')+'.'('0'..'9')+;

COMMENT : '/*' .*? '*/'     ->   channel(HIDDEN);
SL_COMMENT : '//' .*? '\r'? '\n'   -> channel(HIDDEN);
```

```
WS: (' ' | '\t' | '\r' | '\n')+   -> channel(HIDDEN);
```

Listing B.1: Grammar for OZ parsing

Table B.1: Object-Z symbols, token names in lexer and their corresponding OZ symbols

Object-Z	**Token**	**OZ**	**Object-Z**	**Token**	**OZ**
$\upharpoonright$	VISIBLE	`visible`	$\mathbb{P}$	POWER	`!P`
Δ	DELTA	`delta`	$\mathbb{P}_1$	POWER1	`!P1`
$\forall$	FORALL	`forall`	$\mathbb{F}$	FINITE	`!F`
$\exists$	EXISTS	`exists`	$\mathbb{F}_1$	FINITE1	`!F1`
$\exists_1$	EXISTS_1	`exists1`	seq	SEQ	`seq`
self	SELF	`self`	seq_1	SEQ1	`seq1`
$\varnothing$	EMPTYSET	`{}`	iseq	ISEQ	`iseq`
$[\![\,]\!]$	EMPTYBAG	`|[]|`	bag	BAG	`bag`
$\langle\,\rangle$	NCH	`[]`	$\times$	CARTESIAN	`><`
$\in$	ELEM	`in`	$\notin$	NELEM	`~in`
$\subseteq$	SUBSET	`<<=`	$\subset$	STR_SUBSET	`<<`
$\cup$	UNION	`++`	$\setminus$	DIFFERENCE	`\`
$\cap$	INTERSECT	`**`	$\#$	COUNT	`#`
min	MIN	`min`	*max*	MAX	`max`
$..$	RANGE	`..`	$\bigcup A$		`++(A)`
$\bigcap A$		`**(A)`	$\leftrightarrow$	RELATION	`<-->`
$\mapsto$	MAPLET	`|->`	$\rightarrow$	TOT_FUNC	`-->`
$\nrightarrow$	PART_FUNC	`-|->`	$⤔$	PART_INJ	`>-|->`
$\rightarrowtail$	TOT_INJ	`>--->`	$⤀$	PART_SUR	`-|->>`
$\twoheadrightarrow$	TOT_SUR	`--->>`	$⤖$	BIJEC	`>-->>`
$⇻$	F_PART_FUNC	`-||->`	$⤕$	F_PART_INJ	`>-||->`

Table B.2: Object-Z symbols, token names in lexer and their corresponding OZ symbols (cont.)

Object-Z	**Token**	**OZ**	**Object-Z**	**Token**	**OZ**
dom	DOM	`dom`	ran	RAN	`ran`
$\lhd$	DOM_RESTR	`<|`	$\rhd$	RAN_RESTR	`|>`
⩤	DOM_AR	`<||`	⩥	RAN_AR	`||>`
$\oplus$	OVERRIDE	`>O<`	$\frown$	CONCATENATE	`+^+`
$\upharpoonright$	FILTER	`filter`	$\upharpoonleft$	EXTRACT	`extract`
head	HEAD	`head`	*last*	LAST	`last`
front	FRONT	`front`	*tail*	TAIL	`tail`
rev	REV	`rev`	prefix	PREFIX	`prefix`
suffix	SUFFIX	`suffix`	in	IN_SEQ	`inseq`
$\frown/\ A$		`+^+(A)`	$\sharp$	MULTIPLICITY	`#`
$\otimes$	SCALING	`(><)`	⋿	IN_BAG	`inbag`
$\sqsubseteq$	SUBBAG	`subbag`	*items*	ITEMS	`items`
⩁	BAG_DIFFERENCE	`|-|`	$\uplus$	BAG_UNION	`|+|`
$*$	MULT	`*`	$/$	DIV	`/`
div	INT_DIV	`div`	mod	MOD	`mod`
$+$	PLUS	`+`	$-$	MINUS	`-`
$=$	EQUALS	`=`	$\neq$	NEQUALS	`~=`
$<$	LT	`<`	$\leqslant$	LTE	`<=`
$>$	GT	`>`	$\geqslant$	GTE	`>=`
Σ	SUM	`sum`	*succ*	SUCC	`succ`
$\mathbb{B}$	BOOL	`!B`	$\mathbb{Z}$	INTEGER	`!Z`
$\mathbb{N}$	NAT	`!N`	$\mathbb{N}_1$	PNAT	`!N1`
$\mathbb{R}$	REAL	`!R`	*Char*	CHAR	`!C`
true	TRUE	`true`	*false*	FALSE	`false`
$\Leftrightarrow$	EQUIV	`<=>`	$\Rightarrow$	IMPL	`=>`
$\vee$	OR	`or`	$\wedge$	CONJ	`and`
$/$	NOT	`~`	$\downarrow$	CLASS_HIER	`|v`
::=	FTDEF	`::=`	==	ABBRDEF	`==`
$\widehat{=}$	ISDEF	`^=`	$\wedge$	AND	`&&`
$\parallel$	PAR	`||`	$\parallel_!$	APAR	`||!`
$\Box$	NCH	`[]`	$\fatsemi$	SEQ_OP	`0/9`
$\bullet$	DOT	`@`	$\bigwedge$	DAND	`(&&)`
$\Box$	DNCH	`([])`	$\fatsemi$	DSEQ	`(0/9)`
$'$	PRIME	`'`	?	QUEST	`?`
!	EXCL	`!`			

Bibliography

[Abr74] Jean-Raymond Abrial. Data semantics. In *Data Base Management, Proceeding of the IFIP Working Conference Data Base Management, Cargèse, Corsica, France, April 1-5, 1974.*, pages 1–60, 1974.

[AN06] Adnan Ashraf and Aamer Nadeem. Automating the Generation of Test Cases from Object-Z Specifications. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, pages 101–104, Washington, DC, USA, 2006. IEEE Computer Society.

[CC04] David Crocker and Judith Carlton. A High Productivity Tool for Formally Verified Software Development. Internet, September 2004. `http://www.eschertech.com/papers/pdpaper.pdf`, last loaded on 29.10.2017.

[CM05] Gareth Carter and Rosemary Monahan. Introducing the Perfect Language. Technical Report NUIM-CS-TR-2005-06, Department of Computer Science, National University of Ireland, Maynooth, June 2005.

[CMM$^+$98] David Carrington, Ian MacColl, Jason McDonald, Leesa Murray, and Paul Strooper. From Object-Z Specifications to ClassBench Test Suites. *Journal on Software Testing, Verification and Reliability*, 10, 1998.

[Cro04] David Crocker. Safe Object-Oriented Software: The Verified Design-By-Contract Paradigm. In *Practical Elements of Safety: Proceedings of the Twelfth Safety-Critical Systems Symposium, Birmingham, UK, 17–19 February 2004*, pages 19–41, London, 2004. Springer London.

[CS90] David Carrington and Graeme Smith. Extending Z for Object-Oriented Specifications. Technical report, Department of Computer Science, The University of Queensland, Australia, 1990.

[DD90] David J. Duke and Roger Duke. Towards a Semantics for Object-Z. In *VDM '90: Proceedings of the Third International Symposium of VDM Europe on VDM and Z - Formal Methods in Software Development*, pages 244–261, London, UK, 1990. Springer-Verlag.

[DKRS91] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. The Object-Z Specification Language: Version 1. Technical report, Software Verification Centre, Department of Computer Science, The University of Queensland, Software Verification Centre, Department of Computer Science, The University of Queensland, Australia, 1991.

[DR00] Roger Duke and Gordon Rose. *Formal Object-Oriented Specification Using Object-Z*. MacMillan, 2000.

[Esc] Escher Technologies Limited. *Perfect Developer Language Reference Manual*. `http://www.eschertech.com/product_documentation/Language Reference/LanguageReferenceManual.htm`, version 7.0, last loaded on 29.10.2017.

[FHY94] M. Fukagawa, T. Hikita, and H. Yamazaki. A Mapping System from Object-Z to C++. In *Software Engineering Conference, 1994. Proceedings., 1994 First Asia-Pacific*, pages 220–228, Dept. of Computer Science, Meiji University, Higashimita, Tama-ku, Kawasaki 214, Japan, Dec 1994.

[Gri95] Alena Griffiths. An Extended Semantic Foundation For Object-Z. Technical report, Department of Computer Science, The University of Queensland, Australia, 1995. `ftp://svrc.it.uq.edu.au/techreports/tr95-39.ps.gz`, last loaded on 22.10.2017.

[Gri96] Alena Griffiths. A Semantics for a Simple Sub-language of Object-Z. Technical report, Department of Computer Science, The University of Queensland, Australia, 1996. `ftp://svrc.it.uq.edu.au/techreports/tr96-33.ps.gz`, last loaded on 22.10.2017.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hor00] I. Horton. *Beginning Java 2: JDK 1.3 Edition*. Programmer to programmer. Wrox Press, 2000.

[ISO02] Information Technology ISO/IEC. Z Formal Specification Notation – Syntax, Type System and Semantics, ISO 13568:2002 International Standard, 2002.

[Joh96] Wendy Johnston. A Type Checker for Object-Z. Technical report, Software Verification Centre, Department of Computer Science, The University of Queensland, 1996.

[JR93] Wendy Johnston and Gordon Rose. Guidelines for the Manual Conversion of Object-Z to C++. Technical report, Software Verification Centre, Department of Computer Science, The University of Queensland, 1993.

[Kim07] Tim G. Kimber. Object-Z to Perfect Developer, September 2007. `http://www.doc.ic.ac.uk/~tk106/ObjectZ_project.pdf`, last loaded on 29.10.2017.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[Lig01] David Lightfoot. *Formal Specification Using Z.* Palgrave, 2001.

[MCMS99] L. Murray, D. Carrington, I. MacColl, and P. Strooper. TinMan - A Test Derivation and Management Tool for Specification-Based Class Testing. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 32. Proceedings*, pages 222 –233, 1999.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction (2Nd Ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[MMS97] Jason McDonald, Leesa Murray, and Paul Strooper. Translating Object-Z Specifications to Object-Oriented Test Oracles. In *APSEC '97: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, pages 414–423, Washington, DC, USA, 1997. IEEE Computer Society.

[MS98] Jason McDonald and Paul Strooper. Translating Object-Z Specifications to Passive Test Oracles. In *ICFEM '98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, pages 165–174. IEEE Computer Society, 1998.

[MSH03] Jason McDonald, Paul Strooper, and Dan Hoffman. Tool Support for Generating Passive C++ Test Oracles from Object-Z Specifications. In *APSEC '03: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, page 322, Washington, DC, USA, 2003. IEEE Computer Society.

[Par07] Terence Parr. *The Definitive ANTLR Reference.* The Pragmatic Bookshelf, 2007.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2nd edition, 2013.

[PB04] Richard F. Paige and Phillip J. Brooke. Integrating BON and Object-Z. *Journal of Object Technology*, 3(3):121–141, 2004.

[PF11] T. Parr and K. Fisher. LL(*): The Foundation on the ANTLR Parser Generator, 2011. `http://www.antlr.org/papers/LL-star-PLDI11.pdf`, last loaded on 29.10.2017.

[RC92] G.-H. Rafsanjani and S. J. Colwill. From Object-Z to C++: A Structural Mapping. In *Proceedings of the Z User Workshop*, pages 166–179, London, UK, 1992. Springer-Verlag.

[Smi92] Graeme Smith. *An Object-Oriented Approach to Formal Specification.* PhD thesis, University of Queensland, 1992.

[Smi94] Graeme Smith. A Logic for Object-Z. In *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, 1994.

[Smi95a] Graeme Smith. A Fully Abstract Semantics of Classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[Smi95b] Graeme Smith. A Logic for Object-Z (Additional Rules). Technical report, Software Verification Centre, Department of Computer Science, The University of Queensland, 1995.

[Smi95c] Graeme Smith. Formal Verification of Object-Z Specifications. Technical report, Software Verification Centre, Department of Computer Science, The University of Queensland, 1995.

[Smi95d] Graeme Smith. Reasoning about Object-Z Specifications. In *APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference*, page 489, Washington, DC, USA, 1995. IEEE Computer Society.

[Smi00] Graeme Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.

[Spi89] J. M. Spivey. The Z Notation: A Reference Manual. *Prentice-Hall International Series In Computer Science*, 1989.

[SRBC92] Susan Stepney, C. J. Van Rijsbergen, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[Ste06] Brian Stevens. Implementing Object-Z with Perfect Developer. In *Journal of Object Technology*, volume 5, pages 189–202, March-April 2006. `http://www.jot.fm/issues/issue_2006_03/article5`, last loaded 29.10.2017.

[WS03] Kirsten Winter and Graeme Smith. Compositional Verification for Object-Z. In *ZB 2003: Formal Specification and Development in Z and B*, pages 280–299. Springer-Verlag, 2003.