IMPERIAL COLLEGE LONDON
DEPARTMENT OF COMPUTING

Object-Z to Perfect Developer

by

Tim G. Kimber

MCS IN COMPUTING SCIENCE
INDIVIDUAL PROJECT REPORT

First Supervisor: Dr Krysia Broda
Second Supervisor: Dr Alessandra Russo

SEPTEMBER 2007

Abstract

Formal specification is the process of creating precise, mathematical models of a proposed system. Such models provide an unambiguous description of the functionality of a system. The specification is the basis for a mathematical proof that implementation satisfies the stated requirements under all circumstances. *Object-Z* is a widely used object-oriented extension of the Oxford formal specification language Z. A specification is structured in a modular, object-oriented way to facilitate conversion to object-oriented programming languages. Perfect Developer is an object-oriented model verification tool that uses a less well known, less abstract language, *Perfect*. However, unlike any existing Object-Z tool, Perfect Developer incorporates an automated theorem prover to verify that specifications written with it are sound. This project has gone beyond previous work, and delivered a detailed formal mapping of a large part of the Object-Z language into Perfect. Furthermore, an EBNF grammar defining a textual version of Object-Z is defined. The grammar provides the basis for a translation tool that is capable of fully translating simple, beginner level Object-Z specifications. The translations provide immediate feedback for the user, by clarifying the meaning of Object-Z expressions (e.g. What inputs and outputs does a composite operation actually have? What is its overall precondition?), and allow formal verification by the Perfect Developer tool. This project provides a genuine link between two complementary software engineering techniques, allowing each to enhance the other.

Acknowledgments

I would like to thank Krysia Broda and Alessandra Russo for their enthusiastic guidance and encouragement; Vanessa Ho Von for her helpful suggestions regarding the use of ANTLR and StringTemplate; David Crocker and Judith Carlton of Escher Technologies for providing a copy of Perfect Developer; and my wife, Wendy, for proof-reading this report, and for single-handedly keeping our lives together for the past four months.

Contents

Chapter 1

INTRODUCTION

1.1 AIMS AND MOTIVATION

Object-Z[1] is a formal specification language which uses mathematical expressions to model complex and safety-critical systems. The specification captures the essential properties of such a system, and the states to which they apply. This allows developers to check that their code cannot lead to a state in which the system violates the specification, and provides a mathematically sound way to ensure the system is error-free. However, the heavily mathematical and semi-graphical notation of Object-Z make it difficult, especially for those new to the language, to be confident that they have expressed what they meant to, and that the specification itself is sound.

Perfect Developer[2] (PD) is an object-oriented software development tool, developed by Escher Technologies in the UK, also based on formal methods. PD users create an abstract data model in its language, Perfect, which implements Verified Design by Contract, an extension of the Design by Contract system introduced in Eiffel[3]. The methods of the classes in Perfect are expressed as preconditions and postconditions, with added assertions, and safety and liveness properties, using similar expressions to those in Object-Z. Once the model has been created, implementation code can be added and the whole thing can be verified using PD's built-in theorem prover.

Thus, Object-Z is a widely used and understood modelling language that lacks automated tool support, and Perfect Developer is an object-oriented model verification tool that uses a less well known, less abstract language. This provides an opportunity to link the two and enhance the capabilities of both. Therefore, the formal aims of this project were to:

1. Construct a formal mapping of Object-Z into Perfect

2. Build a tool to capture Object-Z specifications and translate them into Perfect using the mapping developed in (1)

3. Using the Perfect Developer verification tool, provide a function to check translated specifications

1.2 RELATED WORK

Despite several reports of work on automation of Object-Z editing and checking[4,5,6,7,8,9,10], there are as yet very few type-checkers available and no verification tools. To address this situation, several projects have been undertaken by Imperial students to provide some mechanism for checking Object-Z specifications[11,12,13,14,15,16,17]. The majority of these have focussed on translating Object-Z into the Java Modelling Language (JML) and so that option has been well studied. One of the Imperial MSc theses (Dixon[12]) and one other study (Stevens[18]) have also investigated the translation of Object-Z into Perfect. However, the mappings set out are quite high level, making implementation unfeasible, and indeed no implementation is reported. This project goes further towards providing a detailed, comprehensive mapping and reporting a full implementation.

1.3 OUTCOMES

After familiarization with the two languages (see Chapters 2 and 3), largely by working through the shopping scanner example described in Chapter 7, the formal mapping presented in Chapter 4 was created. This work was prioritized based on the perceived "usefulness" of each aspect of Object-Z, with those constructs most frequently encountered mapped first. In all 75-80% of Object-Z has been mapped to Perfect, allowing most simple specifications to be translated. This is in line with the project's aim to help those new to Object-Z understand it better, and be more comfortable working with it. Large sections of more complex models also fall within the mapping and, with more work the potential clearly exists to extend the mapping to the whole language.

Development of the mapping was carried out side-by-side with development of the tool to perform the translation and verification. Creation of this tool was an important aspect of the project as it provided feedback on the accuracy and completeness of the mapping, and proof that the translation was feasible in practice. After investigating different options for the underlying translation mechanism, an ANTLR-produced parser was chosen. More details on ANTLR and the parsing methodology can be found in Chapters 5 and 6.

So, tool development had four main parts: creation of an input grammar; creation of a translation grammar and output templates; creation of auxiliary Java data structures; and creation of the graphical user interface. The details of this work can be found in Chapter 6

Finally, the tool was evaluated with reference to two case studies, as detailed in Chapter 7.

1.4 CONCLUSION

The project has gone beyond previous work, and delivered a detailed formal mapping of a large part of the Object-Z language into Perfect, with every indication that translation of almost all specifications is theoretically possible. Furthermore, a grammar for a textual version of Object-Z has been created from scratch, and this grammar is more comprehensive even than the mapping, allowing future development to proceed from a sound footing. The grammar provides the basis for a translation tool that is capable of fully translating simple, beginner level Object-Z specifications. The translations provide immediate feedback for the user, by clarifying the meaning of Object-Z expressions (e.g. What inputs and outputs does a composite operation actually have? What is its overall precondition?), and allow formal verification by the Perfect Developer tool. With further work, it seems likely that a robust and useful tool, capable of translating far more complex specifications could be developed. In summary, this project provides a genuine link between two complementary software engineering techniques, allowing each to enhance the other.

Chapter 2

OBJECT-Z

2.1 INTRODUCTION

2.1.1 Formal specification

Many strategies have been developed to enable creation of software that meets stated requirements and is error-free. Good testing can ensure the correct outcome in a range of likely and even unlikely scenarios, but this range is limited by the test data used (i.e. the test team's imagination and diligence). When the system is extremely complex, and particularly where errors can involve danger to life, a more rigorous method is needed.

Formal specification is the process of creating precise, mathematical models of a proposed system. Such models provide an unambiguous description of the functionality of the system. The specification is the basis for the implementation of the system, and for formal verification of it, i.e. a mathematical proof that the implementation meets the specification under all circumstances.

2.1.2 Z and Object-Z

Object-Z is an object-oriented extension of the Oxford formal specification language Z. Z and Object-Z provide a method of specifying the set of states that a system can legally be in, i.e. the allowable range of values of all variables, and the legal ways of moving from one state to another, the system's operations. In the case of Object-Z it is the state of the objects in the system that is defined. The specification is structured in a modular, object-oriented way to facilitate conversion to object-oriented programming languages.

2.1.3 Credit card bank accounts case study

To illustrate the central features of an Object-Z specification, we will follow the simple credit card bank account system described in Duke & Rose, Chapter 1[19]. This system is envisaged as a collection of credit card account objects. Each account is conceived as an object that records two numbers, the current balance of the account (positive if in credit, but often negative) and

the credit limit. Money can be withdrawn or deposited, which will change the balance, however, a withdrawal which causes the account to exceed its limit is not permitted. There is also an operation to withdraw the full amount available (up to the limit). Finally, the limit on a given account can be either $1000, $2000 or $5000.

2.2 THE CLASS CONSTRUCT

Z is a semi-graphical notation which uses boxes to represent different system states. Object-Z takes this approach and applies it to objects by making each class a named box, containing various component constructs, including other boxes. The Object-Z specification of a credit card account object, as informally described above, is shown in Figure 1.

2.2.1 Visibility list

The visibility list:

$$\upharpoonright (limit, balance, Init, withdraw, deposit, withdrawAvail)$$

itemizes the parts, or features, of the class which are visible to the environment containing a CreditCard object. In object-oriented terms the visibility list defines the interface of the class. If the visibility list is omitted, all features of the class are visible by default. This ensures that, as one would expect, at least one feature of a class must be visible

2.2.2 Constants definition

Class constants are specified by an open schema box, or *axiomatic definition.*

$$\begin{array}{|l} limit: \mathbb{N} \\ \hline limit \in \{1000, 2000, 5000\} \end{array}$$

This follows the typical Z structure of a list of declarations, above the line, and a list of formulas in first order predicate logic (FOPL), below the line. In the example, limit is declared as a natural number, and constrained to be one of the set {1000, 2000, 5000}. In general, any state in which the value of the declared quantities makes the group of formulas below the line (the *predicate*) true is a permitted state of the object. However, the object can only move from one state to another via the specified operations (see below), and since

limit is a constant, no operation should change its value. The net effect of this is that different credit cards may have different limits, but each one has an unchanging value for limit, which is either 1000, 2000 or 5000.

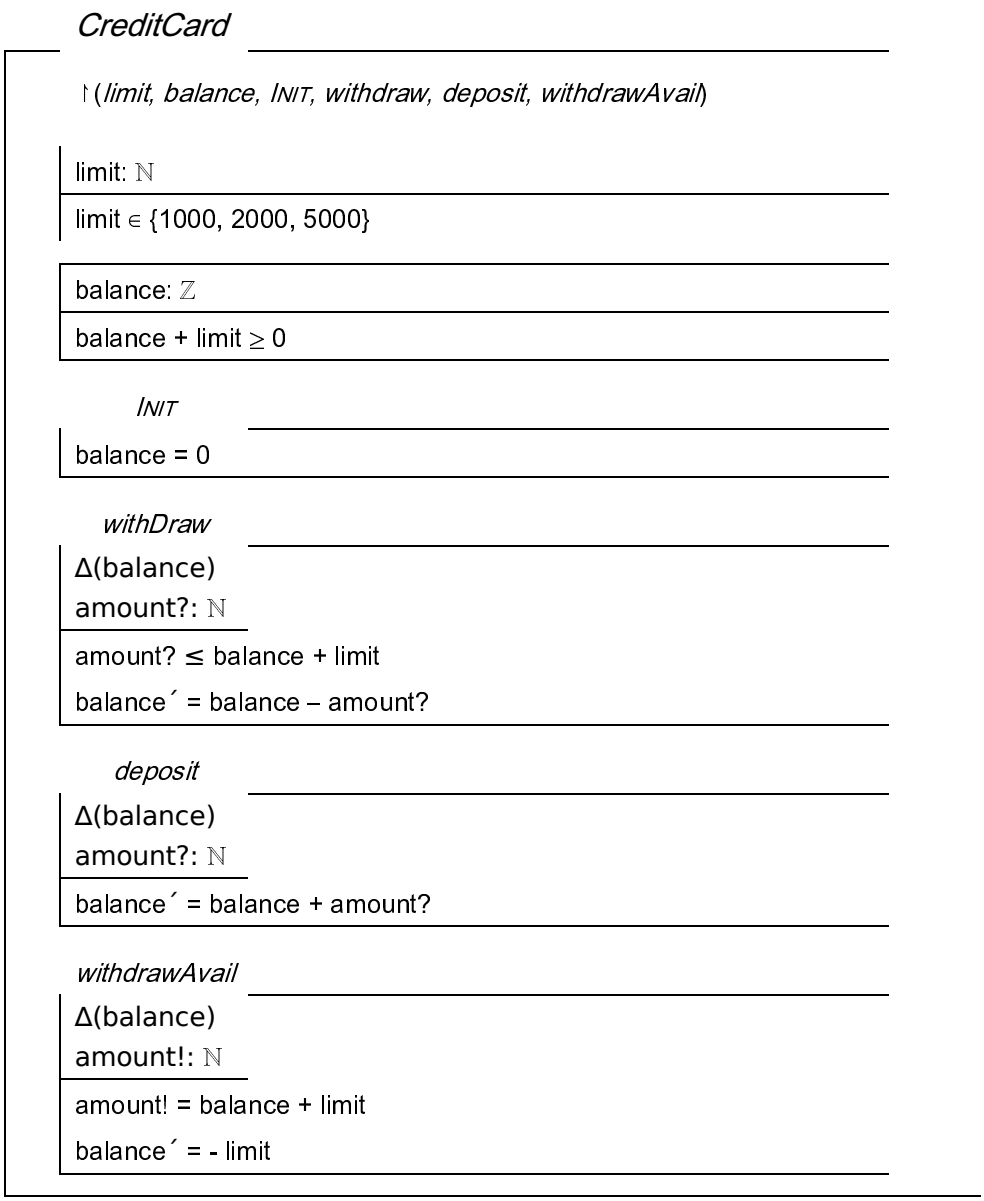
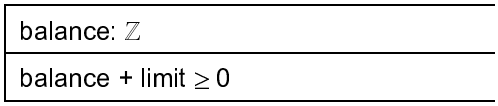


Figure 1. The Object-Z schema for class CreditCard

2.2.3 State schema

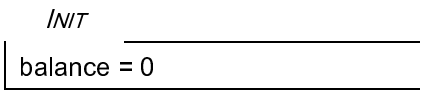
The state schema is a closed, unnamed box construct.



It follows exactly the same pattern as the constant schema, with declarations and predicate. This time though, the declared attributes correspond to the class variables, and the predicate gives *class invariants*. In the example, the only variable is the integer, balance. The invariant specifies that balance + limit must be greater than or equal to zero, i.e. balance cannot be more negative than –limit. CreditCard objects must satisfy this condition before and after every operation.

2.2.4 Initial schema

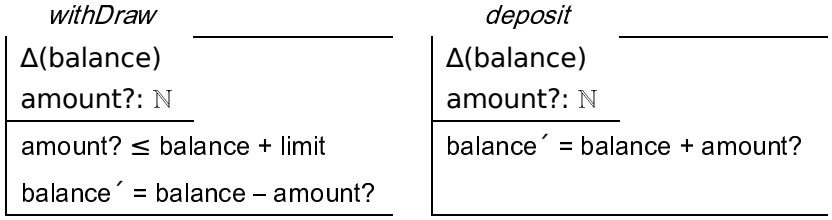
The initial schema is another box, containing only a predicate and always named INIT.



The predicate of the initial schema must be true for all objects in their initial state, i.e. after they have been created and before they have undergone any operations. The initial schema in the example specifies that all credit card accounts have an opening balance of zero.

2.2.5 Operation schemas

The remaining boxes in the CreditCard class specification are operation schemas:



withdrawAvail

Δ(balance)
amount!: ℕ

amount! = balance + limit
balance´ = - limit

Operation schemas define allowed mechanisms for the object to move from one state to another. The declaration part of the schema starts with a *delta list,* Δ(balance). The delta list shows all the class attributes which are changed by the schema. Any attributes not listed will implicitly remain unchanged when the object undergoes the operation. After the delta list comes the declaration of any inputs or outputs, or *communication variables,* that are passed between the object and the environment as part of the operation. Following the Z convention inputs are decorated with a '?' and outputs with a '!'.

Finally the FOPL formulas in the predicate specify constraints on the state variables and communication parameters before and after the operation. Again, the Z convention of unprimed and primed symbols respectively denoting the value of a state variable before and after the operation is used, and only states of the object which make the predicate true in its entirety conform to the specification.

Statements in the predicate specifying constraints on the value of inputs, or of state variables, before the operation amount to *preconditions.* If an input or a state variable violates a precondition then the operation is said to be *inapplicable* to the object. This is different to the programming situation where a function fails or causes an error – an object cannot undergo an inapplicable operation in Object-Z.

2.3 OBJECTS AND OBJECT INTERACTION

Once individual objects have been specified in a class construct, they can be included in larger systems. The credit card account example is expanded in Figure 2 which shows the specification of the class TwoCards. TwoCards contains two instances of the CreditCard class, c_1 and c_2, which are used to

spread and manage a larger debt. TwoCards has operations to deposit and withdraw money from the different cards, and to transfer between them.

2.3.1 Secondary variables

The two instances of CreditCard, c_1 and c_2, are declared in the state schema of TwoCards in the normal way. These are primary variables. However, the other variable, totalbal is a secondary variable, declared after the Δ symbol. Secondary variables are defined in terms of primary variables and change whenever the latter do.

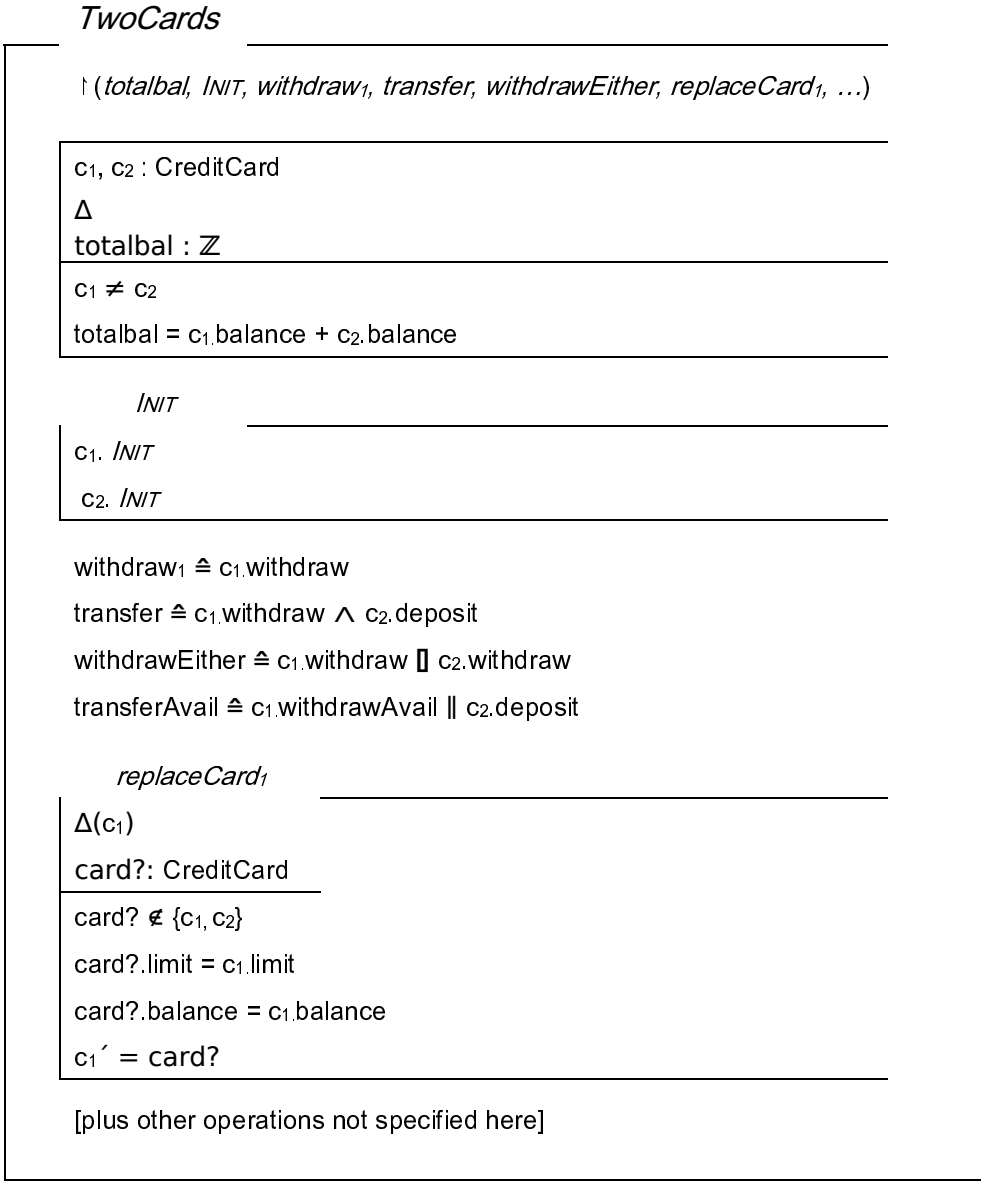


Figure 2. Object-Z specification of the class TwoCards

In the example, totalbal is defined as c_1.balance + c_2.balance (the primary variables do not need to be declared in the same object as the secondary) and will update whenever either balance changes. Secondary variables do not have their values manipulated directly in schema predicates (e.g. totalbal′ = 5).

2.3.2 Initial schema referencing

TwoCards' INIT schema

INIT

c_1. *INIT*

c_2. *INIT*

specifies the initial state in terms of the initial state of c_1 and c_2. This specifies that a TwoCards object in its initial configuration consists of two CreditCard objects, both in their initial configurations.

2.3.3 Operation expressions

In addition to boxed operation schemas, operations can be defined as combinations of other, existing operations. The *TwoCards* class construct contains these four expressions:

$withdraw_1 \triangleq c_1.withdraw$

$transfer \triangleq c_1.withdraw \wedge c_2.deposit$

$withdrawEither \triangleq c_1.withdraw \,[]\, c_2.withdraw$

$transferAvail \triangleq c_1.withdrawAvail \parallel c_2.deposit$

The ≙ symbol can be read as "is defined as". The simplest of these expressions defines operation *withdraw*$_1$ of *TwoCards* as applying operation *withdraw* to object c_1. This is known as operation *promotion*. There are no communication variables mentioned in the declaration of *withdraw*$_1$, however, since *withdraw* receives an input, *amount?*, from the environment (see Figure 1), so too does the promoted operation. Also, *withdraw*$_1$ is only applicable if *withdraw* is applicable to c_1, i.e. if *amount?* ≤ c_1.*balance* + c_1.*limit* .

2.3.3.1 Conjunction expressions

The operation *transfer* is defined as a *composite* operation which is the conjunction of *withdraw* applied to c_1 and *deposit* applied to c_2, signified by the '∧' operator. The *transfer* operation is thus defined as equivalent to both of

these individual operations running in parallel, that is to say independently, except that the input to both, *amount?*, will be the same. This joint input only comes about because the names of the inputs in the two operation schemas are identical, otherwise no equating of the two occurs. In the case of *transfer*, the effect, not surprisingly, is to deduct an amount from the balance of one card and add it to the other, effectively transferring a single sum. The composite operation, *transfer*, is only applicable if *withdraw* is applicable to c_1 and *deposit* is applicable to c_2.

2.3.3.2 Choice expressions

The third composite operation of TwoCards, *withdrawEither*, is defined using the choice operator '[]':

$$withdrawEither \triangleq c_1.withdraw \; [] \; c_2.withdraw$$

This signifies that *withdrawEither* offers a choice between two alternative outcomes. Operation withdraw will be applied to either c_1 or c_2 according to the following rules:

- the communication variables of both possible operations must be the same. In this case the one input is *amount?*
- if neither possible operation is applicable, nor is the composite. So, if *withdraw* is not applicable to either c_1 or c_2 then *withdrawEither* is not applicable to the *TwoCards* object
- if one possible operation is applicable but not the other, then the one that is applicable will run
- if both possible operations are applicable then one or the other, but not both, will run

2.3.3.3 Parallel composition

Parallel composition is similar to conjunction, but also facilitates inter-object communication. In the expression:

$$transferAvail \triangleq c_1.withdrawAvail \parallel c_2.deposit$$

the '||' operator signifies that *transferAvail* is defined to be *withdrawAvail* applied to c_1 and *deposit* applied to c_2, run in parallel, but with the input to the *deposit*

operation, *amount?,* being equal to the output of the *withdrawAvail, amount!*. Thus, outputs of one operation and inputs of the other that have the same base name are paired, equated, and also hidden from the environment, i.e. *transferAvail* itself has no input from, or output to, the environment. If there are the appropriate pairs, this communication can be two-way.

2.3.3.4 Sequential composition

The final way of combining operations is by using the sequential composition operator, '⨟'. If *CreditCard* had an operation that output a message confirming the balance of the account, let's call it *printBal*, then the expression:

$$transferConfirm \triangleq transferAvail \fatsemi c_2.printBal$$

would be equivalent to transferring all funds from c_1 to c_2 and then printing the new balance of c_2. So, sequential composition is the only operator that includes the concept of ordering the operations it combines, introducing an intermediate state, which is the starting point for the second operation. (Although, *transferConfirm* will be atomic when it is applied.) Sequential composition provides inter-object communication like '‖', pairing and hiding inputs and outputs, but only going from left to right, because of its sequential property.

2.4 OBJECT AGGREGATION

Larger systems with arbitrary or variable numbers of objects can be modelled using set notation. Figure 3 shows the class *CreditCards* which contains the attribute *cards* of type '$\mathbb{P}$CreditCard'. This denotes a set of *CreditCard* objects of variable size. The only constraint on this set is that:

$$\forall c:cards \bullet c.limit = commonlimit$$

which uses the Object-Z convention 'member : collection' to declare the local variable, c, to be an arbitrary element of *cards,* and specifies that all *CreditCards* in *cards* have the same limit.

Collections can also be modelled as other, more refined, types of sets: relations, functions and sequences. For a fuller description of the notation used to define these sets and specify changes to them, see Duke and Rose (Appendix A: Background Notation)[19], or Lightfoot[20].

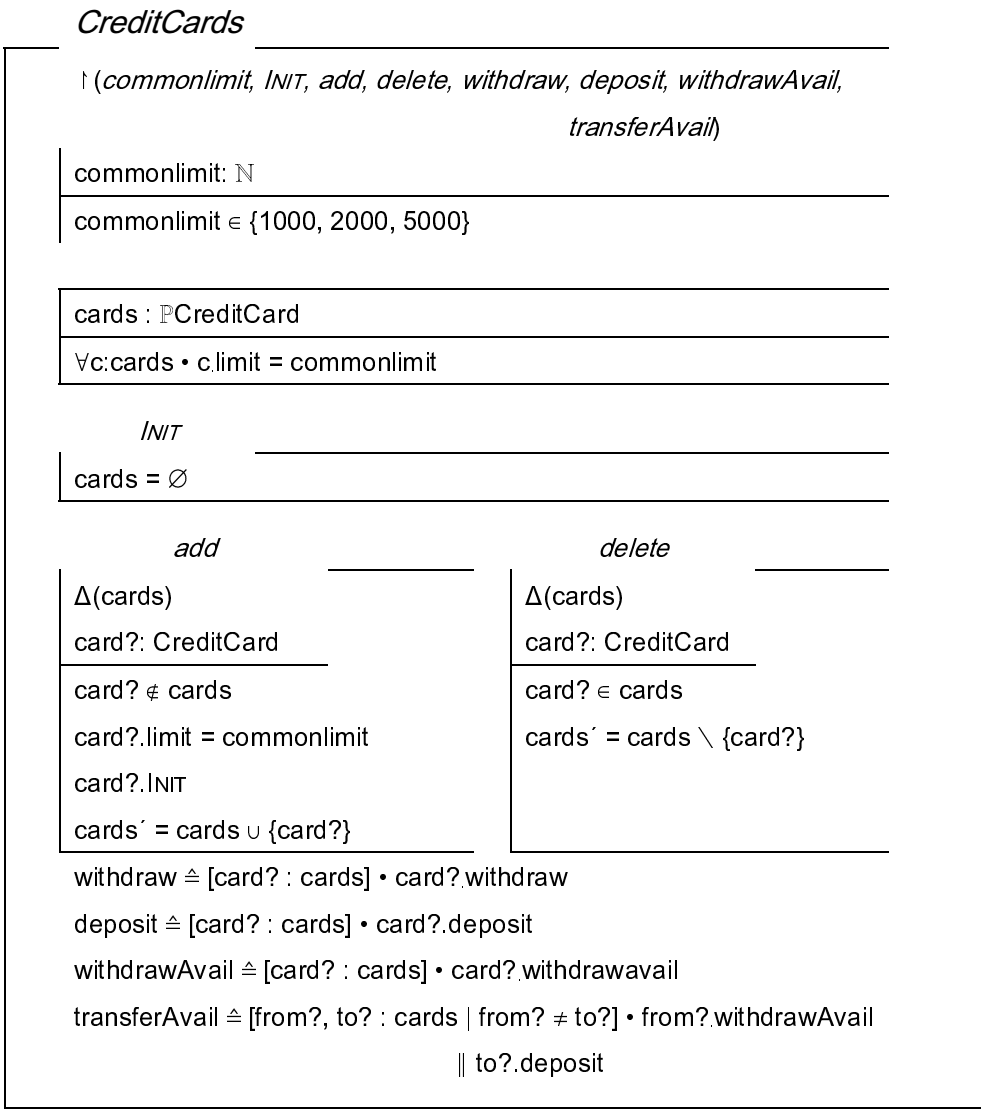


Figure 3. Object-Z schema for the class CreditCards.

2.4.1 Modifying aggregates

The operations *add* and *delete* in class *CreditCards* change the composition of *cards* by defining its value after they have been performed to include a new element, *card?,* or exclude an existing element, *card?*.

2.4.2 Selecting objects

The operation *withdraw* in *CreditCards*:

withdraw ≙ [card? : cards] • card?.withdraw

takes an input *card?* which is selected from *cards* by the environment. This is denoted by the "text schema", [card? : cards], containing the declaration of *card?*. The full version of this kind of schema is seen in operation *transferAvail*, where the objects *from?* and *to?* are declared as members of *cards* that satisfy the predicate $from? \neq to?$. The general form is:

[declarations | predicate]

as with other types of schema. The text schema is followed by the '•' symbol which indicates that the schema on the left "enriches" the environment of the expression on the right.

2.5 INHERITANCE

Object-Z includes a mechanism for both single and multiple inheritance. This may be used to indicate an object-oriented subclass relationship between two specified classes, although an Object-Z class that inherits from another is not implicitly a subtype.

When a class is inherited by another it is shown above the constant and/or state schema, such as the *CreditCardCount* class in Figure 4 which inherits the *CreditCard* class, extending its behaviour by counting the number of withdrawals made.

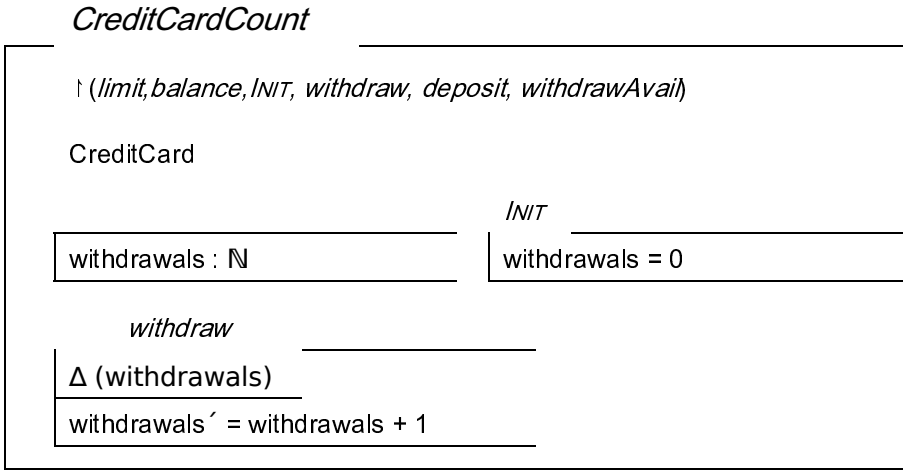


Figure 4. The class CreditCardCount.

The visibility list is not inherited, so *CreditCardCount* declares its own. Other parts of the new class will be the conjunction of what is shown in Figure 4 with the corresponding section of *CreditCard*. So, the new declaration withdrawals : ℕ is added to the state schema of *CreditCard* to define the state of a *CreditCardCount* object. Since operation *withdraw* has already been declared in *CreditCard*, the behaviour of the operation in *CreditCardCount* is, again, determined by the conjunction of the original definition and the additions shown in Figure 4.

2.5.1 Renaming

Any feature of a class can be renamed when it is inherited. For example, a semantically equivalent version of *withdraw* in *CreditCardCount* could have been derived as shown in Figure 5 by renaming the *withdraw* operation of the inherited *CreditCard* class.

The renaming notation is [newname/oldname].

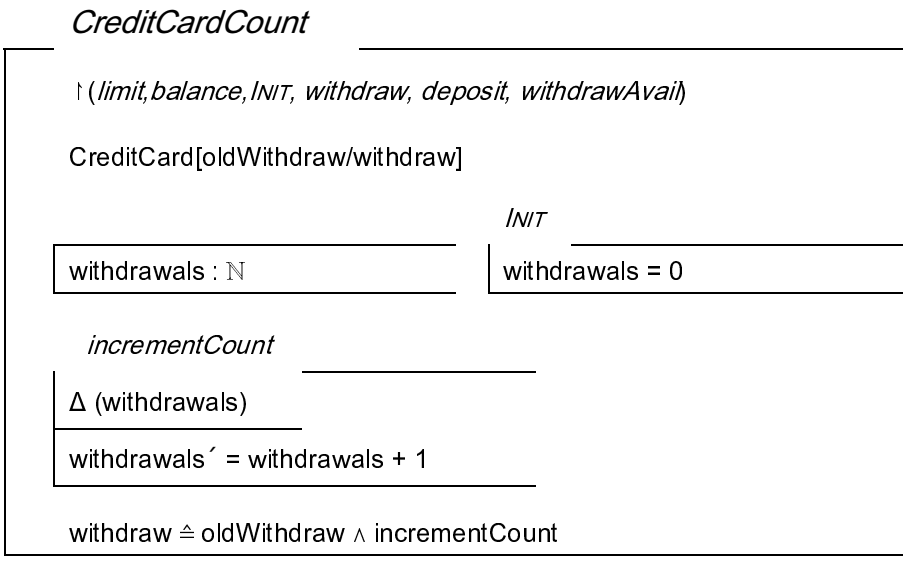
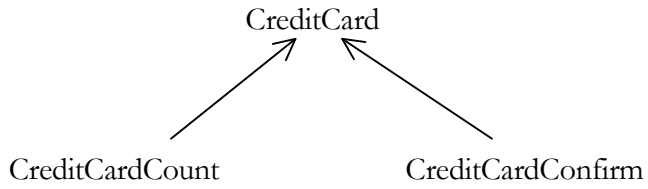


Figure 5. CreditCardCount using renaming.

2.5.2 Polymorphism

Declaring a variable with the polymorphic operator '↓' signifies that the object it represents may be of the class declared or any class that inherits from it. So, given the inheritance hierarchy:



the *cards* set of the class *GenCreditCards* shown in Figure 6 may contain *CreditCard, CreditCardConfirm* or *CreditCardCount* objects.

The operations *withdraw, deposit* and *transferAvail* of *GenCreditCards* will accept any *CreditCard* or one of its subclasses as input, but *withdrawConfirm* has a constraint in its object selection schema indicating that its input is limited to objects of type *CreditCardConfirm* (the class *CreditCardConfirm* is not shown).

GenCreditCards

↾(*INIT, add, delete, withdraw, deposit, transferAvail, withdrawConfirm*)

cards : ℙ↓CreditCard

∀c:cards • c.limit = commonlimit

INIT

cards = ∅

add

Δ(cards)

card?: ↓CreditCard

card? ∉ cards

card?.INIT

cards´ = cards ∪ {card?}

delete

Δ(cards)

card?: ↓CreditCard

card? ∈ cards

cards´ = cards \ {card?}

withdraw ≙ [card? : cards] • card?.withdraw

deposit ≙ [card? : cards] • card?.deposit

transferAvail ≙ [from?, to? : cards | from? ≠ to?] • from?.withdrawAvail
‖ to?.deposit

withdrawConfirm ≙ [card? : cards | card? ∈ CreditCardConfirm] •
card?.withdrawConfirm

Figure 6. The class GenCreditCards.

This concludes this summary of the main features of the Object-Z language.

Chapter 3

PERFECT DEVELOPER

3.1 INTRODUCTION

Perfect Developer is an object-oriented development tool for producing software that is mathematically proven to be correct. To use Perfect Developer, you begin by constructing a model using its own language, Perfect. Perfect implements a form of Design by Contract, as introduced in Eiffel[3]. This entails specifying preconditions and postconditions for all methods, establishing a contract between the client code calling the method and the method code. Assertions and general properties about the system's required behaviour are added to generate the specification. The model is then refined manually to code or a code generation function can attempt to do it for you. To prove that the produced software is correct, *Perfect Developer* first generates "proof obligations" from the preconditions, invariants, assertions etc. and then attempts to verify that the model satisfies them using its automated theorem prover.

3.2 THE PERFECT LANGUAGE

This overview of the Perfect language is based on the online tutorial[21] available at the Perfect Developer website[22].

3.2.1 Class structure

The definitions of the variables, methods etc. of a Perfect class are divided into two or possibly three different sections. The most commonly seen sections are abstract and interface, as shown in this translation of the credit card account example:

```
class CreditCard ^=
abstract
        const limit: nat ^= 1000;
        var balance: int;
        invariant balance + limit >= 0;
interface
        build{!balance: int}
        pre balance = 0;
end;
```

All data is declared in the abstract section and all publicly visible methods are declared in the interface section. The third section is the confined section. Methods placed in confined are visible to child classes only.

3.2.2 Invariants

As well as data, class invariants such as the constraint on the balance of the credit card class are declared in the abstract section usng the keyword "invariant".

3.2.3 Constructors

Every class must have at least one constructor and each constructor must define the values of all abstract data members. A constructor is defined using the keyword "build" followed by a list of parameters in curly braces, {}. Data initialization may be done directly from the constructor parameters by using the member name as the parameter name and prefixing the parameter declaration with ' ! '. Alternatively, the data members may be initialized in the constructor postcondition:

```
build{balance: int}
    post balance! = 0;
```

Placing the ' ! ' decoration after *balance* in this postcondition declares that the value of *balance* changes and its final value is 0.

3.2.4 Collections

Perfect has three built-in collection types, **set of** *X*, **seq of** *X* and **bag of** *X*, where X may be any built-in or declared type. There are also many operators and operations provided to manipulate these collections, including:

construction: `set of int {1,2,5}`

membership: `x in my_set`

subset: `this_set <<= big_set`

union/concatenation: `a_seq ++ b_seq`

cardinality: `# my_seq`

universal quantification: **forall** identifier::collection :- condition

existential quantification: **exists** identifier::collection :- condition

conditional construction: **those** identifier::collection :- condition

member selection: **that** identifier::collection :- condition

non-deterministic selection: **any** identifier::collection :- condition

collection modification: **for** identifier::collection **yield** expression

Finally, there is also a **map of (**X **->** Y**)** type which represents a set of type X, each with an associated value of type Y. This is equivalent to a function in Z, and indeed has methods *dom* and *ran* which return the domain and range of the mapping.

3.2.5 Functions and selectors

After constructors, the next types of method are functions and selectors. The simplest form of function is just to redeclare an abstract variable, making it accessible from outside the class:

```
class CreditCard ^=
abstract
        const limit: nat ^= 1000;
        var balance: int;
        invariant balance + limit >= 0;
interface
        function balance;
        build{!balance: int}
                pre balance = 0;
end;
```

This is read-only access since a function cannot modify any values, neither abstract variables nor parameters. Redeclaring a variable as a selector provides read-write access.

Functions can, in fact, be defined to return any information contained within the class or calculated from it, as long as they change no values or have any other "side-effects". This function is defined to return true or false depending on the input value *price:*

```
function canIBuyIt(price: nat) : bool
        ^= limit + (balance – price) >= 0;
```

3.2.6 Schemas

A method that is able to change abstract variables or parameter values is called a schema. A schema is declared in a similar way to a function, with the exception that it must always have a postcondition, rather than a ^= definition. Postconditions can be expressed in a variety of different ways. Some of the more common are shown in the expanded *CreditCard* class:

```
class CreditCard ^=
abstract
        const limit: nat ^= 1000;
        var balance: int;
        invariant balance + limit >= 0;
interface
        function balance;
        function limit;
    build{!balance: int}
          pre balance = 0;

    schema !withdraw(amount: nat)
          pre amount < balance + limit
          post change balance satisfy balance' = balance – amount;

    schema !deposit(amount: nat)
          post balance! = balance + amount;

    schema !withdrawAavail(amount!: out nat)
          post balance! = -limit,
              amount! = balance + limit;
end;
```

Prefixing the schema name with ' ! ' signifies that the method can alter the abstract data of the invoked object; without the decoration it can only alter its parameters.

3.2.6.1 Calling schemas

The next form of postcondition involves a call to another schema:

```
class TwoCards ^=
abstract
   var c1, c2 : CreditCard;
   invariant c1 ~= c2;

interface
   function c1;
   function c2;
```

```
    schema !transfer(amount: nat)
        pre amount <c1.balance + c1.limit
        post c1!withdraw(amount) & c2!deposit(amount);

    build{!c1:CreditCard, !c2:CreditCard}
        pre c1 ~= c2 & c1.balance = 0 & c2.balance = 0;
end;
```

This call behaves exactly as a method call in any object-oriented language would. Again, the ' ! ' decoration after the name of the object whose schema is being called indicates that the value of the object (its state) may be changed.

3.2.6.2 Conditionals

The general form of a conditional in Perfect is:

```
([ test1]: expression, [test2]:expression2)
```

and this maybe used in postconditions to make changes dependent on conditional tests.

This concludes this brief introduction to the Perfect language. Further details of those sections of Perfect that provide translations of Object-Z expressions can be found in the next chapter.

3.3 MODEL VERIFICATION

The Perfect Developer tool can be used to perform various checks on a model once it has been written. Syntax is checked, then fundamental errors such as illegal accesses of non-visible class members. Finallly, Perfect generates and verifies "proof obligations" for all stated invariants, preconditions, conditional tests etc. The tool's automated theorem prover attempts to prove each condition in turn. A full list of the verification conditions generated by Perfect Developer is available at the Escher Technologies website, but some of the ones more relevant to the abstract model are:

> *Expressions modified by schema are independent.* Generated for each call to a schema that modifies more than one object, to check that these objects are independent of each other.
>
> *Objects modified in parallel are independent.* Generated for parallel postconditions (i.e. where conditions are combined with ',' or '&', and

forall postconditions). The verification condition checks that the objects modified by each component of the condition are independent of each other.

Operand of [that | any] has at least one qualifying element. Generated at every *any* expression and every *that* expression with a condition. For a *that* or *any* with a condition, the verification condition checks that there exists an element of the collection that satisfies the condition. For an *any* with no condition checks that the collection is non-empty.

Post-assertion valid. Generated whenever a post-assertion is declared by a method declaration.

Postcondition specifies value for uninitialised data. Generated for all constructors and schemas with "out" parameters. A check is generated for each uninitialised data member or "out" parameter to ensure that it has a value specified for it before being used.

Precondition satisfied. Generated at the point of call to any function, operator, selector, constructor or schema with a precondition.

Type constraint satisfied. Generated for parameters, return values, variables written to by schemas, and objects modified in part by a postcondition with a constrained type.

Type constraint satisfied in assignment. Generated wherever a postcondition requires a value be assigned to a variable with a more constrained type.

Obligations are either proved, refuted, or left unproven once a time-limit is reached. Detailed information about the proofs and attempted proofs is generated by Perfect Developer and saved into files for later inspection. It is these checks and the feedback from them that we shall be using to verify the translated Object-Z specifications.

Chapter 4

MAPPING OBJECT-Z TO PERFECT

4.1 OPERATORS

The operators in Table 1 have equivalents in Perfect as shown, and will not be discussed further in this chapter.

Table 1. Object-Z operators and their Perfect equivalents.

	Object-Z	Perfect
arithmetic	+,-,*,/,%	+,-,*,/,%
comparison	>,<,>=,<=	>,<,>=,<=
set membership	∈	in
subset	⊆	<<=
strict subset	⊂	<<
union	∪	++
set difference	\	--
intersection	∩	**
set size	#	#
sequence concatenation	⌒	++
logical equivalence	⇔	<==>
logical implication	⇒	==>
logical and	∧	&
logical or	∨	|
logical negation	¬	~

4.2 EQUALITY AND OBJECT IDENTITY

Object-Z semantics define that the value associated with an object identifier is the identity of the object, as opposed to its state (i.e. the values of its data members) (Duke & Rose[19], p13). So, the expression $c_1 \neq c_2$ in Object-Z means that c_1 and c_2 represent distinct objects. The internal states of c_1 and c_2 may be identical, but this is not sufficient to make them equal. Perfect, however, defines equality of objects to be determined by the values of their data members, and does not provide a means of equating objects by identity.

x = 5	maps to→	x = 5
x = y (x,y are "primitives" or sets)	maps to→	x = y
x = y (x,y are objects)	maps to→	no mapping

4.3 TYPES

All Object-Z built-in sets map directly to Perfect built-in types.

ℕ	maps to→	nat
ℤ	maps to→	int
ℝ	maps to→	real
𝔹	maps to→	bool
Char	maps to→	char

4.4 DECLARATIONS

Single and multiple variables are declared in the same way in both languages.

x : T	maps to→	x : T
x,y : T	maps to→	x,y : T

4.5 LITERALS

4.5.1 Character literals

Perfect uses single forward quotes to declare character literals, so:

'x'	maps to→	`x`

4.5.2 Declared literals

When using literals that have been declared as freetype constants, they must be postfixed with '@className' in Perfect, so

pm´ = gordon becomes pm´ = gordon@Politician

i.e.:

declaredLiteral	maps to→	declaredLiteral@className

4.5.3 Current object literal

The current object in Object-Z is identified by the keyword **self**. Perfect also has this keyword:

self	maps to→	self

4.6 SETS

Perfect contains built-in collection types **set of**, **bag of**, **pair of**, **triple of**, **map of** and **seq of**, allowing a 1-1 mapping of the majority of Object-Z set types:

ℙ X	maps to→	set of X
ℙ(X × Y), X↔Y	maps to→	set of(pair of(X,Y))
X↔Y↔Z	maps to →	set of(pair of(pair of(X,Y),Z)) etc.
X→Y	maps to→	map of (X->Y)
X↦Y	map to →	map of (X->Y)
seq X	maps to→	seq of X

4.6.1 Set enumeration

An expression enumerating a set in Object-Z, such as: x = {1,2,3} translates into Perfect as:

x = set of int{1,2,3}

So, it is necessary to know the type of *x* in the Object-Z, so that it can be added to the Perfect code. In particular, the type cannot be determined from the elements in the enumeration, since it may be an empty set.

x = {a,b,c}	maps to→	x = set of type_of_x{a,b,c}
x = {}	maps to→	x = set of type_of_x{}

4.6.2 Set expressions

The generalized Object-Z expressions:

{declarations | predicate} and {declarations | predicate • expression}

define a set as a collection of values obeying particular constraints. , e.g.:

$$\{z: \mathbb{Z} \mid z > 3\}$$

which translates into Perfect as:

```
those z: int :- z > 3
```

However, this use of the "those" form is only valid as a constrained type class definition. Within any other expression "those" may only operate on an existing collection, not a type. This is because sets, as opposed to types, in Perfect must be finite. This means that an Object-Z expression which uses set abstraction, { }, to define a set "on the fly", such as:

$$x = \#\{z: \mathbb{Z} \mid z > 0 \land z < a * b / 5\}$$

cannot be mapped into Perfect.

name == {decs | pred}	maps to→	**class** name ^= **those** decs **:-** pred**;**
name == {decs | pred • expr}	maps to→	no mapping
exp ({decs | pred})	maps to→	no mapping
exp ({decs | pred • expr})	maps to→	no mapping

The expression a .. b also defines a set in Object Z. The variables a and b must be integers or natural numbers and the members of the resulting set are the continous numbers from a up to b inclusive, or an empty set if a > b. Perfect has a similar expression, but a .. b returns a sequence, so we must take the range of this sequence to get the equivalent result to the Object-Z expression (see 4.6.3.1 Domain and range).

a .. b	maps to→	(a .. b).ran

4.6.3 Relations

4.6.3.1 Domain and range

The Perfect types **bag of**, **map of** and **seq of** have built-in methods *dom* (map and seq only) and *ran* which return the appropriate set.

dom r	maps to→	r.dom
ran r	maps to→	r.ran

4.6.4 Sequences

A sequence has a special enumeration syntax in Object-Z, which maps to a Perfect seq of constructor just as for sets.

x = ⟨a,b,c⟩	maps to→	x = seq of type_of_x{a,b,c}

4.7 PREDICATE LOGIC

4.7.1 Quantified expressions

Universally quantified statements such as:

$$\forall x: \mathbb{N}; y: \mathbb{N} \bullet y = x * x \Rightarrow y > x$$

and existentially quantified statements such as:

$$\exists n : \mathbb{N} \bullet n > 5$$

translate directly into Perfect as:

```
forall x: nat, y : nat :- y = x * x ==> y > x
exists n : nat :- n > 5
```

∀decs • pred	maps to→	forall decs :- pred
∃decs • pred	maps to→	exists decs :- pred

4.7.2 Bound variable declarations

A variable declared within a quantified logical expression can belong to a type (given or user-defined), or it can belong to a collection, as in:

∀c:cards • c.limit = commonlimit

from the CreditCard example (see page 14). Perfect also allows this second kind of declaration, but replaces the **' : '** with a double colon, **' :: '**.

∀x:Type	maps to→	forall x : Type
∀x:collection	maps to→	forall x :: collection

4.8 FREETYPES

A freetype can be declared in Object-Z by listing its elements explicitly in a declaration such as this:

Grocery ::= apple | coke | crisps | banana | curry_sauce

The equivalent in Perfect is a class generated by an enumeration:

class Good ^= **enum** apple, coke, crisps, banana, curry_sauce **end**;

name ::= elem {| elem }*	maps to→	class name ^= enum elem {, elem}* end;

4.9 CLASSES

An Object-Z class schema is equivalent to a Perfect abstract class declaration.

ClassName … _____	maps to→	class ClassName ^= … end;

4.9.1 Visibility list

The Object-Z visibility list has no direct translation into Perfect, but the visibility of a feature affects the section of the Perfect class in which it appears. The semantics of Object-Z visibility are illustrated in Figure 7. In the diagram, Class B inherits from Class A, while Class C is independent. Class A has four features, variable var1 and operation op1, which are not visible, and variable var2 and operation op2 which are visible.

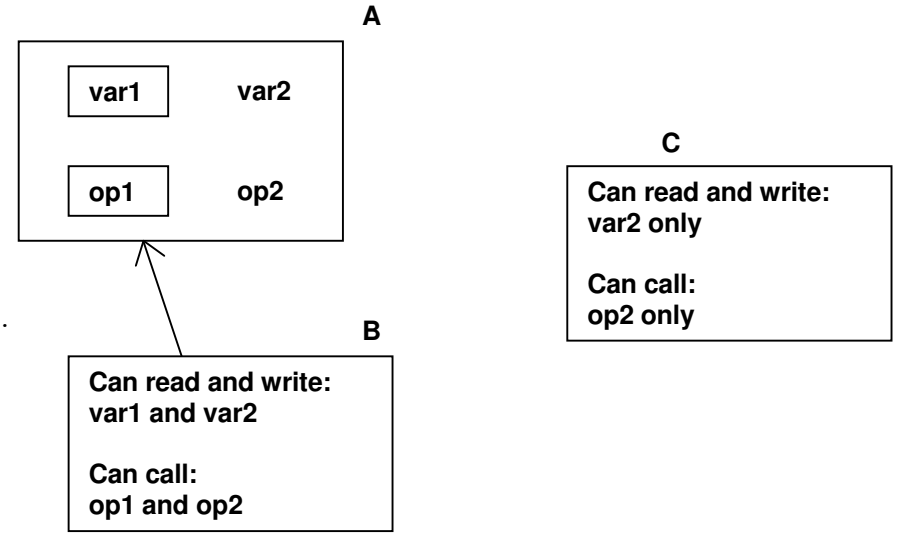


Figure 7. Semantics of Object-Z visibility.

An instance of Class B, as a child of Class A, inherits all of A's features, meaning that its own operations can modify both variables declared in A and call both operations. Inheritance in Object-Z is total in that there is no notion of inherited, but not accessible features as in some OO programming languages. An instance of Class C can only access the visible features (var2 and op2) of an A object. In the case of var2 this is read-write access though, since only constants are immutable. These are the semantics that the mapping into Perfect should replicate. To do so, all visible features map to declarations (or redeclarations, see below) in the interface section, and all non-visible features to the confined section. The confined section allows access by all child classes.

However, this is not the full story. To be able to generate preconditions for composite operations (see page 43) it is necessary to make all variables accessible (to be read) by all classes in Perfect. This exceeds the access to non-visible variables that Object-Z affords, but it is necessary to make the

composite operations mappable. Giving function access means that the variable can only be read, not altered, by classes that would not have access to it in Object-Z. Nevertheless, any illegal accessing of the variable that is present in the Object-Z spec will not be picked up by Perfect Developer. Therefore, a system that implements the mapping will need to independently check that variable accesses are legal.

visible feature	maps to→	interface declaration
non-visible feature	maps to→	interface declaration (not equivalent to Object-Z)

4.9.2 Constants schema (axiomatic schema)

As explained on page 6 (2.2.2 Constants definition) Object-Z constants represent an unchanging value, but do not have to be declared as a literal value that all instances of the class share. There is no equivalent to this in Perfect. A constant in Perfect is an alias for a particular value, e.g.

```
const four: nat ^= 4;
```

and cannot be set through a constructor or any similar mechanism. So, the closest we can come to the Object-Z situation is to declare a variable in place of the constant, and restrict opportunities to change it to a minimum. This can be done as far as changes by other classes are concerned, but not for changes within the operations of the constant's class itself. So, as with access to non-visible attributes, an implementation of the mapping should verify that constants are not modified independently of any checks made by Perfect Developer.

4.9.2.1 Constant declarations

The Object-Z constant declaration:

$$limit: \mathbb{N}$$

is mapped to an abstract variable declaration in Perfect. The constant is also redeclared as a function in the interface section of the Perfect class. This allows the constant to be accessed by other classes, but not changed.

As we are mapping the constant into an "immutable variable" it must also be initialized in the class constructor.

x : Type	maps to→	abstract var x : Type;
	plus→	interface function x;
	plus→	build{…, !x: Type,…}

4.9.2.2 Constant schema predicate

Logic statements in the constant schema predicate map into class invariants and preconditions on the class constructor, e.g.:

limit ∈ {1000, 2000, 5000} becomes

abstract

invariant limit **in** set of nat{1000, 2000, 5000};

build{!limit:nat,…}

pre limit **in** set of nat{1000, 2000, 5000}…

expression	maps to→	abstract expression;
	plus→	build{…} pre expression;

4.9.3 State schema

4.9.3.1 Primary variables

A non-visible state schema variable (attribute) maps in the same way as a constant. A visible attribute, however, requires an additional element because visible Object-Z attributes can be modified from outside the class in which they are declared.

The Perfect mechanism to give read-write access to a variable is to redeclare it as an interface selector. However, this creates additional problems. Any invariants that restrict the value of the variable are automatically invalidated by declaring it as a selector, which allows any value to be assigned. In Object-Z it is assumed that the specification covers the entire system, and includes all operations that could modify a variable. If none of these declared operations violate the invariant, then there is no problem. Perfect, on the other hand, envisages other as yet unknown client classes that have the *potential* to assign

an illegal value to the variable. In Perfect a class must ensure that its invariants are never broken, not leave it up to the clients.

So, to allow a variable to be changed by other classes we have to include a schema to set its value safely. The schema must have any invariants that depend on the value of the attribute in question as preconditions, and will have one parameter – the new value to assign to the attribute. Thus, this particular component of the mapped class requires quite some effort to construct. The class invariants must be collected, analysed to see if they are relevant, and re-engineered to apply to the new parameter's identifier.

x:Type exp(x)	maps to→	abstract var x : Type; invariant exp(x);
	plus→	interface function x;
	plus→	build{…, !x: Type,…} pre exp(x);
	plus→ (if x visible)	schema !set_x(param:Type) pre exp(param) post x! = param;

Functions

Functions in Object-Z map to "map of" collections in Perfect. Object-Z includes several types of function, though, each of which introduce different implicit conditions on the mapping. Since Perfect only has one mapping type, these implicit conditions must be added as extra class invariants, and constructor and "set schema" preconditions.

partial function, f:X↔Y	maps to→	var f: map of (X -> Y);
total function, f:X → Y	maps to→	var f: map of (X -> Y); invariant forall x:X :- x in f;
injection, f:X ⤔ Y	maps to→	var f: map of (X -> Y); invariant forall x1,x2::f :- x1 ~= x2 ==> f[x1] ~= f[x2]
surjection, f: X↛ Y	maps to→	var f: map of (X -> Y); invariant forall y:Y :- y in f.ran

4.9.3.2 Secondary variables

Secondary variables cannot be mapped in the same way as primary ones. Secondary variables in Object-Z are defined in terms of the class's primary variables; they do not vary independently (see page 10). As such, mapping to an abstract variable in Perfect would be, at the least, rather complex. Any operation that changed the primary variables of a class would have to be defined to also update the secondary variables, or they would get "out of synch" and violate class invariants related to them. This would complicate and bloat the Perfect class.

As can be seen, secondary variables are not really variables at all, they are a convenient shorthand for pieces of information constructed from the real state variables. Which is an ineligant way of saying that they are *functions* of the primary variables. So, a function in Perfect is a better choice for a mapping. Returning to the *TwoCards* example from Chapter 2, secondary variable *totalbal*:

c_1, c_2 : CreditCard

Δ

totalbal : ℤ

$c_1 \neq c_2$

totalbal = c_1.balance + c_2.balance

would map to a function such as:

```
function totalbal : int
    ^= c1.balance + c2.balance;
```

The return type of the function is int, the declared type of *totalbal,* and the definition statement is provided by the Object-Z class invariant which determines *totalbal*. This is a very straightforward example, though. Secondary variables are not always defined by a simple assignment, or even by a single statement. The mapped function must be generalized to this form:

```
function totalbal : int
    satisfy result = c1.balance + c2.balance;
```

which is be defined by providing a list of conditions that the predefined identifier, *result*, must satisfy. In principle, this provides the required mapping.

The only complication is that equality statements, such as result = c1.balance + c2.balance in the example function, are treated as assignments by Perfect, even in a satisfy clause. So, if the secondary variable is mentioned in more than one equality:

```
function totalbal : int
    satisfy result = c1.balance + c2.balance
        & result = available - totallimit;
```

Perfect will consider that *result* is being set twice within the same function. So, we have to rewrite the condition so that *result* only appears once. This can be done with an existential statement as follows:

```
function totalbal : int
    satisfy exists r: int :- r = c1.balance + c2.balance
        & r = available – totallimit & result = r;
```

Finally, we note that any state schema invariant that involves a secondary variable, such as "totalbal = c1.balance + c2.balance" becomes part of the function definition *only*. It is not needed as a class invariant in Perfect, since the function definition will ensure that it is always satisfied.

Δ sec : Type --- exp(sec)	maps to→	interface function sec : Type satisfy exists temp : Type :- exp(temp) & result = temp;

4.9.4 Initial schema

The initial schema of an Object-Z class contains a predicate section only. The Perfect equivalents of the expressions in this predicate must be added to the precondition of the constructor of the Perfect class, e.g.:

INIT

balance = 0

becomes:

```
build{…}
    pre balance=0 & …;
```

/NIT expression	maps to→	build{…} pre expression;

Where the predicate includes a reference to the initial configuration of an object, e.g. from the *TwoCards* class (see Figure 2):

/NIT

c_1. */NIT*

c_2. */NIT*

the expression ob.INIT must be expanded into the full predicate from the specification of the type of "ob". So, the TwoCards example maps to:

```
build{…}
    pre c1.balance=0, c2.balance=0 …;
```

To express the general case we first define the following functions:

Definitions

- type(ob) maps an object to its type
- init_pred(type) returns the Perfect translation of the initial schema predicate of the class "type"
- vars(type) returns a sequence containing the names of the attributes of class "type"
- vars(ob) returns a sequence containing the names of the attributes of object "ob" (i.e. each one will be of the form "ob.var_name").

and,

- the rename operator [/] which takes whatever expression precedes it and substitutes the names or list of names on the left of the '/' for the name or list of names on the right, .e.g.:

"x = 5" [y/x] = "y = 5"

Then:

ob.INIT	maps to→	init_pred(type(ob)) [vars(ob) / vars(type(ob))]

4.9.5 Operation schemas

An Object-Z operation schema contains three basic elements as described in section 2.2.5 Operation schemas (page 8). These elements are the delta list, the declarations of communication variables, and the predicate. For the purposes of mapping, we shall further divide the declarations into inputs and outputs, and the predicate into a *precondition* and a *postcondition*.

4.9.5.1 Extracting preconditions and postconditions

The postcondition of an operation is defined as:

- the set of expressions or sub-expressions within the predicate that describe the state after the operation has been performed, i.e. expressions which contain primed and/or output variables (e.g. x', var!)

And, in this context, a sub-expression is defined as:

- a FOPL statement that can be generated by taking the predicate and recursively dividing it at the principle logical connective, but only if the connective is a conjunction or implied conjunction (line breaks are implied conjunctions).

Thus, the postcondition of this predicate:

$$x = 0 \land x' = 5$$

is:

$$x' = 5$$

but the postcondition of this predicate:

$$(x = 0 \land x' = 5) \lor (x > 0 \land x' = 10)$$

is: $(x = 0 \land x' = 5) \lor (x > 0 \land x' = 10)$

because the principal connective is '∨', and so the expression cannot be subdivided.

The precondition of an operation is defined as:

- the set of expressions or sub-expressions in the predicate that describe states before the operation has been performed that make the operation applicable, i.e. do not contain any primed or output variables.

Where a sub-expression is defined as for a postcondition, plus the following:

- a statement formed by removing the postcondition parts from any disjunction arising from the initial subdivision, not containing an implication or equivalence.

So, to return to the example, the precondition of the predicate:

$$(x = 0 \land x' = 5) \lor (x > 0 \land x' = 10)$$

is: $x = 0 \lor x > 0$

It should be noted from the example that:

$$\text{precondition} \cap \text{postcondition} = \varnothing$$

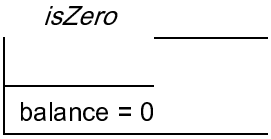
where precondition and postcondition are the sets defined above, is not always true.

4.9.5.2 No-change operations

An Object-Z schema maps to either a function or a schema in Perfect, depending on whether it modifies any values (object variables or output parameters). If the schema has no outputs and an empty delta list, then it

makes no changes and must be mapped to a function (a schema in Perfect always modifies some variable).

The reader may be wondering what would be the point of an operation that makes no changes – it would leave the system in the same state as if no operation had been applied. This is correct, but the point lies in whether or not the operation is applicable. A no-change operation can still have a predicate, and this entire predicate would be a *precondition* (try applying the definitions above). When the precondition of an operation is false, the operation is inapplicable, and if the operation is combined with others into a composite operation expression (see page 11) it will determine when the whole expression is applicable. This allows no-change operations to play the role of conditionals in Object-Z operation expressions. For example, if *CreditCard* defined an operation *isZero* as follows:



then the composite operation *depositIfZero* in *TwoCards:*

$$depositIfZero \triangleq c_1.isZero \land c_1.deposit$$

would add money to *CreditCard* object c_1, only if its balance were 0.

So, having given no-change operations a raison d'être, we continue with mapping them. Although the Object-Z operation has no outputs, a Perfect function must have an output. The simplest solution seems to be to map to a function that returns a boolean value (true or false). So, the *isZero* operation becomes:

```
function isZero : bool
    ^= balance = 0;
```

As the example shows, the function is declared to be "defined as" (^=) the predicate of the operation. Thus, the operation returns true whenever the predicate is true.

Any inputs to the operation are mapped to parameters of the function:

```
function isValue(val:int) : bool
    ^= balance = val;
```

Definitions:

- in_vars(op) and out_vars(op) are functions that return the set of names of the input and output variables of operation *op*
- in_var_decs(op) and out_var_decs(op) are functions that return the Perfect declarations of the input variables of operation *op*
- deltalist(op) is a function that returns the list of names of variables in the deltalist of operation *op*
- postcond(op) and precond(op) are functions that return the Perfect expressions equivalent to the postcondition and precondition of operation *op*

op	if (deltalist(op) = ∅ ∧ out_vars(op) = ∅) maps to→	function op (in_var_decs(op)) : bool ^= precond(op);

Evaluation

It must be acknowledged that this mapping for no-change operation schemas does not provide a direct equivalence with Object-Z. The Perfect function does not capture the idea of the operation being inapplicable since it will always execute, and it does return a value, true or false. However, since Perfect functions are guaranteed not to alter the data of their object, it does leave the system in the same state, and it does provide a method by which the truth value of the Object-Z schema predicate may be determined. Therefore, this solution is presented as being sound and complete.

4.9.5.3 Change operations

As hinted at above, an operation that either changes object variables or has an output maps to a Perfect schema. A Perfect schema has parameters, a

precondition and a postcondition, and these more-or-less equate to their Object-Z equivalents.

The schema's parameters and precondition are simply the Object-Z operation's parameters and precondition expressions mapped into Perfect using the mappings given already.

The schema's postcondition maps to a Perfect "change…satisfy…" expression. There are several alternative forms of postcondition in Perfect, but the "change…satisfy…" form is the most general. Object-Z postconditions can sometimes be expressed in quite non-specific terms, e.g. an operation to find the lowest value in a set of integers could have a postcondition such as:

$$(res! \in the\_set) \land \forall i:the\_set \bullet (i >= res!)$$

This does not assign a specific value to res! but it does capture the state in which res! must be equal to the lowest value in the_set, whatever that may be. Such a postcondition cannot be expressed as any of the other forms of Perfect postcondition, since they are all designed to expect explicit assignment of values.

The change part of the change…satisfy… expression is a list of all variables that the schema declares it may change. This equates to the deltalist of the Object-Z operation, plus any output variables. So, the full mapping is:

op	if (deltalist(op) ≠ ∅) maps to→	schema !op (var_decs(op)) pre precond(op) post change deltalist(op), out_vars(op) satisfy postcond(op);
	if (deltalist(op) = ∅ ∧ out_vars(op) ≠ ∅) maps to→	schema op (var_decs(op)) pre precond(op) post change deltalist(op), out_vars(op) satisfy postcond(op);

4.9.6 Composite operations

4.9.6.1 Operation promotion

Consider operation schema *setX* in class *Part:*

setX

Δ(x) a?: Z
a? > 0 edits < max x′ = a?

If class *System* contains an instance of *Part*, p, then *System* can define a new operation expression which calls p.*setX*:

$$setP \triangleq p.setX$$

where *setP* "promotes" *setX* to be an operation of *System*. To promote the equivalent Perfect setX schema:

```
schema !setX(a: int)
        pre a > 0 & edits < max
        post change x satisfy x′ = a;
```

we use the schema call form of postcondition, and restate any preconditions:

```
schema !setP(a: int)
        pre a > 0 & p.edits < p.max
        post p!setX(a);
```

The precondition must be repeated because *setP* should be inapplicable when *setX* is inapplicable. Any member variables of *Part* appearing in the precondition must be renamed in the promoted precondition to identify them as belonging to object p. (Note that the variables must have been redeclared as Perfect interface functions for the precondition to access them legally. This is the reason for all variables to have function access, as discussed on page 32.) The postcondition of *setP* invokes *setX* by calling the schema directly. This reflects the object-oriented semantics of Object-Z. The only alternative

to this would be to promote the individual postcondition statements, but this would require write-access to object p's variables, breaking the object-oriented encapsulation. Finally, where an Object-Z operation exchanges inputs and outputs with the environment, so too does the promoted operation (Duke & Rose[19] p 15), so *setP* has identical parameters to *setX*.

Using the definitions on pages 37 and 41, promoting operation op to newop by invoking op on object ob:

$$newop \triangleq ob.op$$

maps into Perfect by creating schema newop as follows:

newop ≙ ob.op	maps to→	schema newop
deltalist(newop) = deltalist(op) [vars(ob)/vars(type(ob))] in_vars(newop) = in_vars(op) out_vars(newop) = out_vars(op) precond(newop) = precond(op) [vars(ob)/vars(type(ob))] postcond(newop) = ob.op(in_var_decs(op),out_var_decs(op))		

4.9.6.2 Operation conjunction

Object-Z operation conjunction:

$$newop \triangleq op1 \wedge op2$$

produces a composite operation that is equivalent to performing both component operations in parallel, and where any matching input or output parameters are equated.

Matching input variables

If *op1* and *op2* are defined as:

op1

Δ(x) num?: Z
num? < limit x´ = num?

op2

Δ(y) num?: Z
y´ = num?

then *newOp* will set both x and y to the same value, assuming the value meets the condition in *op1*. To conjoin the two equivalent Perfect schemas:

```
schema !op1(num: int)
        pre num < limit
        post change x satisfy x´ = num;

schema !op2(num: int)
        post change y satisfy y´ = num;
```

we conjoin the preconditions and the postconditions:

```
schema !newop(num: int)
        pre num < limit
        post !op1(num) & !op2(num);
```

The conjunction operator in Perfect, when applied to schema calls like this, means that the schemas will execute in parallel, thus providing the required equivalence with the Object-Z composition.

The result is different if there are matching output variables (see below).

The example uses operations of the current class to keep things simple, but it should be noted that the conjunction is of promoted operations, and that it is the promoted preconditions and postconditions that are conjoined.

There are two complications. An input and an output with matching base names (e.g. num? and num!) are distinct variables under conjunction. Such names pose a problem in Perfect because the form num? is an illegal identifier, and the forms num and num! are considered to be related by Perfect (num! is the value of num after it is changed by a schema). Therefore, if two conjoined operations include parameters with the same base name, but they are not both inputs or both outputs, it is necessary to rename one to a name that does not exist in the current scope.

Secondly, Perfect does not permit "parallel" changes to the same object, so a postcondition such as:

```
post obj!op1 & obj!op2;
```

causes a compilation error in Perfect. The changes can be made in sequence using the **then** operator, however, this does not reflect the Object-Z semantics and forces one operation to precede the other. In other words, if the operations are performed left-to-right, op2 will not begin from the same state as op1, but from an intermediate state where ojb has already been modified by op1. If op2 uses the value of ("reads from") any variables that op1 has changed, then the final state produced by Perfect will not be equivalent to the one produced by Object-Z. This means that we must exclude any conjunctions of the form:

$$\text{newop} \triangleq \text{a.op1} \wedge \text{b.op2}$$

when a=b≠self ∧ (deltalist(op1) ∩ unprimed(postcond(op2)) ≠ ∅). Where unprimed(exp) is the set of state variables that appear unprimed in the expression exp.

Given this condition, conjoining Object-Z operations a.op1 with b.op2 to produce newop:

$$\text{newop} \triangleq \text{a.op1} \wedge \text{b.op2}$$

where,

Definitions

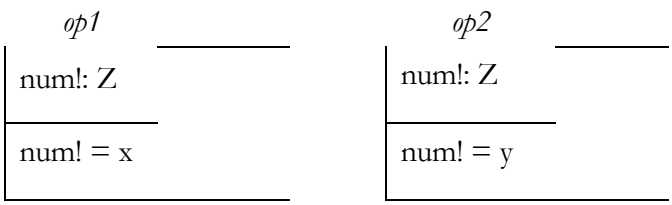
- promoted operations a.op1 and b.op2 individually map into Perfect as schemas newop1 and newop2
- function o_var_match(a,b) returns a list of output variables of operations a and b whose names match input variables of the other operation
- freeid is a list of all valid Perfect ids that are not in the current scope

maps into Perfect by creating schema newop as follows:

if out_vars(op1) ∩ out_vars(op2) = ∅: **newop ≙ a.op1 ∧ b.op2**	**maps to→**	**schema newop**
such that: deltalist(newop) = deltalist(newop1) ∪ deltalist(newop2) in_vars(newop) = in_vars(newop1) ∪ in_vars(newop2) out_vars(newop) = (out_vars(newop1) ∪ out_vars(newop2)) [freeid/o_var_match(newop1,newop2)] precond(newop) = precond(newop1) '&' precond(newop2)		
if a ≠ b ∨ a = b = self: postcond(newop) = (postcond(newop1) **'&'** postcond(newop2)) [freeid/o_var_match(newop1,newop2)]		
if a = b ∧ (deltalist(op1) ∩ unprimed(postcond(op2)) = ∅): postcond(newop) = (postcond(newop1) **'then'** postcond(newop2)) [freeid/o_var_match(newop1,newop2)]		
if a = b ∧ (deltalist(op1) ∩ unprimed(postcond(op2)) ≠ ∅): no mapping		

Matching output variables

If the two operations have matching output variables, such as:



there is an extra consideration. The mapping presented above would give this Perfect schema for the conjunction (newop ≙ a.op1 ∧ b.op2):

```
schema newop(num!:out int)
    post a.op1(num!) & b.op2(num!);
```

However, Object-Z also imposes the implicit precondition that the quantities equated in the outputs must be equal before the operation for the composite to be applicable. It should be apparent that the operation would be internally inconsistent were this not the case. To express this extra precondition in Perfect we use an existentially quantified expression to test whether a valid output is possible. This expression declares a temporary variable, substitutes it for the output variable in both postconditions, and conjoins them. The result for the example would be:

```
pre exists i:int :- i = a.x & i = b.y
```

This particular condition could have been expressed more simply as:

```
pre a.x = b.y
```

however, other forms of postcondition not involving explicit assignment, such as the "findMin" example on page 41, cannot simply be equated, and so the former solution is preferred. So, the example operation is now:

```
schema newop(num!:out int)
    pre exists i:int :- i = a.x & i = b.y
    post a.op1(num!) & b.op2(num!);
```

Unfortunately, the restrictions on parallel changes in Perfect also affect this kind of conjunction. Even though op1 and op2 are being applied to different objects, both operations still modify num!, which Perfect disallows. Therefore, the final version of this mapping introduces a temporary parameter to pass to the second operation:

```
schema newop(num!:out int)
    pre exists i:int :- i = a.x & i = b.y
    post (var temp:int; a.op1(num!) & b.op2(temp!));
```

The value of temp can be safely thrown away since the precondition ensures that temp! will be equal to num!. It is still important to include op2 in the postcondition, however, since it may make other changes to object b, so omitting it altogether would result in a system state that would not be equivalent to the Object-Z operation.

if out_vars(op1) ∩ out_vars(op2) = out_both ≠ ∅: **newop ≙ a.op1 ∧ b.op2**	**maps to→**	**schema newop**
such that: deltalist(newop) = deltalist(newop1) ∪ deltalist(newop2) in_vars(newop) = in_vars(newop1) ∪ in_vars(newop2) out_vars(newop) = (out_vars(newop1) ∪ out_vars(newop2)) [freeid/o_var_match(newop1,newop2)] precond(newop) = 'exists' (∀v:out_both • i:type(v)) :- (postcond(op1) '&' postcond(op2)) [i/v]) '&' precond(newop1) '&' precond(newop2)		
if a ≠ b ∨ a = b = self: postcond(newop) = (∀v:out_vars(newop2) • 'var' i:type(v)) (postcond(newop1) **'&'** postcond(newop2)[i/out_vars(newop2)]) [freeid/o_var_match(newop1,newop2)]		
if a = b ∧ (deltalist(op1) ∩ unprimed(postcond(op2)) = ∅): postcond(newop) = (postcond(newop1) **'then'** postcond(newop2)) [freeid/o_var_match(newop1,newop2)]		
if a = b ∧ (deltalist(op1) ∩ unprimed(postcond(op2)) ≠ ∅): no mapping		

4.9.6.3 Choice composition

Choice composition (see page 12) does not involve any parameter pairing or renaming, since two Object-Z operations that are joined by a choice expression must have identical communication variables (Duke & Rose[19], p.17). If this condition is satisfied, one or other operation will run, depending on which is applicable. These semantics are conveniently matched by the opaque conditional expression in Perfect. The general form of a Perfect conditional is:

([test1]: action1, [test2]: action2, …)

which behaves as an "if…else…" clause with the first action whose guarding test evaluates to true being executed. However, adding the keyword "opaque"

before the first guard changes the semantics to that of non-deterministic choice. That is to say, if more than one test evaluates to true any one of the respective actions may be "chosen" to be executed.

So, if the following operation is defined:

Dec

Δ(my_var) x?: Z
my_var > x my_var´ = my_var – x

≡

schema !dec(x:int)
pre my_var > x
post change myvar
satisfy myvar´ = my_var - x

in the class of objects a and b, and operation newop is declared as:

newop ≙ a.dec [ ] b.dec

then Perfect schema newop will be:

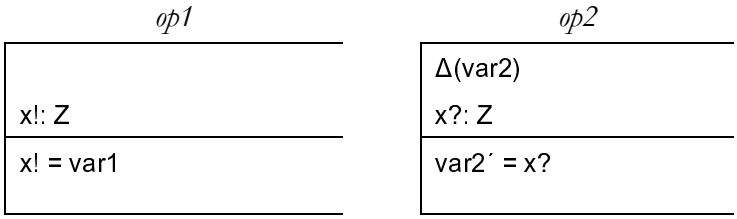
schema !newop(x:int)
pre a.myvar > x | b.my_var > x
post (**opaque** [a.my_var > x]: a!dec, [b.my_var > x]: b!dec);

newop ≙ a.op1 [ ] b.op2	**maps to→**	**schema newop**
such that: deltalist(newop) = deltalist(newop1) ∪ deltalist(newop2) in_vars(newop) = in_vars(newop1) = in_vars(newop2) out_vars(newop) = out_vars(newop1) = out_vars(newop2) precond(newop) = precond(newop1) ' | ' precond(newop2) postcond(newop) = '(opaque ['precond(newop1)']:' postcond(newop1) ' , [' precond(newop2) ']:' postcond(newop2) ',)'		

4.9.6.4 Parallel composition

The essence of parallel composition in Object-Z is the fact that the output of one operation is equated with the input of another. This produces so-called

"inter-object communication" (see page 12). Other than this important point, parallel composition is just like conjunction. The communication has consequences for the Perfect mapping though. A naïve mapping of this example:



comp ≙ a.op1 || b.op2

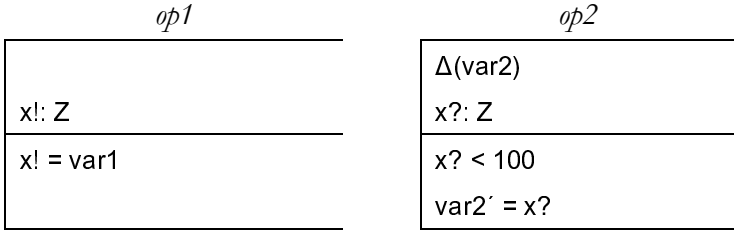
would give this schema:

```
schema !comp
        post (var x:int ; a.op1(x!) & b.op2(x));
```

mapping the paired communication variable into a temporary variable within the postcondition. This reproduces the Object-Z semantics of hiding the parameter from the environment, and seems to take the output of op1 and pass it into op2. Unfortunately, this will not compile. Using the conjunction operator in the postcondition causes the schemas to be executed in parallel, but Perfect will insist that op2 cannot be run until op1 completes, because until this point x has no value. There is no way around this in Perfect, and so we can only map to schemas using the "then" sequential operator. As discussed in 4.9.6.2 Operation conjunction, this is a limited mapping since it will not be equivalent to the Object-Z operation if op1 changes any variables that op2 depends on.

Preconditions

If we expand the example to include a precondition in op2:



we have another problem. The precondition involves the value of the input, which has now become a hidden communication parameter. Since this variable only exists within the operation we cannot state any conditions about its value before the operation is applied. However, the precondition still applies to the, parallel, Object-Z operation. If it is not satisfied the whole operation should be inapplicable. Of course, the input to op2 depends on the output from op1; in other words, it depends on the postcondition of op1. What we require is that the postcondition of op1 satisfies the precondition of op2. This can be expressed in Perfect, using an existential expression as in the conjunction schema with equated outputs (see page 47). The schema now becomes:

schema !comp
pre **exists x :int :- x = a.var1 & x < 100**
post (var x:int ; a.op1(x!) then b.op2(x));

Two-way communication

There is a further restriction on this part of the mapping, namely that we cannot map an operation involving two-way communication. This happens in Object-Z when both operations have inputs that are paired with a complementary output. Since we cannot put both operations on the left of "then", this cannot be expressed in Perfect.

newop ≙ a.op1 || b.op2	**maps to→**	**schema newop**
Definitions hidden_vars = {(in_vars(newop1) ∩ out_vars(newop2)) ∪ (in_vars(newop1) ∩ out_vars(newop2))}		
deltalist(newop) = {deltalist(newop1) ∪ deltalist(newop2)} in_vars(newop) = {in_vars(newop1) ∪ in_vars(newop2)} \ hidden_vars out_vars(newop) = {out_vars(newop1) ∪ out_vars(newop2)} \ hidden_vars precond(newop) = precond(newop1) '&' 'exists' (∀v:hidden_vars • v:type(v)) :- postcond(op1) '&' precond(op2))		

if a ≠ b ∨ (a = b ∧ {deltalist(op1) ∩ unprimed(postcond(op2)} = ∅):
postcond(newop) = '(' (∀ v:hidden_vars• 'var' v ':' type(v) ';') (postcond(newop1) **'then'** postcond(newop2)) ')'
if a = b ∧ (deltalist(op1) ∩ unprimed(postcond(op2)) ≠ ∅):
no mapping
if {in_vars(newop1) ∩ out_vars(newop2)} ≠ ∅ ∧ {in_vars(newop1) ∩ out_vars(newop2)} ≠ ∅:
no mapping

4.9.6.5 Sequential composition

Sequential composition of two operations results in a composite that is equivalent to performing the first operation followed by the second. Like parallel composition, inter-object communication is possible (Duke & Rose[19], p. 189), but now the order of the operations is determined by the composition expression, so two-way communication cannot occur. In terms of the mapping this means that sequential composition is equivalent to the non-null case of the parallel composition mapping.

newop ≙ a.op1 § b.op2	**maps to→**	**schema newop**
Definitions hidden_vars = {(in_vars(newop1) ∩ out_vars(newop2)) ∪ (in_vars(newop1) ∩ out_vars(newop2))}		
deltalist(newop) = {deltalist(newop1) ∪ deltalist(newop2)} in_vars(newop) = {in_vars(newop1) ∪ in_vars(newop2)} \ hidden_vars out_vars(newop) = {out_vars(newop1) ∪ out_vars(newop2)} \ hidden_vars precond(newop) = precond(newop1) '&' 'exists' (∀v:hidden_vars • v:type(v)) :- postcond(op1) '&' precond(op2))		
postcond(newop) = '(' (∀ v:hidden_vars• 'var' v ':' type(v) ';') (postcond(newop1) **'then'** postcond(newop2)) ')'		

Or, to put this another way, a parallel composition can only be mapped if the parallel operator could be replaced by the sequential operator without changing the meaning of the expression. i.e. parallel composition is redundant in this Object-Z to Perfect mapping.

4.9.6.6 Environment enrichment

The final composite operator is the environment enrichment operator, ' • '. This was seen in the CreditCard example of Chapter 2 (see page 14), in the definition of operation *withdraw*:

$$withdraw \triangleq [card?:cards] \bullet card?.withdraw$$

where the input *card?* is defined, or "selected", in the in-line schema [card?:cards] and then *card?*'s own operation *withdraw* is applied. Since the schema supplying the extra variable(s) in the enrichment must always be evaluated first, ' • ' is non-associative (the convention used is left-association).

Environment enrichment is often regarded differently than the other operators, but the Object-Z syntax rule for operation expressions (Duke & Rose[19], p. 218) makes it clear that ' • ' joins two operation expressions to form a new one just as ∧ or || do. The in-line schema can be thought of as an anonymous operation, with input variables and a precondition. If the input variable is selected from a set, there is an implicit precondition, but the schema may also contain an explicit predicate. This operation is then essentially conjoined to the operation represented by the other operand. The key difference with other composition expressions is that the inputs to the schema "operation" may also appear on the other side of the operator, with operations applied to them.

So, the Perfect equivalents of the operations in the example would be [card?:cards] ≡

```
schema [anon] (card!:CreditCard)
    pre card in cards;
```

and card?.withdraw

```
≡   schema !withdraw (amount: int)
        pre amount ≤ card.balance + card.limit
        post card!withdraw(amount);
```

so, withdraw ≙ [card?:cards] • card?.withdraw

≡ schema !withdraw(card!:CreditCard, amount: int)
pre card in cards & amount ≤ card.balance + card.limit
post card!withdraw(amount);

In general:

newop ≙ op1 • a.op2	**maps to→**	**schema newop**
deltalist(newop) = deltalist(newop2) in_vars(newop) = in_vars(newop1) ∪ in_vars(newop2) out_vars(newop) = out_vars(newop2) precond(newop) = precond(newop1) '&' precond(newop2) postcond(newop) = postcond(newop2)		

4.10 UNMAPPED FEATURES OF OBJECT-Z

All the features included in this mapping have now been described. This leaves parts of the language unmapped: most notably inheritance, but also distributed operation composition and generic classes. Perfect has an inheritance feature, and it can be envisaged that this mapping is feasible. However, it was not possible to include the details of this in this project, so it is left to a future study.

A full evaluation of the completeness of the mapping is given in Chapter 7.

Chapter 5

TOOL DESIGN

5.1 INTRODUCTION

As stated in Chapter 1, the aim of the implementation part of the project was to build a tool that uses the reported mapping to translate Object-Z specifications into Perfect, allowing them to be verified automatically. The steps involved in this process are shown in Figure 8.

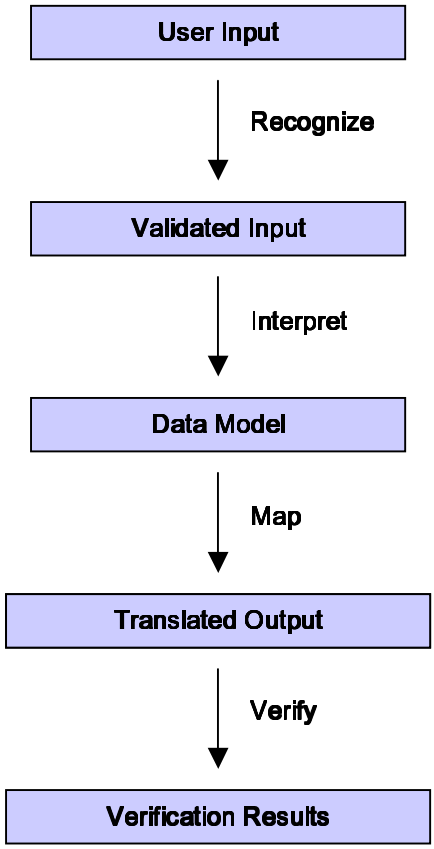


Figure 8. Steps involved in automated Object-Z verification.

The key features of the planned tool that influenced the decisions over the technologies to use were:

- input format should be simple for the user to understand and write
- representation of Object-Z syntax rules should be easy to follow and work with during development
- there should be good separation between the data model and the output generator (or the model and the view to use MVC terminology)

The two approaches considered were XML schema and EBNF grammar.

5.2 XML

There have been several, apparently separate, attempts to define a "Z markup language" (ZML) based on Extensible Markup Language (XML)[23,24,25]. The central idea of these initiatives is to provide a way that Z specifications can be stored and exchanged between different groups, possibly using different tools to work on them. The most frequently referenced one of these is the Community Z Tools (CZT) project (http://czt.sourceforge.net/), which provides an open source toolkit, based on the ISO Z standard, for new Z tools to be based on.

XML is a flexible data exchange format which tags different elements of data so that they can be identified and processed accordingly. Rules determining the valid structure of a data type are defined using either a Document Type Definition (DTD) or the more recent and more generic form, the XML schema. These definitions are written in a similar format to XML documents themselves, with the structure being built up through nesting many layers of language element tags (indented <angle-bracketed> tokens).

Once a DTD or schema has been defined for a data type it can be used as the basis of a parsing tool that will read in data, add the appropriate tags and create a structured XML document for further interpretation. Indeed there are tools such as Sun's JAXB that can generate parser classes straight from the schema, so that the data definition is the only development that is necessary. XML also comes equipped with the related Extensible Stylesheet Language (XSL) which is used to define how XML documents should be displayed (in

web pages or other viewers) and XSL Transformations (XSLT), a language for transforming one XML document into another.

So, the Object-Z to Perfect tool could be built by creating an XML schema defining Object-Z syntax, an XSLT stylesheet to transform an Object-Z XML document into a Perfect XML document, and an XSL stylesheet to output a Perfect file from the Perfect XML. The advantages and disadvantages of this approach were considered.

XML has the advantage that it is a widely used and flexible format. There is good tool support for it and immense amounts written about it. It is a widely understood technology, ideal for data exchange between different groups or companies using separately designed systems that would otherwise find it hard to communicate. However, it has the disadvantage that the definitions and documents are very long and not very human readable. Indeed, the DTDs and schemas downloaded from existing projects such as CZT were quite impenetrable. The first question to answer was how a user might enter valid data. Trying to discern this from the schema was not a simple task. Unfortunately, the projects provided no documentation on the subject either. XML editing tools with schema support such as Microsoft Visual Studio did not solve the problem, and even getting the editor tool downloadable from CZT to work was beyond the wit of the author, despite considerable time and effort. The difficulty experienced trying to solve this, quite fundamental, question counted heavily against XML on the second criterion given above, namely that the syntax rules should be easy to follow and work with during development. It also precluded an answer to the first point regarding user-friendliness. XML should score highly in terms of separation of model and output though, since the translation and output would be confined to the XSLT component.

5.3 EBNF GRAMMAR[26]

Grammars are sets of rules that also define structure in data, specifically in languages, most frequently computer languages. BNF or Backus-Naur Form is a notation that was developed by John Backus and Peter Naur to describe the syntax of the Algol 60 programming language[27]. In a BNF grammar each rule, or *production*, has one or more lists of symbols, with each symbol corresponding to another rule or an actual character sequence (a "terminal"). For example, rules defining alternatives for a variable declaration in Java might be (alternatives are separated by |):

```
declaration := type id ' ; '
                    | type id ' = ' literal ' ; '
type :=             ' int ' | 'double ' | …etc.
```

Phrases are constructed by starting with the top-level rule, and replacing the subrule symbols in it by one of their own symbol lists. Once the phrase has no more rule symbols to replace, i.e. it only contains terminals, it constitutes a legal piece, or *sentence*, of the language. Thus, the language is defined as the set of all sentences that can be produced in this way. Language recognition is then a similar process where the choice of which form of the rule should be used is restricted by the input string.

EBNF, or Extended BNF, includes three operators that make the rules more concise and easier to read:

- ? : which means that the symbol (or group of symbols in parentheses) to the left of the operator is optional (it can appear zero or one times)
- * : which means that something can be repeated any number of times (and possibly be skipped altogether)
- + : which means that something can appear one or more times

BNF and EBNF are very concise ways to unambiguously define a language, making the grammars easily understandable and simple to work with. This also means that, like XML, recognizers and parsers can be generated directly from a grammar, and tools to do this have long been in existence. One such tool is ANTLR (Another Tool for Language Recognition), an open source, parser generator that produces parser code in several languages from an EBNF grammar (http://www.antlr.org/). After recommendation from Vanessa Ho Von, who used ANTLR in her recently completed project "An environment for mapping Object-Z specification into Java/JML annotated code"[11], and Dr Alessandra Russo, ANTLR was investigated as a second language definition option.

5.3.1 ANTLR

ANTLR, in fact, produces three types of output file from a grammar: a lexer, a parser and a tree parser. The lexer transforms the raw character input into grouped, multicharacter "tokens" (the terminals of the EBNF description).

The token stream produced by the lexer is then read by the parser, which either executes actions immediately (e.g. prints some output specified within its rules), or transforms the tokens again into a more structured form called an Abstract Syntax Tree (AST). The AST form is used when the translation being performed is of a more complex nature, requiring the software to "walk" over the input, possibly several times, to produce the correct output. The AST form is much more "cleaned up" compared with the token stream produced by the lexer, and easier to navigate. An example AST of a simplified Object-Z specification, produced by ANTLR is shown in Figure 9.

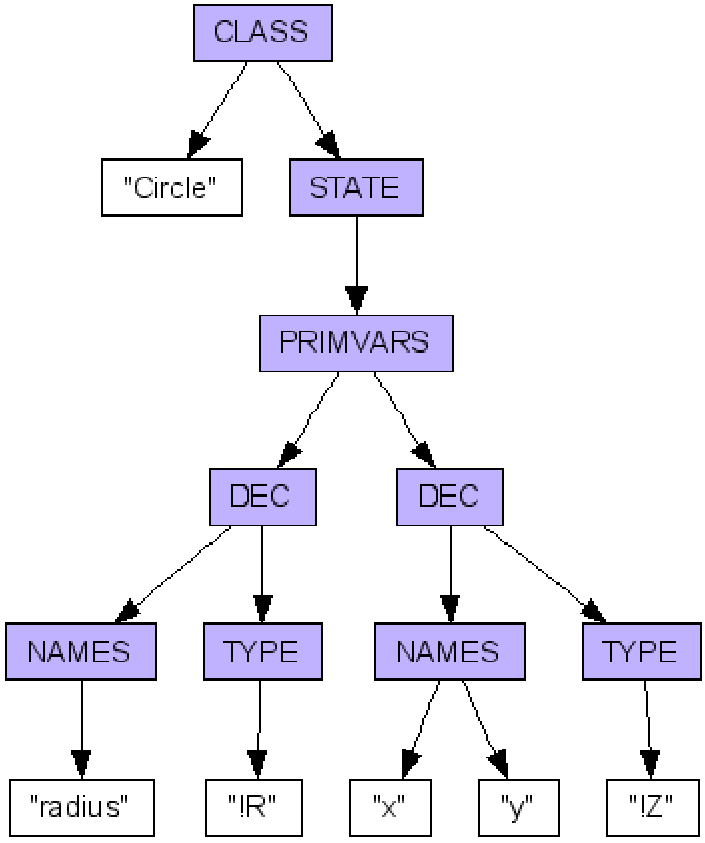


Figure 9. Tree form of a simplified Object-Z specification of a circle. Language concept "imaginary tokens" are shaded, terminal tokens are white.

The tree reflects the hierarchy of the Object-Z (see Figure 10). The shaded nodes in the figure, representing the abstract language concepts, are "imaginary tokens" inserted by the parser, and the open boxes are the

terminals. A more detailed description of AST construction is given on page 71.

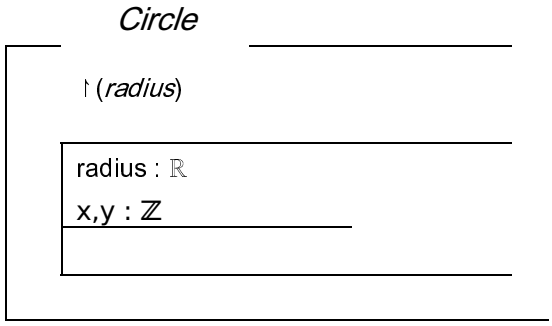


Figure 10. Simplified Object-Z specification of a circle.

5.3.2 StringTemplate

The final step when using ANTLR to translate a language is to generate output. ANTLR has a sister project called StringTemplate which is designed for this purpose. Code is embedded into the ANTLR tree grammar which passes the appropriate pieces of data into templates specified in a StringTemplate file. These templates are essentially just strings with holes in them that are filled using the data gleaned from the input. It is possible, however, to create multiple sets of templates to use with a single translation tool, and swap the templates in and out at runtime. This means that one application can translate input into multiple output languages or formats. The template mechanism is also highly flexible with the ability to pass complex objects as well as simple string data, iterate through data in list form and include data conditionally. The full ANTLR translation process is shown in Figure 11.

Thus, EBNF grammars and in particular ANTLR grammars are a concise, simple to develop way of creating an end-to-end translation tool. The Abstract Syntax Tree format provides an intuitive data model that interfaces cleanly with the StringTemplate output engine, while keeping model and view separate and allowing pluggable output modules to be added. Finally, creation of an ANTLR grammar is a fairly straightforward task, meaning that a custom grammar can be created for the Object-Z tool, giving maximum control over the input format and development of the tool in general. For all these reasons an ANTLR grammar was preferred to XML/XSLT as the underlying translation mechanism.

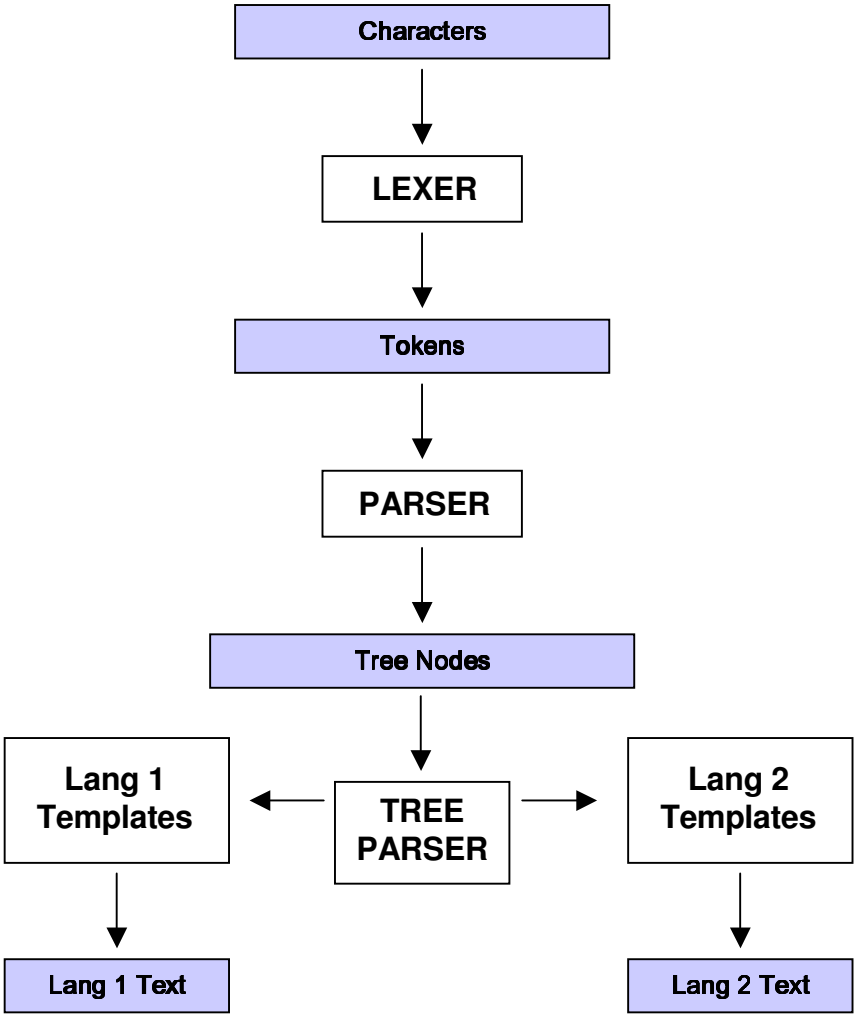


Figure 11. The ANTLR translation workflow.

5.4 DATA MODEL ARCHITECTURE

The backbone of the data model is the ANTLR grammars and the abstract syntax tree that the tree grammar produces, as explained in section 5.3.1. In addition to this, classes are needed to provide abstractions of elements within the Object-Z specification, e.g. "OperationSchema" or "StateAttribute". Instances of these classes are created and passed to the appropriate template when simply substituting a section of input text for another string does not provide the level of sophistication needed. These classes were created as and

when they were needed, and contain just the level of detail necessary to carry the required data. In other words, no overall design was necessary, and this part of the model will be explained in Chapter 6 on Implementation.

5.5 USER INTERFACE

It was decided to build the user interface for the tool using Java Swing. Although the tool was developed to run on a Windows platform, because the version of Perfect Developer provided by Escher Technologies was a Windows version, Java will make porting to other operating systems a simple matter. Also, the Swing library provides a large array of custom classes for displaying and formatting text output and good support for multithreading.

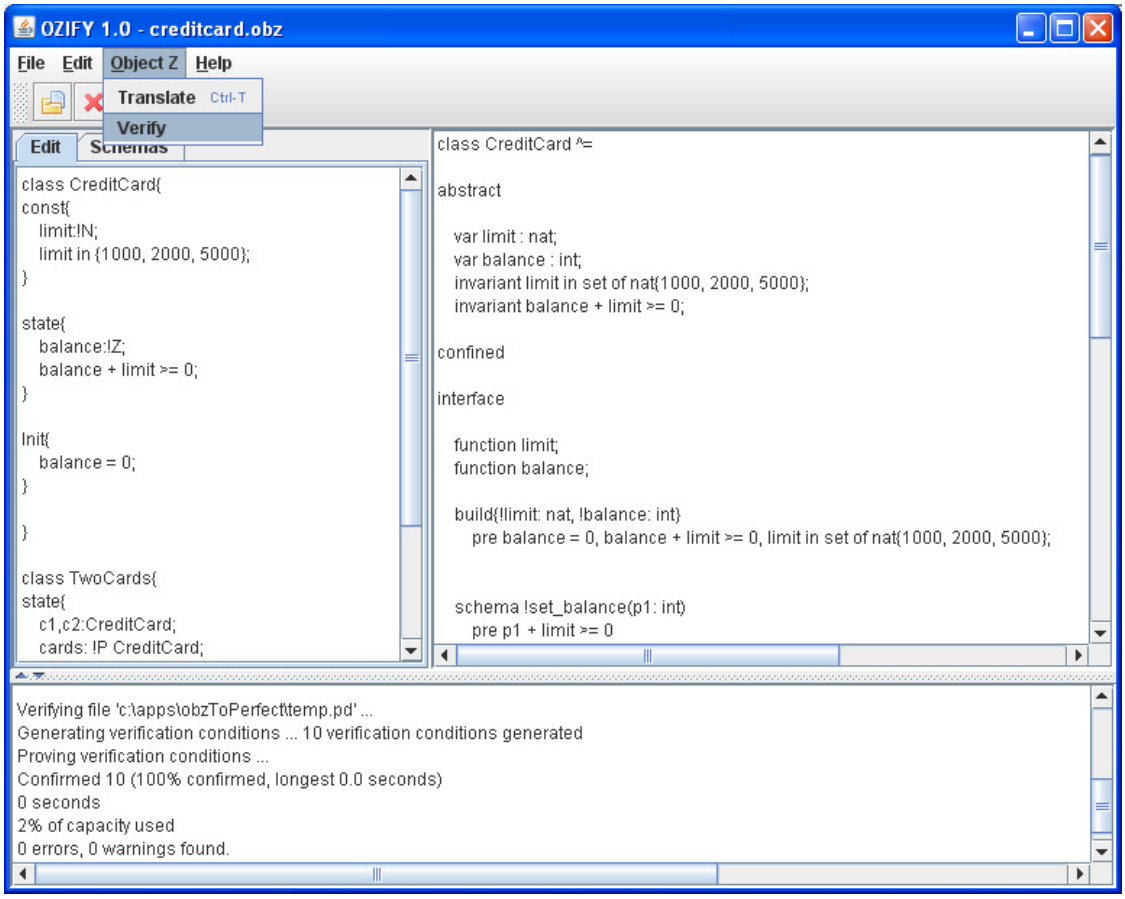


Figure 12. The user interface.

The user interface is shown in Figure 12. This consists of three main panels. The first, on the upper left of the screen, is the input area where the user enters their Object-Z specification as plain text. The content of this panel is fed into the translation software when the user chooses the Translate option on the button bar or menu. The upper right panel displays the translation into Perfect. This generated Perfect code is also saved into a temporary file on the user's hard disk. When the user is happy with their specification they will click the Verify command, which spawns a background thread and runs the Perfect Developer tool (which must be installed at a known location on the user's computer), with the temporary Perfect file as input. The output from Perfect Developer is captured by the tool and displayed in the bottom panel.

5.6 USER INPUT FORMAT

The final part of the design phase was to look at the way a user would input their Object-Z specification. As mentioned at the beginning of this chapter, the aim was to let users input their specification in as simple and intuitive a way as possible. Object-Z's graphical format makes it slow and awkward to input faithfully, so it was decided to create a text-only representation of Object-Z. In defining this the objectives were:

- keep as much of the actual Object-Z syntax as possible
- provide alternatives when graphic boxes or mathematical non-keyboard-friendly symbols are called for
- make the syntax completely unambiguous to minimize the complexity of the parser grammar

Drawing on a combination of C++/Java syntax, such as using braces {} to delimit schemas and borrowing several of the Perfect operators, a representation was created. This is fully described in Appendix A.

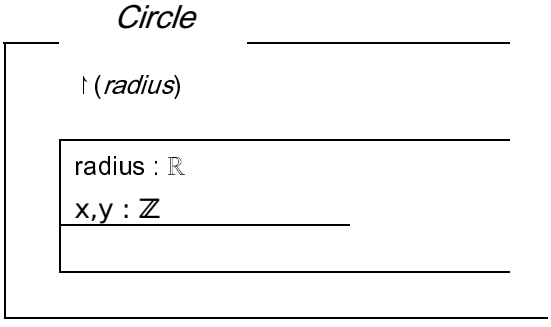
Chapter 6

TOOL IMPLEMENTATION

To enable ANTLR to produce the Java translation software, two grammars were required. The first of these defines the syntax of the input language as a series of EBNF rules. ANTLR uses this grammar to produce a lexer and a parser, which, between them, recognize the structure in the input and convert it into abstract syntax tree form. This first grammar is only concerned with the input language, it performs no translation. The second grammar is similar in structure to the first, but its rules deal with the tree structures produced by the lexer/parser software. It also contains code snippets, or "actions", attached to each rule which drive the translation and output. ANTLR builds a tree parser from this grammar.

6.1 OZ-TEXT

As mentioned in section 5.6, the tool was designed to accept input in plain text form. We will refer to this text representation of Object-Z specifications as OZ-Text. OZ-Text is essentially a rewritten version of Object-Z and contains all the same constructs. Graphical schemas were replaced by text delimiters and unusual operators have keyboard-friendly equivalents. So, the trivial Object-Z specification:



is rewritten in OZ-Text as:

```
class Circle{
    visible(radius);
    state{
        radius: !R;
        x,y: !Z;
    }
}
```

This input text produces the abstract syntax tree shown in Chapter 5, Figure 9. A full description of OZ-Text can be found in Appendix A.

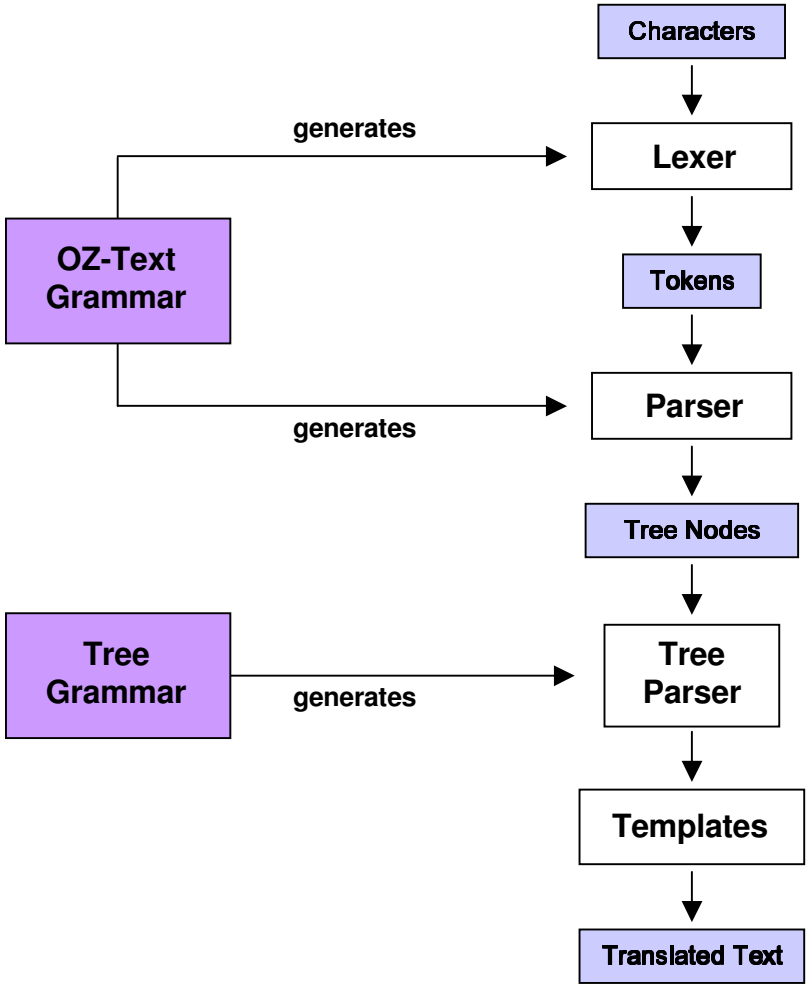


Figure 13. The role of grammars in the translation process.

6.2 OZ-TEXT RECOGNITION

The first ANTLR grammar, defining OZ-Text syntax, is based on the syntax of Object-Z. Duke and Rose[19] include a list of Object-Z production rules in Appendix C of their book, and this was used as the basis of the grammar. To fit the ANTLR format, the rules were divided into two sets: the lexer rules which define the language tokens (the "vocabulary" of the language) and the parser rules which define the language structure in terms of the lexer tokens.

6.2.1 Lexer rules

Lexer rules define tokens in terms of text strings that the lexer should match, and are generally quite straightforward. The majority of the lexer rules define a multi-character token for an operator or keyword. These rules have a single production, and are of little interest. An example of this kind of rule is:

```
PARTIAL FUNCTION  : '-|->' ;
```

which defines an ascii sequence representing the Z partial function symbol.

The remaining lexer rules define more abstract language concepts like identifiers, using patterns of characters. These rules are shown below.

```
INT         :  ('0'..'9')+ ;
REAL        :  INT '.' INT;
ID          : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
COMMENT     :  '/*' (options {greedy=false;} : .)* '*/'
               {$channel = HIDDEN;} ;
SL_COMMENT :  '//' (options {greedy=false;} : .)* '\n'
               {$channel = HIDDEN;};
WS          :  (' '|'\t'|'\n'|'\r\n')+ {$channel=HIDDEN;};
```

An integer is defined in the rule *INT* as one or more decimal digit characters in sequence (the `'+'` operator means one or more). So, the string "9" contains one *INT* token, and so does the string "4495". An identifier is any upper or lower case letter followed by any sequence (zero or more) of letters, digits or underscores. Finally, everything matched by the comment and whitespace (*WS*) rules is placed onto ANTLR's *hidden* channel. ANTLR has the concept of multiple channels that tokens are placed on to. The parser will only look at one of the channels, so any tokens placed on another, like *hidden* will be ignored. This allows the parse to concentrate on the important parts of

the text without actually throwing the other characters away. Occasionally it is necessary to retrieve tokens from the other channels, such as in the case of text containing quote-delimited strings or character literals where the whitespace is important.

6.2.2 Parser rules

ANTLR generates LL*, or top-down, parsers. Therefore, the parser rules represent top-down, hierarchical grammatical structures within the language. The Object-Z productions which they are based on are written in the same way. The first few rules in the grammar are:

```
specification :   def+ ;

def           :   classDef
              |   nonClassDef;

classDef      :   classHeading '{' visibility? inheritance?
                  localDefs? stateSchema? initialSchema?
                  operations? '}' ;
```

So, the starting rule, *specification*, will match one or more definitions (*def*), and a *def* can be a *classDef* or a *nonClassDef* etc. The *classDef* rule shows the '{' '}' delimiters selected to denote the beginning and end of schemas.

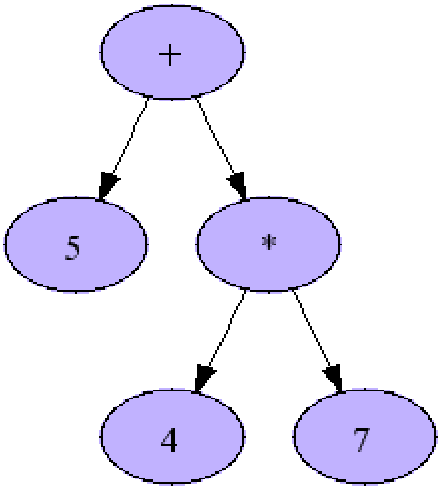
6.2.2.1 Expression hierarchies

There are three main hierarchies within the grammar: predicates, expressions and operation expressions. Within these hierarchies operator precedence is determined by the position of a rule in the hierarchy. The higher the precedence of an operator (i.e. the more "tightly" it binds its operands), the lower it is placed in the rule hierarchy. For example, two of the rules for arithmetic expressions are as follows:

```
additiveExp   :   product (('+'|'-') product)*;

product       :   unaryExpr (('*'|'/'|'%') unaryExpr)*
```

So, the first rule that the string " 5 + 4 * 7 " will match is *additiveExp*, with "5" as one *product* and "4 * 7" as the other. The substring "5" matches the *product* rule as a single *unaryExp* (the *unaryExp* rule is not shown) while "4 * 7" matches with "4" as one *unaryExp* and "7" as another. The overall parsed structure will be:



giving the correct precedence. The predicate hierarchy parses logical expressions in the same manner, and the operation expression hierarchy does the same for Object-Z composite operation expresssions. The full grammar is listed in Appendix B.

6.2.2.2 Tree construction

After determining the underlying structure in the input, the second task of the OZ-Text grammar is to generate the abstract syntax tree output. This is achieved by adding tree construction expressions to the parser rules. There are two forms of tree constructions expression. The first inserts operators into the parser rule itself to indicate the tree structure. e.g. the *equivalence* rule

```
equivalence   :   implication (IFF^ implication)*;
```

includes the ' ^ ' tree operator which indicates that the IFF token should be the root node of the tree. The second tree construction operator, ' ! ', signifies that a token should be omitted from the output tree. So, rule *decs*:

```
decs          :   dec (';'! dec)*;
```

matches rule *dec* multiple times with a ' ; ' delimiter. The ' ! ' operator removes the delimiters from the output tree. This results in a simpler tree and removes potential ambiguities.

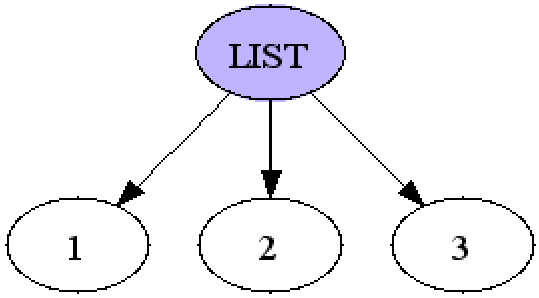
The second form of tree construction expression follows a parser rule, rather than being embedded in it, and is of the form:

```
                    -> ^(root child child …)
```

So, one of the productions for the *variable* rule is:

```
variable     :   '[' (expression (',' expression)*)? ']'
             -> ^(LIST expression*)
```

This rule will match the string "[1, 2, 3]" and produce the tree:



stripping out all the "cluttering" punctuation and introducing the *imaginary token* LIST. The LIST token is unique to this particular type of tree fragment, and so there is no possibility of ambiguity in the second grammar. This is a very flexible way of constructing trees, which can greatly simplify the output.

6.3 OUTPUT GENERATION

6.3.1 Tree grammar

The second grammar is used to generate a tree parser. This parser is fed the tree structures produced by the OZ-Text grammar, and so the output-generating grammar contains rules that recognize tree fragments. The overall structure of the grammar is the same as the first grammar, although somewhat condensed (see below). The tree fragments are defined in exactly the same way that they were written in the tree construction expressions described above. e.g. one production of the *expression* tree grammar rule is:

```
expression    :   ^(LIST expression*)
```

and this would match the tree fragment shown above. Note that the input matched OZ-Text grammar rule *variable*, whereas the tree fragment this rule produced has now matched tree grammar rule *expression* which is recursive.

This is because the parser has already recognized the structure in the input and built it into the tree. Therefore, the tree grammar does not need to be so detailed in its structure, and the hierarchies can be flattened out into fewer rules.

6.3.2 Templates

The tree parser passes data from the rules it matches into StringTemplate objects. These StringTemplates are the part of the program that actually implements the Object-Z to Perfect mapping, as illustrated below. The StringTemplates build strings by taking tokens matched during the parse and adding them to partial strings defined in a template file. Once all the templates have been generated, the final stage is to write them to output by calling the toString() method of the template returned by the top-level rule. This template will contain all the other templates in whatever structure the parse has generated, and so the full output text will be emitted.

As an example, let us look at the full version of the *expression* production introduced above (NB this is not the full *expression* rule, which has many productions):

```
expression    :   ^(LIST e+=expression*)
                  -> list(type={$typein},elements={$e})
```

The production matches a tree with a LIST root node and any number of *expression* children. Each of the children will generate a template output itself, and these are all put into a temporary collection, labelled *e*. The template output expression, shown in bold, constructs an instance of the *list* template. This template has two parameters. The first, *type*, is set equal to variable *$typein*, defined elsewhere in the grammar. The collection of *expression* templates, *e*, is passed in as the second parameter, *elements*.

The set of Perfect output templates includes this for a list:

```
list(type,elements) ::=
             "seq of <type>{<elements; separator=\", \">}"
```

meaning that parameters *type* and *elements* are inserted into the string:

```
seq of      {     }
```

to complete the Perfect expression. Because *elements* is a multivalued parameter, a *separator* string can be defined, which the template automatically places between each value in the collection. So, for the input "[1,2,3]" the string stored in the *list* template object will be "seq of int{1,2,3}".

The basic output mechanism so far described takes fragments of the input text matched by a rule, constructs an instance of a template and returns this as the result of the rule. This is sufficient for straightforward rewriting of expressions, and indeed much of the translation can be made this way. However, there are situations, such as setting the *type* parameter in the list example above, where this has to be built upon further.

6.3.3 Auxiliary data structures

An Object-Z specification often contains state variable declarations and a predicate in a state schema. An equivalent Perfect class will contain matching declarations of the state variables and class invariants in its abstract section. However, it will also contain redeclarations of the variables as functions in either its confined or interface sections. Furthermore, the state variables must be included in the list of parameters to the class constructor, and the invariants must be preconditions. The Object-Z specification has no equivalent to these redeclarations to drive their output in the Perfect translation.

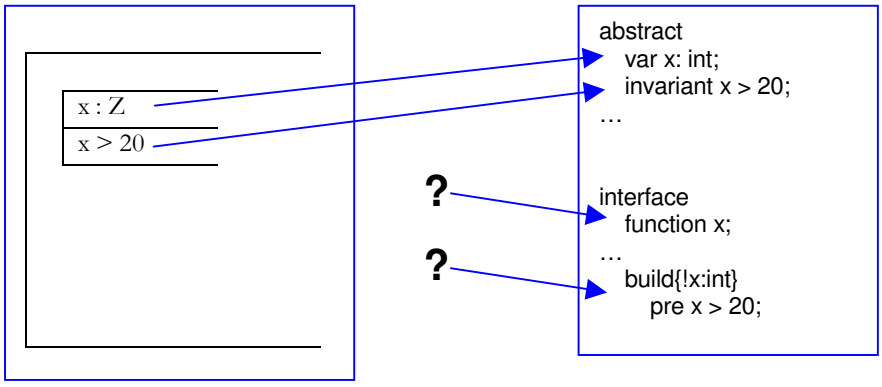


Figure 14. The contents of an Object-Z specification do not have a one-to-one mapping to the contents of an equivalent Perfect class.

Similarly, classes in Object-Z specifications contain operation schemas and operation expressions. Some of these operations may be included in the class's visibility list, others may not, but the positioning of the operation declaration in the specification is not dependent on this. In a Perfect class however, the section that schemas and functions are declared in determines whether they are visible to other classes or not.

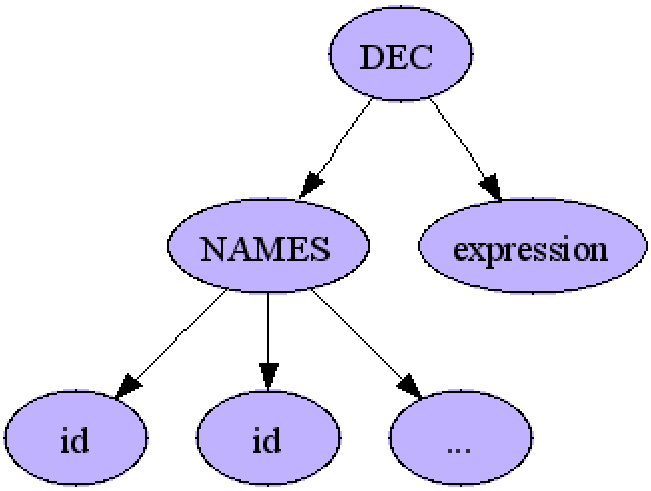
These are both instances where the Object-Z specification does not map into Perfect in a way that allows it to be translated by a simple rewrite. The answer is to put the information from certain sections of the parsed input into data structures, and then retrieve it later as needed. Such auxiliary data classes were created to provide abstractions of Object-Z constructs such as Attributes, Axioms and OperationSchemas. These were then integrated into the translation by adding code "actions" to the tree grammar. These actions are executed in addition to template construction code, when the rules match. For example, the full version of the *dec* tree grammar rule, which parses a tree fragment produced by a variable declaration, is as follows:

```
dec returns [Set<Attribute> attributes]
@init{
        $attributes = new TreeSet<Attribute>();
        List<Ident> n = new ArrayList();
}
@after{
        Iterator<Ident> iter = n.iterator();
        if ($e.type != null && $e.type.equals("total")){
           while(iter.hasNext()){
              Ident i = iter.next();
              Attribute a = new Function(i.getName(),$e.st.toString(),
                 false, true, i.getTemplate(), $e.left, $e.right);
              a.setDecoration(i.getDecor());
              $attributes.add(a);
           }
        } else {
           while(iter.hasNext()){
              Ident i = iter.next();
              String type = $e.type == null ? $e.st.toString() : $e.type;
              Attribute a = new Attribute(i.getName(),type,false, false,
                 i.getTemplate());
              a.setDecoration(i.getDecor());
              $attributes.add(a);
           }
        }
}
        :  ^(DEC ^(NAMES (id {n.add($id.ident);})+) e=expression[null])
           -> dec(names={n},type={$e.st},collection={$e.type!=null})
        ;
```

The grammar production rule and template construction expression are the bottom two lines of code. Everything that precedes this is concerned with building Attribute objects to save the information in the declaration(s). At the top of the code excerpt, the rule header declares that it returns a set of

Attribute objects. This return value can be accessed from other rules that contain a *dec* symbol in their productions using the syntax `$dec.attributes`.

The section of code after the header begins with `@init`, signifying that this code executes before anything else happens whenever the parser matches this rule. The code in the init section initializes the *attributes* return set, and creates another List object, *n*. Control now passes to the actual production rule which matches trees of the form:



The rule contains an embedded action, `{n.add($id.ident);}`, which executes every time the *id* rule is successfully matched. This action adds the Ident data object returned by the *id* rule to the local List, *n*. Finally, once the production rule has matched fully and the embedded actions have executed, the `@after` code runs. This code goes through the list of names collected in the List *n*, and constructs an Attribute object for each one. The Attribute object contains information about the name of the attribute, its type (derived from the expression rule), its visibility (initially set to "not visible", but later compared to the contents of the visibility list), and whether it is a communication parameter (*dec* also matches for operation schema declarations), and if so whether an input or an output. Finally, the constructed Attribute is added to the *attributes* return set. This completes the full execution of the *dec* rule, and assuming this is a state schema declaration, the *attributes* set is now returned up through the *stateSchema* rule to the *classDef* rule where it persists until the whole class structure has been parsed and output. These

objects allow tasks such as type checking of variables found in operations, and visibility checking, but can also be passed directly as parameters to templates, providing far more information than the local rule can parse on its own.

6.3.3.1 Output using data objects

The *attributes* set from the *dec* rule in the previous example is passed up the rule hierarchy and added to a Schema object which is built by the *stateSchema* rule. A Schema object abstracts an Object-Z schema, and has a collection of declared Attributes, and a collection of StringTemplate objects containing the schema's axioms (see Figure 15).

The Schema object is in turn passed from *stateSchema* up to rule *classDef*. Because of the lack of a one-to-one relationship between the Object-Z features like constants schema, state schema etc. and the equivalent Perfect code, rules such as *stateSchema* do not emit output themselves. The Perfect output templates for these rules are blank, and instead information is simply collected and passed up to *classDef*. Finally, the objects that have been built by the other rules are passed as parameters in the template construction statement in *classDef*.

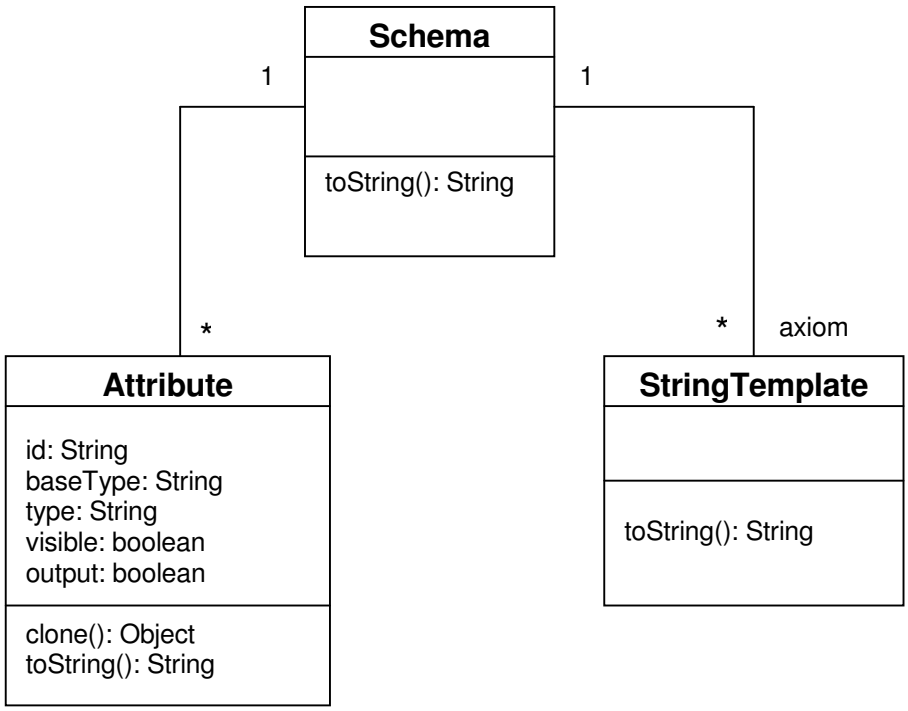


Figure 15. UML class diagram showing associations between a Schema, its attributes and axioms.

Thus, the *class* template, constructed in the *classDef* rule is a blueprint for the whole Perfect class. Sections of this template are shown below:

```
class(head,vis,locals,const,state,initConds,opSchemas) ::= <<

class <head> ^=

abstract

    <const.attributes:vardec(); separator="\n">
    <state.attributes:vardec(); separator="\n">
    <const.axioms:invariant(); separator="\n">
    <state.axioms:invariant(); separator="\n">
    ...

confined<if(opSchemas)>
    <opSchemas:confinedOpSchema()><endif>

interface
    <[const.attributes,state.attributes]:{a |<\n>function <a>;}>

    build{<[const.attributes,state.attributes]:{a |!<a>: <a.type>};
            separator=", ">} ...
    ...

    <state.attributes:setSchema()>
    <opSchemas:interfaceOpSchema()>

end;
>>
```

The first line declares the template and its formal parameter list. The parameters of interest in this discussion are *const* and *state*, which are both Schema objects, one abstracting the constants schema of the Object-Z class and one abstracting the state schema. These parameters are referred to at multiple points in the template, which is able to access the fields of the objects using the syntax `<param.field>`. This is first seen in the abstract section of the class, where the sub-template *vardec* is "applied" to *const.attributes*. Since *const.attributes* is a collection of Attribute objects, the *vardec* template will be written out once for each Attribute in the collection. The definition of *vardec* is:

```
vardec(attribute) ::= "var <attribute> : <attribute.type>;"
```

which again uses the `<param.field>` syntax to emit the type of the Attribute. So, the result is a list of declarations of the form "`var x : int ;`" etc.

The *const* and *state* parameters are accessed again in the first line of the template after keyword "interface" is printed. This time, the StringTemplate list construction syntax combines the attribute collections of the two objects into one collection, `[const.attributes,state.attributes]`, and applies an *anonymous template* to it, defined inside the braces. This results in a list of function declarations being added to the output, one for each attribute. The process is repeated again to declare the parameters of the class constructor. In this way, the whole Perfect class can be built up using data collected from disparate parts of the Object-Z specification.

6.3.4 Building schemas with AxiomFactory and CompositeOpFactory

The most complicated sections of the mapping presented in Chapter 4 are the parts relating to resolving the precondition and postcondition of an operation schema (4.9.5.1 Extracting preconditions and postconditions, p. 38), and to combining operations into composites (4.9.6 Composite operations, pp. 43-55). These are implemented within rules *pred* and *opnExp* of the ANTLR tree grammar, and within the classes AxiomFactory and CompositeOpFactory.

Taking rule *pred* as an example, we can see that the different productions recursively combine Axiom objects to produce new Axioms by calling the static factory methods of the AxiomFactory class:

```
pred [Set<Attribute> tempVars] returns [Axiom axiom]

        :  ...

        |  ^(IFF a=pred[null] b=pred[null])
                {$axiom = AxiomFactory.getEquivalence(a.axiom, b.axiom,
                getTemplateLib());}

              -> equivalence(a={$a.st},b={$b.st})

        |  ^(IMPLIES a=pred[null] b=pred[null])
                {$axiom = AxiomFactory.getImplication(a.axiom, b.axiom,
                getTemplateLib());}

              -> implication(a={$a.st},b={$b.st})

        |  ^('|' a=pred[null] b=pred[null])
                {$axiom = AxiomFactory.getDisjunction(a.axiom, b.axiom,
                getTemplateLib());}

              -> disjunction(a={$a.st},b={$b.st})

        |  ^('&' a=pred[null] b=pred[null])
              {$axiom = AxiomFactory.getConjunction(a.axiom, b.axiom,
              getTemplateLib());}

              -> conjunction(a={$a.st},b={$b.st})

        |  e=expression[null] {$axiom = new Axiom($e.st, $e.post_cond);}

              -> template(expr={$expression.st}) "<expr>"
```

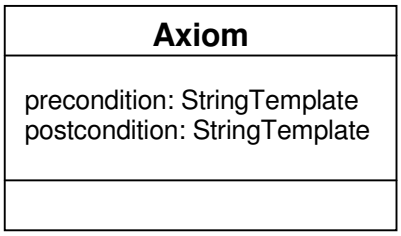
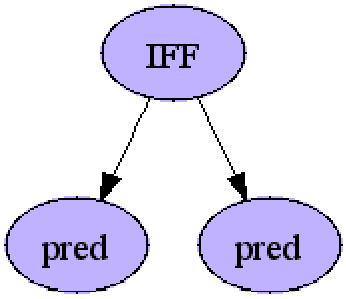


Figure 16. The Axiom class.

The last production is the base case, and matches a single *expression*, not involving a logical connective. The action for this rule creates a new Axiom object. This object will contain a copy of the *expression* matched in the rule, categorized as either a precondition or a postcondition according to the principles on page 38.

The other productions match fragments such as:



and recursively call the *pred* rule. Each call to *pred* returns an Axiom object. These child axioms are then combined in an AxiomFactory method call such as `AxiomFactory.getEquivalence(a.axiom, b.axiom,…)`, which returns a new Axiom representing the whole combined predicate. AxiomFactory takes the preconditions and postconditions of the child predicates and constructs new ones according to the principles of the mapping.

The *opnExp* rule produces OperationSchema objects in a similar way, with recursive calls passing objects to the class CompositeOpFactory for combination. CompositeOpFactory implements the composite operation

parts of the mapping, taking the parameters, preconditions and postconditions of operand OperationSchema objects, and producing the resulting OperationSchemas. The CompositeOpFactory methods implement variable and schema-call promotion, parameter hiding, and generation of implicit preconditions.

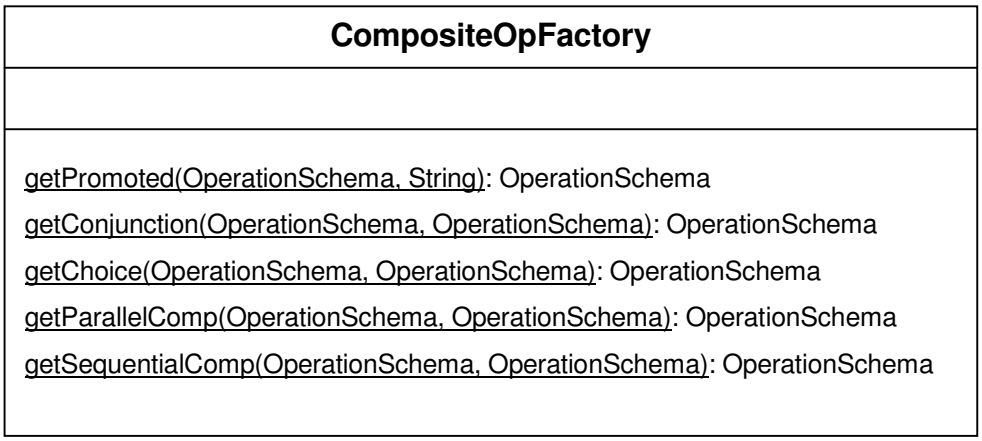


Figure 17. The CompositeOpFactory class showing its static factory methods.

6.4 USER INTERFACE

The user interface was built using the Java Swing library. The user enters text into the first text panel, and the ANTLR generated lexer and parser classes produce a translated output String which populates the second panel.

6.4.1 Concurrency during verification

When the Object-Z specification has been translated, the user can invoke the Perfect Developer verification tool by pressing the Verify button. This is a relatively slow process, producing output a line at a time. In order to prevent the GUI from freezing during verification, and to enable the output to be printed as it is produced, it was necessary to create a separate thread.

This was done by creating a class, PerfectProc, which extends the "worker thread" library class, SwingWorker. The doInBackground() method of SwingWorker was overridden to create a new Process object which executes the Perfect Developer command. Each line of output that Perfect Developer

produces is read and fed into the publish() method of SwingWorker. Finally, the process() method of SwingWorker was also overridden. This method executes each time a call to publish() occurs and outputs whatever is published into the Document model within the results panel. The SwingWorker methods are thread safe and do not require external synchronization.

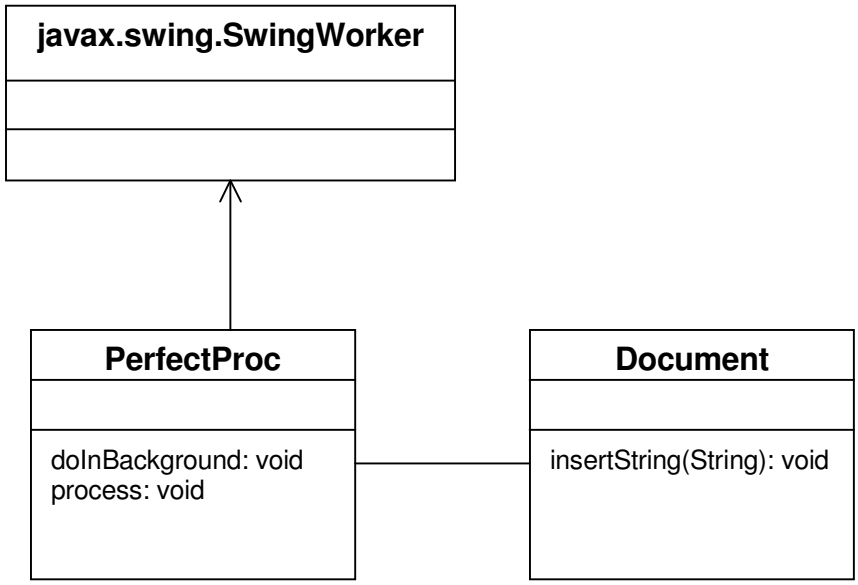


Figure 18. The PerfectProc worker thread class and the Document object that output is published to.

Chapter 7

EVALUATION

7.1 INTRODUCTION

This chapter assesses the success of the project in terms of the completeness of the mapping and the results produced by the implemented system.

7.2 MAPPING COMPLETENESS

The mapping presented in Chapter 4 does not cover 100% of the constructs in the Object-Z language. Table 2 gives a rule-by-rule analysis of which features have mappings and which do not.

Table 2. Coverage of Object-Z syntax in the project mapping. The rules in the first column are exactly those in Appendix C of Duke & Rose's book[19].

Syntax Rule	Coverage
*Specification = Def (Sep Def)**	Mapped
Def = ClassDef | NonClassDef	Mapped
ClassDef = *ClassHeading* *Visibility?* *Inheritance?* *LocalDefs?* *StateSchema?* *InitialSchema?* *Operations?*	Mapped
ClassHeading = ClassName GenFormals?	Mapped
Visibility = '↾' '(' *NameList* ')'	Mapped
*Inheritance = ClassDes (Sep ClassDes)**	Not Mapped – advanced feature
*LocalDefs = NonClassDef (Sep NonClassDef)**	Mapped

Table 2 continued…

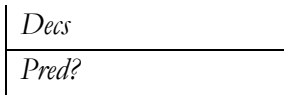
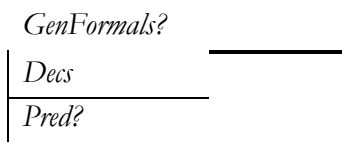
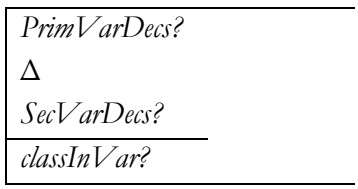
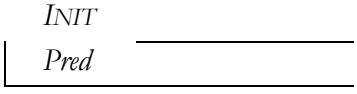
NonClassDef = *GivenTypeDef* | *FreeTypeDef* | *AbbrevDef* | *AxDef* | *GenDef*	Mapped
GivenTypeDef = ' [' *NameList* '] '	Not Mapped – limited use
FreeTypeDef = *Name* ' ::= ' *Branch* (' | ' *Branch*)*	Mapped
Branch = *Name* (' << ' *Exp* ' >> ') ?	Partial Mapping – Optional Exp has limited use
AbbrevDef = *Name GenFormals* ? ' == ' *Exp*	Mapped
AxDef = 	Mapped
GenDef = 	Not Mapped – limited use
genFormals = ' [' *NameList* '] '	Not mapped – limited use
StateSchema = 	
PrimVarDecs = *Decs*	Mapped
SecVarDecs = *Decs*	Mapped
ClassInVar = *Pred*	Mapped
InitialSchema = 	Mapped
Operations = *OpnDef* (*Sep OpnDef*)*	Mapped
OpnDef = *OpnSchemaDef* | *OpnExpDef*	Mapped

Table 2 continued…

OpnSchemaDef = *OpnName* _____ *Deltalist?* *Decs?* _____ *Pred?* _____		Mapped
OpnExpDef = *OpnName* '≙' *opnExp*		Mapped
OpnExp =	'⨾' *SchemaText* '•' *OpnExp*	Not Mapped – advanced feature
	'[]' *SchemaText* '•' *OpnExp*	Not Mapped – advanced feature
	'∧' *SchemaText* '•' *OpnExp*	Not Mapped – advanced feature
	OpnExp '•' *OpnExp*	Mapped
	OpnExp '⨾' *OpnExp*	Mapped
	OpnExp '[]' *OpnExp*	Mapped
	OpnExp '||' *OpnExp*	Mapped
	OpnExp '||ₗ' *OpnExp*	Not Mapped – variation only
	OpnExp '∧' *OpnExp*	Mapped
	OpnExp Renaming	Not Mapped – variation only
	OpnExp '\ (' *NameList* ')'	Not Mapped – variation only
	'[' *Deltalist* ? *Decs* ? ('|' *Pred*) ? ']'	Mapped
	(*Exp* '.') ? *OpnName*	Mapped
	'(' *OpnExp* ')'	Mapped
OpnName = *Name*		Mapped
Deltalist = 'Δ' '(' *NameList* ')'		Mapped
SchemaText = *Decs* ('|' *Pred*) ?		Mapped
Decs = *Dec* (';' *Dec*) *		Mapped
Dec = *NameList* ':' *Exp*		Mapped
Pred =	'∀' *SchemaText* '•' *Pred*	Mapped
	'∃' *SchemaText* '•' *Pred*	Mapped
	'∃₁' *SchemaText* '•' *Pred*	Not Mapped – variation only

Table 2 continued…

	' let ' *LetDefs* ' • ' *Pred*	Not Mapped – limited use
	Pred ' ⇔ ' *Pred*	Mapped
	Pred ' ⇒ ' *Pred*	Mapped
	Pred ' ∨ ' *Pred*	Mapped
	Pred ' ∧ ' *Pred*	Mapped
	' ¬ ' *Pred*	Mapped
	Name ' . INIT '	Mapped
	true	Mapped
	false	Mapped
	BoolExp	Mapped
	' (' *Pred* ') '	Mapped
BoolExp = *Exp*		Mapped
Exp =	' μ ' *SchemaText* ' • ' *Exp*	Not Mapped
	' λ ' *SchemaText* ' • ' *Exp*	Not Mapped
	' let ' *LetDefs* ' • ' *Exp*	Not Mapped
	' if ' *Pred* ' then ' *Exp* ' else ' *Exp*	Not Mapped
	Exp (' × ' *Exp*) +	Mapped
	' ℙ ' *Exp*	Mapped
	Exp Infix Exp	Mapped
	Exp Exp	Mapped
	Prefix Exp	Mapped
	Exp Postfix	Mapped
	ClassHierarchy	Not Mapped
	Exp ' . ' *Name*	Mapped
	Name GenFormals ?	Not Mapped
	Name GenActuals ?	Not Mapped
	' { ' *ExpList* ? ' } '	Mapped
	' { ' *SchemaText* (' • ' *Exp*) ? ' } '	Mapped
	∅	Mapped
	' (' *Exp* (' , ' *Exp*)+ ') '	Mapped
	' ⟨ ' *ExpList* ? ' ⟩ '	Mapped

Table 2 continued…

	Number	Mapped
	' self '	Mapped
	' (' *Exp* ') '	Mapped
LetDefs = LetDef (' ; ' *LetDef*)*		Not Mapped
LetDef = Name ' = = ' *Exp*		Mapped
ClassHierarchy = ' ↓ ' *ClassDes*		Not Mapped – advanced feature
ClassDes = ClassName GenActuals ? *Renaming* ?		Not Mapped
ClassName = Name		Mapped
GenActuals = ' [' *ExpList* '] '		Not Mapped – advanced feature
Renaming = ' [' *RenameList* '] '		Not Mapped
Rename = Name/Name		Not Mapped

A crude count of the rules with mappings produces a figure of 61 out of 88 mapped (69%). Allowing for clustering of similar rules (several of the unmapped rules relate to generic classes, another group to distributed operations etc.) a figure of 75-80% of the language mapped seems about right. This figure could have been higher if mapping completeness had been pursued as the overriding aim of the project, but it was decided to place roughly equal emphasis on the theoretical mapping and the implementation. Producing a working tool was a key goal, and indeed all the mapped parts of the language have been implemented in software. This provided immediate feedback, and demonstrably improved the quality of the mapping.

The rules were mapped in an order designed to give priority to the parts of Object-Z that are used most frequently, at least by those new to the language, whom the tool is aimed at. The author's highly subjective reasoning why a rule was not included is given in the rightmost column of Table 2 where appropriate. Thus, it is hoped that it is the most useful 75% of Object-Z that has been mapped in Perfect. The validity of this claim may be judged by the following case studies.

7.3 TOOL EVALUATION - CASE STUDIES

7.3.1 Case 1: a magnetic key system

This case study is presented in Duke & Rose[19], Chapter 3.

An organisation has decided to implement a system to enable access to the rooms in its offices using magnetic keycards. An employee's keycard is individual to them and will allow them to unlock particular rooms, depending on their status and responsibilities.

7.3.1.1 Informal description

- There is a set of keys and a set of rooms. It will be assumed that the sets of keys and rooms are fixed, i.e. neither keys nor rooms are added to or removed from the system. As the association between keys and rooms is the heart of the system, employees need not be modelled.

- Each key has access rights to a subset of rooms. In general, a key will be able to unlock some rooms, but not others. However, it is possible for some keys to be able to unlock all the rooms, and others to unlock no rooms.

- When a key is inserted into a room's lock, access is granted (i.e. the room unlocks) if and only if the key has access rights to that room.

- The access rights of a key may be changed, i.e. there are operations to add a room to or remove a room from the set of rooms a key can access.

7.3.1.2 Object-Z specification

The class Key is shown in Figure 19. No visibility list is included, so all features of the class are visible. A Key maintains a set, *rooms*, denoting the rooms it can access. (The class Room is defined below). Initially, the set *rooms* is emtpy, i.e. a key can access no room.

The operations *extendAccess* and *rescindAccess* model, respectively, the addition of a room to, and the removal of a room from, those accessible by the key. Details of the room to be added or removed (*rm?*) are supplied by the environment.

The operations *accessGranted* and *accessDenied* model the situation where a given room (*rm?*) is checked to see whether or not the key has access rights to that room.

We can see by inspection that all the features of Key (state schema, initial schema and operation schemas) are included in the mapping.

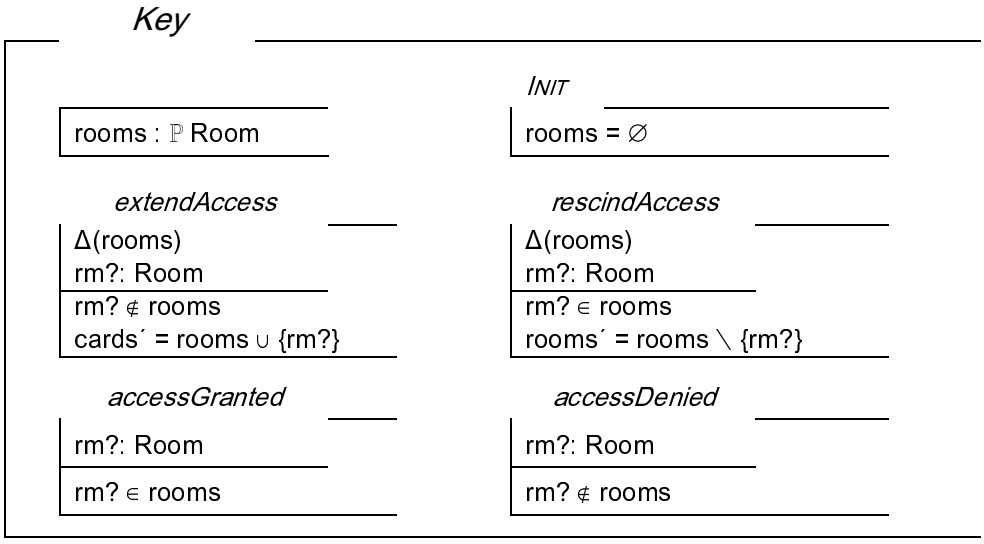


Figure 19. The class Key.

7.3.1.3 Translation

Key is rewritten into OZ-Text like this:

```
class Key{
        state{
                rooms: !P Room;
        }
        Init{
                rooms = {};
        }
        extendAccess{
                delta(rooms);
                rm?:Room;
                ¬(rm? in rooms);
                rooms' = rooms ++ {rm?};
        }
        rescindAccess{
                delta(rooms);
                rm?:Room;
                rm? in rooms;
                rooms' = rooms \ {rm?};
        }
        accessGranted{
                rm?:Room;
                rm? in rooms;
        }
        accessDenied{
                rm?:Room;
                ¬(rm? in rooms);
        }
}
```

And the tool produces this complete translation into Perfect:

```
class Key ^=

abstract

   var rooms : set of Room;

confined

interface

   function rooms;

   build{!rooms: set of Room}
      pre rooms = set of Room{};

   schema !set_rooms(p1: set of Room)
      post rooms! = p1;

   schema !extendAccess(rm: Room)
      pre ~(rm in rooms)
      post change rooms
         satisfy rooms' = rooms ++ set of Room{rm};

   schema !rescindAccess(rm: Room)
      pre rm in rooms
      post change rooms
         satisfy rooms' = rooms -- set of Room{rm};

   function accessGranted(rm: Room): bool
      ^= rm in rooms;

   function accessDenied(rm: Room): bool
      ^= ~(rm in rooms);

end;
```

The two operations that only check whether the key has access to a given room, *accessGranted* and *accessDenied*, have been translated to functions, and those that change the composition of *rooms, extendAccess* and *rescindAccess,* have been translated to schemas.

The Object-Z specification of class Room is shown in Figure 20. A Room has a single attribute, *locked*, a boolean value which is initially false. Operation *supplyId* outputs a room's identity to the environment, and *lock* and *unlock* model locking and unlocking the room.

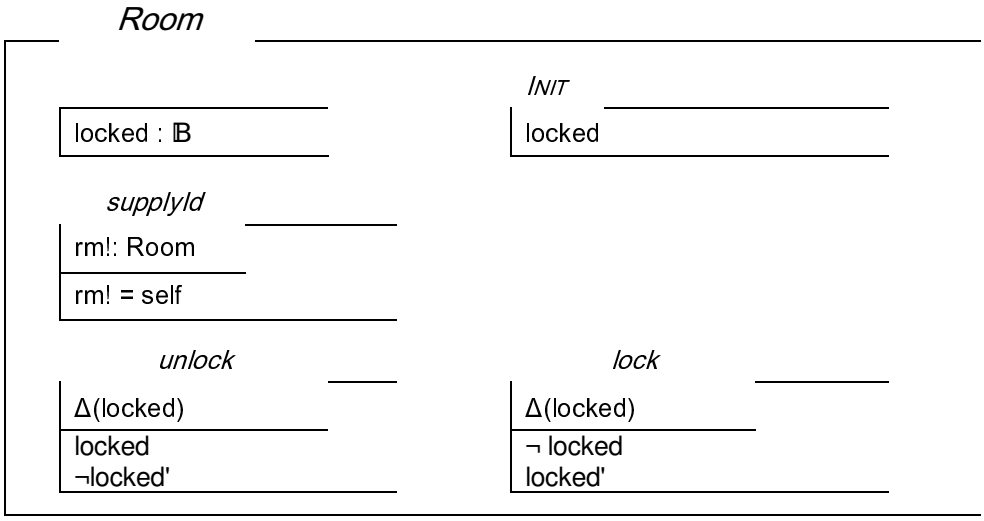


Figure 20. The class Room.

The OZ-Text (left) and translation into Perfect are shown here side-by-side:

```
class Room{

    state{
        locked: !B;
    }

    Init{
        locked = true;
    }

    unlock{
        delta(locked);
        locked = true;
        locked' = false;
    }

    lock{
        delta(locked);
        locked = false;
        locked' = true;
    }

    supplyId{
        rm!:Room;
        rm! = self;
    }
}
```

```
class Room ^=

abstract

   var locked : bool;

interface

   function locked;

   build{!locked: bool}
      pre locked = true;

   schema !set_locked(p1: bool)
      post locked! = p1;

   schema !Unlock
      pre locked = true
      post change locked
         satisfy locked' = false;

   schema !Lock
      pre locked = false
      post change locked
         satisfy locked' = true;

   schema SupplyId(rm!: out Room)
      post change rm
         satisfy rm' = self;

end;
```

The Object-Z notation of giving only the name of a boolean type, rather than equating it to 'true' or 'false' is not yet accepted by the translation tool and so the more familiar style, e.g. locked = true is used. Again, a full translation is produced.

The final class in the case study is *KeySystem* (Figure 21). The attributes *keys* and *rooms* denote the sets of all Key and Room objects in the system. Keys can only access rooms in the system and initially all keys and rooms are in their initial configurations. Operations extendAccess and rescindAccess represent selection of a key, *k?*, by the environment, and application of the same-named operation to that key. Operation *lock* is a distributed choice operation. Distributed operations were not covered in Chapter 2, as they are beyond the scope of this project, but *lock* is included for completeness. The effect of *lock* is to lock any one unlocked room in *rooms*.

The final operation, *insertKey*, captures what happens when a key, *k?*, is inserted into the lock of room *r?*. *insertKey* contains several, nested operation composition expressions. Firstly, *k?* and *r?* are selected by the environment, then *r?.supplyId* outputs the identity of *r?* as parameter *rm!*. This identity becomes a hidden communication variable since the second operand of the ' || ' operator, in parentheses, has input variable *rm?*.

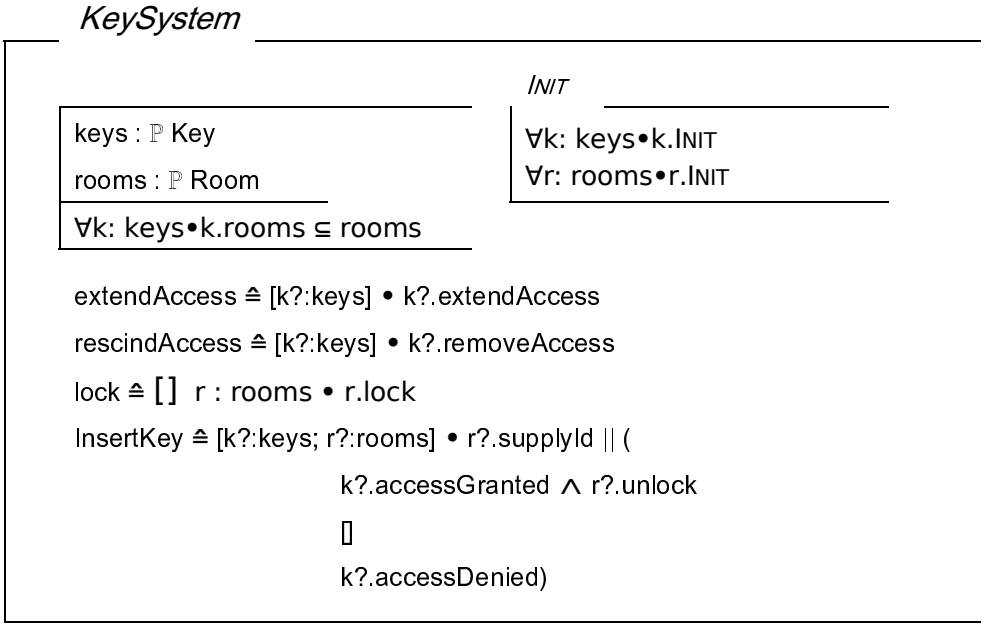


Figure 21. The class KeySystem.

The expression inside the parentheses is a choice, and either *k?.accessGranted* ∧ *r?.unlock* or *k?.accessDenied* will happen, depending on the key's access rights. This nested expression looks complicated to translate, but the mapping can be repeatedly applied to the next highest precedence operator until a final expression is produced. The steps will be:

1. Conjunction of k?.accessGranted and r?.unlock to produce newop1
2. Choice composition of newop1 and k?.accessDenied to produce newop2
3. Parallel composition of r?.supplyId and newop2 to produce newop3
4. Environment enrichment from [k?:keys; r?:rooms] to newop3 to produce *insertKey*.

The recursive implementation of rule ***opnExp*** in the tree grammar allows such recombination of operations to occur during translation:

```
opnExp returns [OperationSchema op]

        :  ^('@' a=opnExp b=opnExp)
           {
                 $op = CompositeOpFactory.getEnrichedOp($a.op,$b.op,
                       getTemplateLib());
           }
        |  ^(s=SEQCOMP a=opnExp b=opnExp)
           {
              $op = CompositeOpFactory.getSequentialComp($a.op,$b.op,
                       getTemplateLib());
           }

        |  ^(c=CHOICE a=opnExp b=opnExp)
           {
                 $op = CompositeOpFactory.getChoice($a.op, $b.op,
                       getTemplateLib());
           }

        |  ^(p=PARALLEL a=opnExp b=opnExp)
           {
              $op = CompositeOpFactory.getParallelComp($a.op, $b.op,
                       getTemplateLib());
           }

        |  ^(amp='&' a=opnExp b=opnExp)
           {
              $op = CompositeOpFactory.getConjunction($a.op, $b.op,
                       getTemplateLib());
           }

        |  ^(SELECTBOX  del=deltalist? ds=decs? ps=pred[null]
        |  ^(MEMBER ob=id? mem=id)
```

The rule returns an OperationSchema Java object. Each of the first five productions matches two *opnExp* rules and passes the OperationSchema objects returned to a CompositeOpFactory method for recombination. The final OperationSchema produced is passed to a template for output.

The translation of KeySystem produced is:

```
class KeySystem ^=

abstract

   var keys : set of Key;
   var rooms : set of Room;
   invariant forall k :: keys :- (k.rooms <<= rooms);

interface

   function keys;
   function rooms;

   build{!keys: set of Key, !rooms: set of Room}
      pre forall k :: keys :- (k.rooms = set of Room{}), forall r :: rooms :- (r.locked =
          true), forall k :: keys :- (k.rooms <<= rooms);

   schema !set_keys(p1: set of Key)
      pre forall k :: p1 :- (k.rooms <<= rooms)
      post keys! = p1;

   schema !set_rooms(p1: set of Room)
      pre forall k :: keys :- (k.rooms <<= p1)
      post rooms! = p1;

   schema !extendAccess(k!: Key, rm: Room)
      pre k in keys & ~(rm in k.rooms)
      post k!extendAccess(rm);

   schema !removeAccess(k!: Key, rm: Room)
      pre k in keys & rm in k.rooms
      post k!removeAccess(rm);

   schema !insertKey(k!: Key, r!: Room)
      pre k in keys & r in rooms & exists rm : Room :- rm = self &
            k.accessGranted(rm) & r.locked = true | k.accessDenied(rm)
      post (var rm : Room; r.supplyId(rm!) then (opaque [k.accessGranted(rm) &
            r.locked = true]: r!unlock, [k.accessDenied(rm)]:pass ));

end;
```

Operation *lock* was not included in the input since distributed choice has not been mapped at all. The composite operations a single operator, here enrichment, have all translated successfully. Nested operation expressions

were the last part of the map to be implemented, and *insertKey* does still contain some minor errors, but the principle of recursive combination of the operations to produce the overall combined result has been successful. A corrected version of insertKey only requires some extra parentheses and substitution of "self " by "r" in the precondition:

```
schema !insertKey(k!: Key, r!: Room)
  pre k in keys & r in rooms & (exists rm : Room :- rm = r &
            (k.accessGranted(rm) & r.locked = true | k.accessDenied(rm)))
  post (var rm : Room; r.supplyId(rm!) then (opaque [k.accessGranted(rm)
            & r.locked = true]: r!unlock, [k.accessDenied(rm)]:pass ));
```

This concludes Case Study 1.

7.3.2 Case 2: a supermarket scanner

This is a worked example from the Perfect Developer website[2], designed as an introduction to Perfect, and based on an assignment presented by Prof. Steve Schneider of Royal Holloway College[28].

7.3.2.1 Informal description

The example concerns the specification of a handheld supermarket barcode scanner, to allow shoppers to scan their own purchases. The scanner has a an LCD display, and buttons labelled "+", "-" and "=".

- When the "+" button is pressed, the scanned item name and its price are displayed, and the item scanned is assumed to have been added to the trolley.

- When the "-" button is pressed, the scanned item name followed by a minus-character and the item price are displayed, and the item scanned is assumed to have been removed from the trolley.

- When the "=" button is pressed, the number of items and the total price of all items in the trolley is displayed.

The items in the trolley are of type *Good* which contains values such as *apple*, *banana*, *tunaroll*. The scanner maintains a price list, which stores the price of every item of type *Good*. The total price of all items in the trolley is normally the sum of the prices of all the items therein. However, "meal deals" are a combination of two or more distinct items, such that no item appears in more than one meal deal. The price of a meal deal is greater than zero but less than the total price of its constituent items.

7.3.2.2 Object-Z specification

There follows an unofficial (i.e. original, and not from either of the sources given above) model of the classes of the shopping scanner system in Object-Z.

Good ::= apple | banana | tunaroll | crisps | coke

Price == {x : ℕ | x > 0}

PriceList

ι(priceOf)

priceOf : Good ⟶ Price

MealDealCombo

ι(meal, SupplyId)

meal : ℙ Good

#meal ≥ 2

SupplyId

m! : MealDealCombo

m! = self

Scanner

ℾ(Init, totalPlus, totalMinus, equals)

INIT

trolley : seq Good
display : seq Char
prices : PriceList

trolley = ⟨⟩
display = ⟨R,e,a,d,y⟩
totalPrice = 0

totalPrice : ℕ
mealDeals : MealDealCombo ⇸ Price
toString : Good ⟶ seq Char
numToString : ℕ ⟶ seq Char
Δ
discount : MealDealCombo ⇸ ℤ

∀x,y : dom mealDeals • (x.meal = y.meal ∨ x.meal ∩ y.meal = ∅)
∀m : dom mealDeals •
  (discount(m) = ∑g : m.meal • (prices.priceOf(g)) − mealDeals(m)
    ∧ discount(m) > 0)

totalPlus

Δ(trolley, display)

g? : Good

trolley´ = trolley^⟨g?⟩
display´ = toString(g?)^" "^numToString(prices.priceOf(g?))

totalMinus

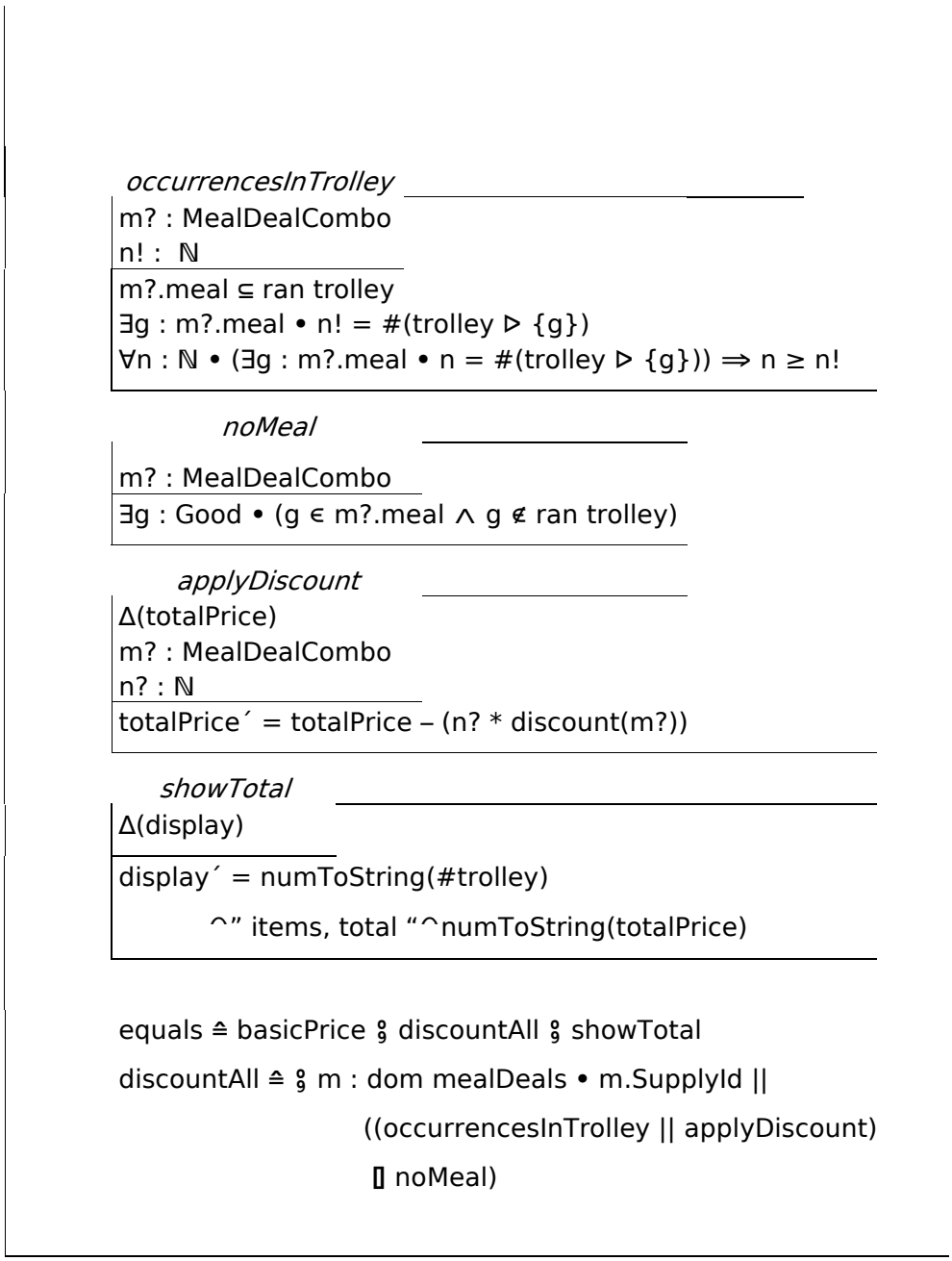
Δ(display,trolley)

g? : Good

(∃n : ℕ • (trolley(n) = g?)
  ∧ trolley´ = squash (trolley ⧹ {n ↦g?})
  ∧ display´ = toString(g?)^"−"^numToString(prices.priceOf(g?))
∨ (g? ∉ ran trolley
  ∧ display´ = ⟨l,t,e,m, ,n,o,t, ,i,n, ,t,r,o,l,l,e,y⟩)

basicPrice

Δ(totalPrice)
totalPrice´ = ∑ i : dom trolley • prices.priceOf(trolley(i))



The level of success translating this more complex specification was moderate. Much of the difficulty, though, is based on yet-to-be implemented operators such as range restriction, ' ⊳ '. This is used above to find the number of occurrences of a particular article in the trolley : #(trolley ⊳ {g}). The operator creates a new sequence from *trolley*, with elements limited to *g,* and then the length, ' # ' is taken. Operators such as this reduce the size of a

specification by encapsulating much longer expressions in a single symbol. Mapping them should pose no problem if the tool is developed into a full application, but the priority in this project was placed on the more fundamental concepts of Object-Z.

Taking each part of the shopping scanner specification at a time: the first four classes, *Good, Price, PriceList* and *MealDealCombo* can all be completely translated. In *Scanner* itself, all state variables translate with the exception of *discount*, which is a secondary function. Secondary functions will require a variant of the secondary variable mapping on page 35, but have not been covered. The first class invariant translates but the sum of operator, ' ∑ ' in the second has not been implemented. The full *INIT* schema translates, as does the operation *totalPlus*. The 'squash' and maplet, ' ↦ ', operators of *totalMinus* have not been implemented, and *basicPrice* contains, ' ∑ '. *occurrencesInTrolley* lacks the ' ⊳ ' operator, but *noMeal, applyDiscount* and *showTotal* all translate fully. The *equals* expression of two sequential compositions is translated, as is a large part of the *discountAll* expression, with the exception of the initial distributed sequential operator.

So, in summary, the shopping scanner provided a stern test of the translator, and points to areas that can provide future developments. Even so, well over 50% of this specification can be handled, and only two non-operator related problems (secondary function and distributed composition) were encountered.

7.4 PERFECT VERIFICATION

Returning to Case Study 1, a verification check with the Perfect Developer "backend" produces this output:

```
Generating verification conditions ... 7 verification conditions generated
c:\apps\obzToPerfect\temp.pd (68,14): Warning! Did not attempt to prove
condition is satisfiable.
Proving verification conditions ...
Confirmed 7 (100% confirmed, longest 0.0 seconds)
```

The warning in the second line relates to the *SupplyId* operation which Perfect's theorem prover cannot verify without more implementation detail. Perfect Developer has found seven instances where it expects a certain

condition (class invariant, schema precondition etc.) to be satisfied. All seven are satisfied in the example as specified.

Now, it is possible that the company owning the magnetic key system may decide to limit the access of each key to no more than 10 rooms. So, we add an invariant to the class Key to this effect:

```
#rooms <= 10;
```

Retranslation and running of the verification again, gives this result:

```
c:\apps\obzToPerfect\temp.pd (24,14): Warning! Unable to prove:
Class invariant satisfied (defined at c:\apps\obzToPerfect\temp.pd
(6,22)) in context of class Key [c:\apps\obzToPerfect\temp.pd
(1,1)], cannot prove: #self'.rooms <= 10.
```

Verification has correctly identifed that the new invariant may be broken by operation *extendAccess* which increases the size of set *rooms* without restriction. The error message can certainly be improved on – it does not relate to a named operation, but the error has been found.

Similarly, if the designer of class Key had included a partial visibility list, without rescindAccess, verification reports:

```
c:\apps\obzToPerfect\temp.pd (103,16): Error! Illegal access to
member 'rescindAccess'.
```

Finally, if rescindAccess in KeySystem is defined thus:

```
rescindAccess ^= [k?:keys]@  k?. accessDenied & k?.rescindAccess;
```

the translation correctly combines the identical inputs to *accessDenied* and *rescindAccess* into a single variable, *rm?*:

```
schema !rescindAccess(k!: Key, rm: Room)
   pre k in keys & k.accessDenied(rm) & rm in k.rooms
   post k!rescindAccess(rm);
```

and verification identifies that the *rescindAccess* operation is bound to fail in this situation:

```
c:\apps\obzToPerfect\temp.pd (106,16): Warning! Given 'false', so
trivial proof for: Precondition of 'rescindAccess' satisfied (defined at
c:\apps\obzToPerfect\temp.pd (26,16)) in context of class
KeySystem [c:\apps\obzToPerfect\temp.pd (73,1)].
```

These examples are limited by the incomplete implementation of the tool and are far from exhaustive, but they do illustrate the capacity for an Object-Z specification to be automatically verified.

7.5 CONCLUSION AND FUTURE WORK

This chapter has demonstrated that a large part of the Object-Z language has been successfully mapped into the Perfect language. Furthermore, this mapping has been implemented and the tool produced is capable of fully translating simple, beginner level, Object-Z specifications. The translations provide immediate feedback for the user, by clarifying the meaning of Object-Z expressions (e.g. what inputs and outputs does a composite operation actually have?), and allow formal verification by the Perfect Developer tool. With further work, it seems likely that a robust and useful tool, capable of translating far more complex specifications could be developed.

Indeed, there is still significant work that can be undertaken on this subject, both on the mapping and implementation. The main areas would be:

- *Mapping of remaining features.* In particular, inheritance, generic classes and distributed operations.

- *Processing of Perfect Developer verification output.* The warnings and errors produced by Perfect Developer relate to the translated Perfect file, and it would be extremely valuable to users of the tool to take these messages and "translate then back" into what it means about the Object-Z. The Perfect Developer output provides line number information and this is also available for the Object-Z file from the ANTLR parser. These two could be linked up as a first step, and then further analysis of the meaning of the error could be added.

- *Addition of a graphical view of the Object-Z schemas.* This work has been started, with templates for conversion of the Object-Z input to html format having been written. Unfortunately, the Java Swing support for displaying html, particularly table layout and image loading, was

not as good as anticipated, and this feature could not be completed within the time. No doubt a solution could be found, and this would be a valuable addition to the tool, giving users visual confirmation that their entered schemas matched up to text book examples etc.

- *Improve error handling and feedback in parsing and translation.* ANTLR produces well ordered, useful exceptions during parsing, which contain information on the rule and line within the text in which they were produced. These should be caught and fed back into the GUI, rather than going to standard output as they currently do.

- *Add full set of unit tests to preserve correct functionality.* The application is already quite sizeable and complex. Further development is likely to introduce unintentional bugs, which adding unit tests would prevent.

- *Add syntax highlighting for both OZ-Text and Perfect code.* Highlighting keywords, operators etc., which could be done through the application's OZDocument class in the case of OZ-Text, would aid useability. The warning and error messages produced by Perfect Developer could also be colour-coded as they are in the GUI version of that tool.

- *Allow "inclusion" of one Object-Z file in another.* Currently, all Object-Z classes relevant to a specification must be listed within one file, which can cause it to become long and awkward to work with. If a mechanism to allow files to be included as for C++ or Java, this would simplify project organisation.

- *Integrate Perfect code generation*. Perfect Developer has a code generation facility that users can employ to build a quick implementation of their system to test it in practice. Adding this feature to the Object-Z tool would enable to allow you to confirm that an application that meets its specification also behaves as intended.

References

1. http://www.itee.uq.edu.au/~smith/objectz.html
2. http://www.eschertech.com/index.php
3. http://www.eiffel.com/
4. Ashraf, A. & Nadeem, A. (2006) Automating the Generation of Test Cases from Object-Z Specifications, *in* 'Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International', pp. 101-104.
5. Kassel, G. & Smith, G. (2001) Model checking Object-Z classes: Some experiments with FDR, *in* 'Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific', pp. 445-452.
6. McDonald, J.; Murray, L. & Strooper, P. (1997) Translating Object-Z specifications to object-oriented test oracles, *in* 'Software Engineering Conference, 1997. Asia Pacific ... and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings', pp. 414-423.
7. McDonald, J. & Strooper, P. (1998) Translating Object-Z specifications to passive test oracles, *in* 'Formal Engineering Methods, 1998. Proceedings. Second International Conference on', pp. 165-174.
8. McDonald, J. Strooper, P. & Hoffman, D. (2003) Tool support for generating passive C++ test oracles from Object-Z specifications, *in* 'Software Engineering Conference, 2003. Tenth Asia-Pacific', pp. 322-331.
9. Ramkarthik, S. & Zhang, C. (2006) Generating Java Skeletal Code with Design Contracts from Specifications in a Subset of Object Z, *in* 'Computer and Information Science, 2006. ICIS-COMSAR 2006. 5th IEEE/ACIS International Conference on', pp. 405-411.
10. Wen, Z. Miao, H. & Zeng, H. (2006), Generating Proof Obligation to Verify Object-Z Specification, *in* 'Software Engineering Advances, International Conference on', pp. 38-38.
11. Ho Von, V. (2007) 'An environment for mapping Object-Z specification into Java/JML annotated code', Master's thesis, Imperial College, London.
12. Dixon, E. (2004) 'Object-Z to perfect developer', Master's thesis, Imperial College, London.
13. Giles, N. (2002) 'An Object-Z to JML parser', Master's thesis, Imperial College, London.

14. Heaven, W. J. D. (2001) 'Mapping Object-Z specifications to Java program specifications', Master's thesis, Imperial College, London.

15. Lampropoulos, N. (2006) 'An environment for mapping Object-Z specifications into Java / JML annotated code', Master's thesis, Imperial College, London.

16. Pallis, G. (2005) 'An environment for mapping Object-Z specifications into Java/JML annotated code', Master's thesis, Imperial College, London.

17. Simmonds, M. J. (2002) 'Mapping of Object-Z specifications to ESC/Java specifications', Master's thesis, Imperial College, London.

18. Stevens, B. (2006) 'Implementing Object-Z with Perfect Developer', *in* Journal of Object Technology, vol. 5, no. 2, March-April, 2006. pp. 189-202. Available at http://www.jot.fm/issues/issue_2006_03/article5

19. Duke, R. & Rose, G. (2000) Formal Object-Oriented Specification Using Object-Z, MacMillan Press.

20. Lightfoot, D. (2001) Formal Specification Using Z. Palgrave.

21. http://www.eschertech.com/tutorial/tutorials.htm

22. http://www.eschertech.com/

23. http://czt.sourceforge.net/

24. Sun, J. Dong, J. S. Liu, J. & Wang, H. (2001) Object-Z Web Environment and Projections to UML *in* '10th International World Wide Web Conference (WWW-10)', ACM Press, pp. 725-734.

25. Ankrum, T. S. & Bellaachia, A. (2003) ZML: Z Formal Specifications Using XML. (http://www.seas.gwu.edu/~bell/publications/zml-report.pdf).

26. Garshol, L. M. (2003) BNF and EBNF: What are they and how do they work?' Web-article: http://www.garshol.priv.no/download/text/bnf.html.

27. Backus, J. W. Bauer, F. L. Green, J. Katz, C. McCarthy, J. Naur, P. Perlis, A. J. Rutishauser, H. Samelson, K. Vauquois, B. Wegstein, J. H. van Wijngaarden, A. Woodger, M. & van der Poel, W. L. (1962) Revised report on the algorithmic language Algol 60, 'Numerische Mathematik' **4**(1), 420-435.

28. Schneider, S. (2003) 'Formal Methods at Royal Holloway: Perspectives and Pitfalls'"Teaching Formal Methods: Practice and Experience workshop, Oxford Brookes University'. Available at http://www.cms.brookes.ac.uk/tfm2003/ and at http://www.cs.rhul.ac.uk/research/formal/steve/talks/tfm.ps

Appendix A

TEXT INPUT FORMAT

All schema constructs should be entered in the form: `name{…}`. All other statements inside and outside of schemas should be terminated with a ' ; '. In the following definitions keywords are underlined, other terms can be replaced by any valid identifier. * indicates that the item may appear 0-to-many times.

Free type definition:

```
name ::= element | element | …;
```

Let definition:

```
name == expression;
```

Class schema:

```
name{
…}
```

Visibility list:

```
visible(attrib, attrib,…);
```

Constants schema:

```
const{
    declaration;*
    axiom;*
}
```

State schema

```
state{
    declaration;*
    axiom;*
}
```

Initial schema:

```
Init{
    axiom;*
}
```

Operation schema:

```
name{
    delta(id, id, …);
    declaration;*
    axiom;*
}
```

Operation expression:

```
name ^= expression;
```

Operators

The text input form is shown to the right of the actual Object-Z operator. All other Object-Z constructs and keywords such as "seq" can be entered without modification. No "not versions" (≠ etc.) of operators are provided; expressions should be negated using the '¬' operator.

•	@		ℤ	!Z		∀	forall		′	‘
⨟	0/9		ℕ	!N		∃	exists		∈	in
[]	[]		ℝ	!R		¬	¬		⊆	<<=
‖	||		𝔹	!B		↔	<->		⊇	>>=
∧	&		ℙ	!P		→	->		∪	++
∨	|		×	><		≥	>=		∩	**
Δ	delta		⟷	<-->		≤	<=		\	\
≙	^=		⟶	-->		⟨	[		∅	[]
			⤔	-|->		⟩	]			

Comments

C-style single line, "//" and multiline "/*…*/" comments are both accepted.

Appendix B

OZ-TEXT LEXER/PARSER GRAMMAR

Tree construction expressions have been removed from the grammar to reduce it to its basic EBNF form and aid readability.

```
specification :   def+ ;

def           :   classDef
              |   nonClassDef;

classDef      :   classHeading '{' visibility? inheritance?
                  localDefs? stateSchema? initialSchema?
                  operations? '}' ;

classHeading  :   'class' id genFormals?;

visibility    :   'visible' '(' id (',' id)* ')'';' ;

localDefs     :   nonClassDef (';' nonClassDef)*;

nonClassDef// :
//            |   givenTypeDef
              :   freeTypeDef
              |   abbrevDef
              |   axDef
//            |   genDef;

//givenTypeDef:

freeTypeDef   :   ID '::=' branch ('|' branch)* ';' ;

branch        :   ID ('<''<' expression '>''>')? ;

abbrevDef     :   ID genFormals? '='=' expression ';';

axDef         :   'const' '{' (dec ';')+  (pred ';')* '}';

//genDef:

genFormals    :   '[' ID+ ']' ;

stateSchema   :   'state'   '{'   primVarDecs?   ('^'   secVarDecs)?
              classInVar? '}' ;
```

```
primVarDecs   :   (dec ';')+ ;

secVarDecs    :   (dec ';')+ ;

classInVar    :   (pred ';')+ ;

initialSchema :   'Init' '{' (pred ';')+ '}' ;

operations    :   opnDef+;

//Operation hierarchy
opnDef        :   opnSchemaDef
              |   opnExpDef;

opnSchemaDef  :   ID '{' deltalist? (dec ';')* (pred ';')* '}' ;

opnExpDef     :   ID '^''=' opnExp ';' ;

opnExp        :   seqOpnExp ('@' seqOpnExp)*;

seqOpnExp     :   choiceOpnExp (SEQCOMP choiceOpnExp)*;

choiceOpnExp  :   parallelOpnExp (choiceOp parallelOpnExp)*;

choiceOp:     '['']';

parallelOpnExp   :conjOpnExp (PARALLEL conjOpnExp)*;

conjOpnExp    :   opnAtom ('&' opnAtom)*;

opnAtom       :   '[' deltalist? decs? ('|' pred)? ']'
              |   (id '.' )?
              |   '(' opnExp ')' ;

deltalist     :   'delta' '(' ID (',' ID)* ')' ';' ;

schemaText    :   decs ('|' pred)? ;

decs          :   dec (';' dec)*;

dec           :   id (',' id)* ':' expression;

//predicate hierarchy
pred          :   'forall' schemaText '@' pred
              |   'exists' schemaText '@' pred
              |    equivalence;

equivalence   :   implication (IFF implication)*;

implication   :   disj (IMPLIES implication)?;  //right associative
```

```
disj          :   conj  ('|' conj)*;

conj          :   expression ('&'  expression)*;

//expression hierarchy
expression
              :   setExp (relationInfix setExp)*;

relationInfix
              :   RELATION
              |   CARTESIAN
              |   TOTAL_FUNCTION
              |   PARTIAL_FUNCTION;

setExp        :   POWERSET comparisonExp
              |   'seq' comparisonExp
              |   comparisonExp;

comparisonExp :   setChangeExp (comparisonOp setChangeExp)?;

comparisonOp  :   '<'
              |   '>'
              |   LTE
              |   GTE
              |   EQUALS
              |   'in'
              |   CONTAINS
              |   RCONTAINS;

setChangeExp  :   rangeExp   ((DIFFERENCE   |   UNION   |   INTERSECT)
             rangeExp)*;

rangeExp      :   additiveExp (RANGE additiveExp)?;

additiveExp   :   product (('+'|'-') product)*;

product       :   unaryExpr (('*'|'/'|'%') unaryExpr)*;

unaryExpr     :   '+' variable
              |   '-' variable
              |   '¬' variable
              |   variable;


variable      //: classHierarchy
              :   ID '.' variable
//            |   ID genActuals?
              |   ID '.' INITCALL
              |   'true'
              |   'false'
              |   '{' (expression (',' expression)*)? '}'
              |   lb='{' schemaText ('@' expression)? '}'
```

```
              |   'empty'
              |   '[' (expression (',' expression)*)? ']'
              |   ID variable
              |   INT
              |   REAL
              |   id
              |   'self'
              |   '(' pred (',' pred)* ')';

//genActuals
//            :   expression (',' expression)*;

//classHierarchy :

id            :   type
              |   ID
              |   '#' ID
              |   ID'?'
              |   ID '!'
              |   ID'\''
              |   '\'' ID? '\''
              |   'dom' ID
              |   'ran' ID;

type          :   NATTYPE
              |   INTTYPE
              |   REALTYPE
              |   BOOLTYPE
              ;

//Lexer rules
POWERSET:     '!P';
NATTYPE       :   '!N';
INTTYPE       :   '!Z';
REALTYPE      :   '!R';
BOOLTYPE      :   '!B';
CARTESIAN     :   '><';
RELATION      :   '<-->';
IFF           :   '<->';
IMPLIES       :   '->'|'<-';
TOTAL_FUNCTION:   '-->';
PARTIAL_FUNCTION :'-|->';
GTE           :   '>=';
LTE           :   '<=';
CONTAINS      :   '>>=';
RCONTAINS     :   '<<=';
EQUALS        :   '=';
UNION         :   '++';
DIFFERENCE    :   '\\';
INTERSECT     :   '**';
SEQCOMP       :   '0/9';
PARALLEL      :   '||';
```

```
RANGE         :   '..';
INITCALL      :   'Init';
REAL          :   INT '.' INT;
INT           :   ('0'..'9')+ ;

ID            :
                  ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_
              ')*;

CMT           :   '/*'   (options   {greedy=false;}   :   .)*   '*/'
              {$channel = HIDDEN;} ;

SL_COMMENT    :   '//'   (options   {greedy=false;}   :   .   )*   '\n'
              {$channel = HIDDEN;};

WS            :   (' '|'\t'|'\n'|'\r\n')+ {$channel=HIDDEN;};
```