From Object-Oriented Specification to Implementation: A Formal Refinement Methodology.

Moonsung Yoo

Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_disstheses/6378

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
FROM OBJECT-ORIENTED SPECIFICATION TO IMPLEMENTATION: A FORMAL REFINEMENT METHODOLOGY

A Dissertation

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural Mechanical College in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in

The Department of Computer Science

by

Moonsung Yoo
B.S., Seoul National University, 1978
M.S., Indiana University at Bloomington, 1991
December 1996
ACKNOWLEDGMENTS

I would like to thank Dr. Doris L. Carver, my major adviser, for her help, advice, guidance, and patience through the course of my studies and research at Louisiana State University. She provided me with many useful comments to improve the quality of this dissertation.

I also wish to express my thanks to the committee members: Dr. Sitharama Iyengar, Dr. Bush Jones, and Dr. Kwei Tang for serving and contribution to this research. In addition, I would like to thank the students in the Software Engineering Group for their support.

I would like to express my appreciation to my parents, Soong-Jong Yoo and In-Sook Yoo, and parents-in-law, Hang-Kyu Lee and Bok-Young Lee. Without their financial support and encouragement, this research would not be possible.

Finally, my sincere thanks go to my wife, Jin-Chan and our daughters, Janet and Madeline. Their love and sacrifice during this research are beyond description.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>ii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>Abstract</td>
<td>viii</td>
</tr>
<tr>
<td>Chapter</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Formal specification languages</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Object-oriented paradigm</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Eiffel: An object-oriented language</td>
<td>7</td>
</tr>
<tr>
<td>1.5 Outline of the dissertation</td>
<td>10</td>
</tr>
<tr>
<td>2. RELATED RESEARCH</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Formal Development Methods</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Refinement calculus and refinement methods</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1 Refinement calculus</td>
<td>16</td>
</tr>
<tr>
<td>2.2.2 Refinement methods</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Refinement in VDM</td>
<td>21</td>
</tr>
<tr>
<td>2.3.1 Data refinement</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 Operation refinement</td>
<td>23</td>
</tr>
<tr>
<td>2.4 Other approaches to convert from specification to programming language</td>
<td>25</td>
</tr>
<tr>
<td>2.5 Summary</td>
<td>26</td>
</tr>
<tr>
<td>3. OBJECT-ORIENTATION IN VDM</td>
<td>28</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Object-VDM</td>
<td>28</td>
</tr>
<tr>
<td>3.3 An Example</td>
<td>33</td>
</tr>
<tr>
<td>3.4 Survey of existing object-oriented VDM</td>
<td>40</td>
</tr>
<tr>
<td>3.4.1 Fresco</td>
<td>40</td>
</tr>
<tr>
<td>3.4.2 QoVDM</td>
<td>41</td>
</tr>
<tr>
<td>3.4.3 VDM++</td>
<td>43</td>
</tr>
<tr>
<td>3.4.4 Comparison of existing languages</td>
<td>46</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>47</td>
</tr>
<tr>
<td>4. REFINEMENT IN OBJECT-VDM</td>
<td>49</td>
</tr>
<tr>
<td>4.1 Data Refinement</td>
<td>49</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF FIGURES

1.1 Software Life Cycle ................................................................. 1
1.2 VDM overall structure ............................................................. 4
1.3 VDM specification of the triangle manipulation system .......... 5
1.4 Eiffel code for the STACK class ............................................ 11
3.1 Structure of an Object-VDM specification ............................ 29
3.2 Object-VDM specification of the triangle manipulation system . 31
3.3 VDM specification of the student records system ................. 35
3.4 (part 1) VDM specification of the graduate student records system ... 36
3.5 (part 2) VDM specification of the graduate student records system ... 37
3.6 Object-VDM specification of the student records system .......... 38
3.7 Object-VDM specification of the graduate student records system ... 39
3.8 Overall structure of Fresco .................................................... 40
3.9 Fresco specification of the triangle manipulation system ........ 41
3.10 Oo^DM^PGcification ............................................................ 42
3.11 Oo^DM^specification of the triangle manipulation system ......... 43
3.12 Structure of VDM++ specification ......................................... 44
3.13 VDM++ specification of the triangle manipulation system ...... 45
3.14 Comparison of object-oriented VDMs ................................. 46
4.1 Object-VDM code for a class SET ........................................ 56
4.2 Object-VDM code for a class SEQUENCE ............................. 57
5.14 Eiffel code for the student records system (part 2) ..................................... 105
5.15 Eiffel code for the graduate student records system (part 1) .......................... 108
5.16 Eiffel code for the graduate student records system (part 2) .......................... 109
ABSTRACT

Traditionally, software development models use different methods and techniques in each phase from specification through design to implementation. Significant changes in the representations between phases have been common. The formal development method based on formal specification and stepwise development has been suggested to reduce the change in representation. The formal development method consists of a formal specification and verified design. In the formal specification step, a formal specification language is used to specify an accurate, consistent, and complete system. Vienna Development Method (VDM) is one of the most widely used formal specification languages. A verified design guides the development of the system from specification to executable code. A refinement method is used in VDM for that purpose.

The use of the object-oriented paradigm is another important trend in software engineering. Initially, object-oriented methods were applied primarily during the implementation phase using object-oriented languages. Eiffel is an object-oriented programming language which has many strong facilities such as assertions and genericity. Numerous object-oriented specification languages exist, including object-oriented extensions to VDM. We defined Object-VDM to help remove limitations from existing object-oriented VDM languages.

In this dissertation, we investigate a formal development method in the object-oriented environment since limited research has been done in the area. We defined a refinement method that refines an Object-VDM specification to Eiffel code. There are three stages in this refinement: data refinement, operation refinement, and structure refinement. In data refinement, the mathematical data models in Object-VDM are converted to Eiffel data structures by creating Eiffel libraries. We proved the correctness of the conversion. In operation refinement, we modified and added rules to the
original refinement to obtain Eiffel code. Object-oriented features are converted in the structure refinement step.

In summary, this research provides a refinement method in object-oriented environments. Specifically, the refinement converts Object-VDM specifications to Eiffel codes.
CHAPTER 1
INTRODUCTION

1.1 OVERVIEW

There are six phases in the traditional software life cycle: requirement analysis, specification, design, implementation, testing, and maintenance. The waterfall model which represents these phases is shown in Figure 1.1. Most traditional software development models use different methods and techniques in each phase from specification through design to implementation. Therefore, there are radical changes between phases. To

Figure 1.1: Software Life Cycle

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
solve this problem, a formal development method based on formal specifications and stepwise development has been suggested. The first step of formal development is to specify a system by using a specification language. A formal specification language can specify a system more accurately, consistently, and completely than informal methods. Vienna Development Method (VDM) is one of the most widely used formal specification languages. The second step of formal development is a verified design to take the system from specification to executable code. Another important approach in software engineering is the use of an object-oriented paradigm.

The object-oriented approach has been recognized since the early 1980's as the one of the best ways currently available for structuring software systems. It groups together data structures and the operations performed on them, encapsulates them behind a clean interface, and organizes the entities in a hierarchy based on inheritance. Initially, object-oriented methods were applied primarily during the implementation phase using object-oriented languages. C++, Smalltalk, CLOS, and Eiffel are some widely known object-oriented languages. Eiffel is an object-oriented programming language which has many strong facilities such as assertions and genericity. In recent years, the object-oriented paradigm have been applied to other phases of the software development process. Some research efforts combine an object-oriented paradigm and a formal specification language. VDM has been extended to include object-oriented features [Wil 92a][LS 93][DP 95]. Since existing object-oriented extensions to VDM have a limited set of features, we create Object-VDM, an object-oriented extension to VDM. Limited research has been done for transforming from specification to implementation in object-oriented environments; therefore, we investigate this area.

The chosen specification language is Object-VDM and the target implementation language is Eiffel. In this research, a modified refinement method, a methodology to transform Object-VDM to Eiffel, is presented. This method is based on refinement method in VDM [Jon 90]. The modified refinement method has three steps: data
refinement, operation refinement, and structure refinement. In data refinement, the mathematical data models in object-VDM such as SET, SEQUENCE, and MAP are converted to Eiffel data structures. In operation refinement, we modified and added rules in the original refinement method to obtain Eiffel code. In the structure refinement, object-oriented features are transformed. The rest of this chapter describes the summary of a formal specification language, object-oriented paradigm, Eiffel, and organization of the dissertation.

1.2 **FORMAL SPECIFICATION LANGUAGES**

Formal specification languages used in developing computer systems are mathematically based techniques for describing system properties. The benefits of using formal specification languages include[Win 90]:

- Emphasis on what a system does rather than how the system works.
- Accurate, detailed, and concise documentation of system functions.
- Ability to test the system before code is available.
- Maintenance assistance by providing an unambiguous interpretation of the requirements.

VDM [Jon 80][Jon 90], Z [Spi 88a] [Spi 88b], and LOTOS [EVD 89] are some widely used formal specification languages. Vienna Development Method (VDM) was developed in the IBM laboratory in Vienna. VDM is a model-oriented specification method in which the behavior of a software system is specified by defining abstract data types which model the state of the system, operations and functions on these types. VDM is based on set theory and first order predicate logic. There are two parts in VDM-the implicit specification and the explicit specification. The implicit specification part consists of external variable declarations, pre-conditions, and post-conditions.
The explicit specification part, which uses programming language-like control structures, is mainly used for implementation. Since we use Eiffel for implementation, we use only the implicit specification part of the VDM language. The structure of the implicit specification of operations in VDM is given in Figure 1.2.

In the structure, ext specifies a set of external variable declarations which are either rd (for read only access) or wr (for write-and-read access). The pre construct specifies a precondition which is a predicate that must be satisfied at the beginning of the execution of the operation. The post construct specifies a postcondition which is a predicate that must be satisfied at the end of the execution of the operation. In the post condition, the variable of the previous state is denoted with ← (e.g. var).

More details on VDM can be found [AI 91] and [Daw 91].

To illustrate the use of VDM, we describe a triangle manipulation system. The triangle manipulation system has two classes. One class is triangle for general triangles and another class is equilateral triangle. In a hierarchical structure, triangle is a super class of equilateral triangle. The points of a triangle are denoted by (x,y) coordinates, and the sides use vectors. The position attribute is the vector from the origin to the first point in a triangle. Some of operations for vectors are as follows:

* Vector addition $V + V : \text{Vector} \times \text{Vector} \to \text{Vector}$
* Vector modulus $|V| : \text{Vector} \to \text{Scalar}$
* Vector dot product $\mathbf{V} \cdot \mathbf{V} : \text{Vector} \times \text{Vector} \rightarrow \text{Scalar}$

* Vector rotation $\mathbf{V} \odot \theta : \text{Vector} \times \text{Angle} \rightarrow \text{Vector}$ The general triangle has the following attributes and properties:

\[
\begin{align*}
v_1, v_2, v_3 & : \text{Vector} \\
\text{position} & : \text{Vector} \\
p_1, p_2, p_3 & : \text{Point} \\
v_1 + v_2 + v_3 & = 0 \land p_1 = \text{position} \land p_2 = p_1 + v_1 \land p_3 = p_2 + v_2
\end{align*}
\]

The operations in the triangle system are MOVE and ROTATE, given in Figure 1.3.

**MOVE(v: Vector)**

- **ext wr** position: Vector
- **pre**
  - **post** position = position + v

**ROTATE(\theta : \text{Angle})**

- **ext wr** v1, v2, v3: Vector
- **pre**
  - **post** v1 = v1 \odot \theta \land v2 = v2 \odot \theta \land v3 = v3 \odot \theta

Figure 1.3: VDM specification of the triangle manipulation system

We examine the notation in the MOVE operation in Figure 1.3. The external variable (ext) position is declared writable (wr). The previous state of position is expressed by position. The precondition (pre) is empty, which means no restriction is given in order to begin the execution of a MOVE operation. In the postcondition (post), the current state of position is the vector sum of the previous state of position (position) and v. In summary, VDM is a formal specification language that can specify a system accurately in the early stage of the software life cycle. It has clauses for external variables, preconditions and postconditions which are expressed with mathematical notations and predicate logic symbols.
1.3 OBJECT-ORIENTED PARADIGM

The object-oriented paradigm includes three primary concepts: objects, classes, and inheritance [Weg 87]. An object is an entity that has state, behavior, and identity. A class is a set of objects that share common structure and behavior. Inheritance is a relationship between classes to express specialization and generalization of the concepts. Inheritance is a powerful feature that provides for the reusability and extensibility of software components. There are two kinds of inheritance: incremental inheritance and subtyping inheritance. Incremental inheritance adds 'attributes' and 'operations' to an existing class to get a new class. It reuses the code of superclass, but it does not guarantee that subclass is a specialization of the superclass. Subtyping inheritance, on the other hand, arranges classes in a hierarchical structure so the members of the subclass are also members of the superclass. Subtyping is a limited refinement of the superclass, subject to the substitutability condition which is that an instance of a subtype always can be used in any context in which an instance of the supertype was expected. [Ame 90][Cus 91].

Other features in the object-oriented paradigm are encapsulation, information hiding, and polymorphism. An object is encapsulated if the notion of an operation set and a data set are incorporated in a single entity (i.e. the object). Furthermore, the client should be restricted to accessing the object only through the well defined, external, operational interface. Information hiding implies that a sender does not know how the request is handled by the receiver. Polymorphism occurs when objects respond to the same message with different methods.

Initially, object-oriented methods were applied primarily during the implementation phase using object-oriented languages. C++, Smalltalk, CLOS, and Eiffel are some of the widely known object-oriented languages. The Eiffel language is explained in the Section 1.4.
The object-oriented paradigm is now being applied to other phases of the software development process. Many researchers have tried to apply the object-oriented paradigm to existing formal specification languages. Object-Z [DD 90], MOOZ [MC 91], OOZE [AI 91], Z++ [Lan 91] are some well known object-oriented Z specification languages. VDM also has its object-oriented extensions. In Chapter 3, we introduce a new specification language, Object-VDM. We then review existing object-oriented VDM extensions. We investigate the strengths and limitations of each of the object-oriented extensions to VDM.

1.4 EIFFEL: AN OBJECT-ORIENTED LANGUAGE

The Eiffel language is an object-oriented language designed by Meyer. Eiffel was designed to consider the following factors. The most important factor is reusability which is the ability to produce components that may be used in many different applications. Another factor is extensibility. The third factor is reliability. To reduce errors, Eiffel has facilities for assertions, disciplined exception handling, and static typing. Three other important factors are efficiency, openness, and portability. To achieve reusability, extendibility, and reliability [Mey 94a], Eiffel is designed as an object-oriented language. Eiffel has the following object-oriented features such as classes, information hiding, encapsulation, inheritance, and polymorphism. In addition to these basic facilities for object-oriented programming language, Eiffel also has many other useful facilities [Mey 88].

One of the most important facilities in Eiffel is assertions. By using assertions, developer can state precisely the formal properties of software elements. Assertions may be used to enhance the correctness and reliability of the resulting software. The underlying theory is design by contract [Mey 90] which views the correctness of a software system as the fulfillment of the many small and large contracts between clients and suppliers. The assertion of Eiffel is limited to boolean expression with a
few exceptions. The require clause indicates the pre-condition whereas the ensure clause indicates post-condition. The invariant clause expresses properties which must be ensured on instance creation and maintained by every exported routines. We can use assertions in the following cases:

- the precondition and postcondition part of a routine
- the invariant clause of a class
- the check instruction
- the invariant of a loop instruction

Eiffel has the genericity facility. Classes can have formal generic parameters representing types. For example, class ARRAY[T] has generic parameter T, which can be integers, real numbers, points, etc. There are two kinds of genericity: unconstrained genericity and constrained genericity. In unconstrained genericity, any type is acceptable as a actual generic parameter. In some cases, however, we will need a guarantee that types possess specific properties so that the class text may apply certain operations to the corresponding objects. In constrained genericity, types cannot be arbitrary. Constrained generic parameters must have the types which support certain operations.

Genericity and inheritance are two important techniques related to software quality. If a programming language has both inheritance and Ada-like genericity, it would result in a redundant and overly complex design. If a programming language includes only inheritance, programmers would have difficulty handling the simple cases for which unconstrained genericity offers an elegant expression mechanism. Therefore, in Eiffel the borderline was put at unconstrained genericity. Classes may have unconstrained generic parameters. Constrained generic parameters are treated through inheritance.
The Eiffel language can be considered as a design language. Deferred classes in Eiffel are particularly useful at the design stage. They describe a group of implementation of an abstract data type rather than just a single implementation. The first version of a system may be a deferred class, which will later be defined into one or more effective (non-deferred) classes. Particularly important to this application is the possibility to associate a precondition and a postcondition to a routine even though it is a deferred routine, and an invariant to a class even though it is a deferred class. Assertions make the designer to attach precise semantics to a module at the design stage, long before making any implementation choices. These facilities make Eiffel an attractive alternatives to PDL (Program Design Language) and other traditional design methods as structured design.

Eiffel has many libraries. Examples of these libraries are

- Kernel library
  - includes core component classes such as ANY and other classes for general data types such as ARRAY and STRING

- Support library
  - has classes which supports memory management, browsing, debugging, access to internals of Eiffel structures

- Data structure library
  - supports classes for basic data structures and algorithms such as lists, ques, stacks, trees

- Graphics library
  - includes classes for graphics, window manipulation, graphics user interface
Lexical library

has classes for scanning the text.

Parsing library

supports classes to analyze programs and documents.

Winpack library

has classes for non-graphical windowing.

Eiffel has five control mechanisms: sequencing, null instruction, conditional, multi-branch choice, and loop. The control mechanisms in Eiffel are almost the same as those in traditional languages: however, loops have some non-standard features. Eiffel supports only one form of loop. A single, general form is easy to learn and remember, and everything else may be programmed from it, while traditional programming languages offer five or six variants for loops such as testing at the beginning, the end or the middle, direct, or reverse condition "for" loop offering automatic transition to the next element. Figure 1.4 is an example of Eiffel code, which describes a STACK class with one operation, pop.

In Figure 4, the expert clause lists three attributes (nb-elements, empty and full) and two routines(pop, push). These attributes and routines are available to clients. The precondition and postcondition are written in the require clause and ensure clause respectively. The special notation old, which is permitted only in the postcondition, is used the postcondition of pop. The notation old nb-elements means the value of nb-elements on a pop routine entry.

1.5 OUTLINE OF THE DISSERTATION

In Chapter 2, we examine the related work, including refinement calculus and refinement methods. We investigate existing object-oriented extensions to VDM and.
class STACK[T]
export
    nb-elements, empty, full, pop, push
feature
    pop is
        require
            nb-elements >= 0;
        do
            nb-elements := nb-elements - 1;
            "other instructions to complete the pop operation"
        ensure
            nb-elements := old nb-elements - 1; ...
        end - pop
        .............
        .............
    end - class STACK

Figure 1.4: Eiffel code for the STACK class

identify their limitations in Chapter 3. Then we create Object-VDM, a new VDM extension for object-oriented systems. Object-VDM has facilities for classes, objects, inheritance, encapsulation, and polymorphism. The refinement process begins with an Object-VDM specification as the description of the desired system.

In Chapter 4 we define the refinement process from Object-VDM to Eiffel. There are three stages in the refinement: data refinement, operation refinement and structure refinement. In data refinement, we convert data structures and related operations in Object-VDM to those of Eiffel. The three basic mathematical data structures in VDM are sets, sequences and maps. In order to convert these structures to Eiffel, class libraries for sets, sequences and maps should be constructed. The refinement of predicates(operations) in VDM to the programming languages is studied in operation refinement. The logical operators ¬ (not), ∧ (and), ∨(or) and quantifiers such as ∃(existential quantifier) and ∀(universal quantifier) are converted to programming constructs. In structure refinement, object-oriented facilities are examined to con-
vert object-oriented VDM to Eiffel. Since object-oriented VDM and Eiffel are both object-oriented languages, some object-oriented features can be transformed directly. Classes, inheritance, polymorphism, and initialization must be examined to convert them to the Eiffel code. We verify the refinements by using LPF (logic of partial functions).

Chapter 5 gives a case study to demonstrate the process. Finally, the conclusions and summary are presented in Chapter 6.
CHAPTER 2
RELATED RESEARCH

2.1 FORMAL DEVELOPMENT METHODS

Traditional software development uses different methods and techniques in each phase between specification and implementation. For example, a developer may use a data flow diagram in the specification stage, a structured chart in the design stage and a programming language in the implementation phase. To reduce the radical change in methods and techniques between phases, stepwise refinement was introduced[Wir 71][Dij 72]. Stepwise refinement starts from an abstract requirements specification and proceeds through more and more concrete versions of the program, mostly written in some kind of pseudo code, until the actual program code is produced. In each step a task is divided into a number of subtasks. Refinement of the data structures and algorithms (or operations) should proceed in parallel. The major problem with stepwise refinement is that all processes are informal such that correctness of the system relies on unrigorous methods like code walk-throughs and structured testing.

To solve this problem, formal development methods [San 88][MPS 93] were suggested. Formal development methods are mathematically based approaches to software development that support the rigorous specification, design, and verification of computer system. Formal development methods consist of a formal specification and a verified design. A formal specification language can specify a system more accurately, consistently, and completely by using mathematical symbols and logic. A verified design is developing the system from specification to executable code by using formal proofs or verified rules. Formal development methods can be classified as verification or transformational. Both approaches begin with a formal specification and develop the final program using formally provable steps.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
In the verification approach, the specification is written in a specification language but the implementation is written in a programming language. In this approach, the more concrete version is written and then the concrete version is shown to meet its requirements by a sequence of formal proofs. Any two successive stages in the development process have to be proved equivalent. It is usually not possible to verify directly the implementation against the specification. The great gap between the implementation and the specification generally makes the proofs too complex. Therefore, an evolution process that requires proceeding through a series of intermediate stages had to be adopted in the verification approach in order to keep the individual proofs manageable. Correctness of the formal development is established by discharging defined proof obligations. VDM uses this approach.

In the transformational approach, the evolutionary concept of a stepwise development process is combined with the mathematical rigor of the verification approach; however, it takes a slightly different view: In a verification-oriented development, the next version of program is constructed "on speculation", and then an "a posteriori" verification of its equivalence to the previous version is performed. By contrast, the transformation idea is to derive the next version from the previous one according to pre-verified, formal rules. In other words, one performs a constructive derivation, leading from the initial specification to the final program code. The initial specification is precisely defined, yet neither very detailed, nor efficient. By stepwise refinement and semantics-preserving transitions, the program becomes more efficient and detailed. Compared to the verification approach, the resulting development steps are much smaller. The transformation approaches are due to Gerhart [Ger 75] and Burstall and Darlington [BD 77]. Common patterns of transformations can be packed into derived steps. The most basic transformation rules are unfold and fold [BD 77]. Fold is a formation of a (recursive) call from an expression which is an instance of some function body, or the instructions of an identifier for an certain expression. Un-
fold is the reverse of fold. Unfold replaces a function call by the body of the function, with replacement of the formal parameters by the actual parameters. Fold and unfold must also be supplemented by laws on the data types and the introduction of new definitions based on intuition. Well-known examples of this approach are the Munich CIP project, PROSPECTRA, and refinement calculus. Munich CIP project [Bau 88][CIP 85] and PROSPECTRA [Kri 87][Kri 88] are based on algebraic specifications, while refinement calculus [Mor 90] is based on model-oriented specification.

2.2 REFINEMENT CALCULUS AND REFINEMENT METHODS

The refinement approach is one of the formal development methods that inputs a specification and outputs an implementation. In the refinement approach, an abstract specification is progressively transformed to a more concrete specification. We say that P is refined by Q if any specification specified by P is also satisfied by Q. This relation can appear between abstract program (specification) to concrete program (specification). The specification is refined until executable operations and data are obtained. The refinement approach has two kinds of processes: operation refinement and data refinement. The operation refinement refines the operations to produce executable equivalents. The data refinement replaces the abstract representations of the data to concrete equivalents. Refinement approaches can be used for a various specification. Refinement with algebraic specification languages is developed in [Gou 90] and the use of interpretation between types are described in [Tur 87]. For model-oriented specification languages such as VDM and Z, we use the refinement method [Jon 90], and the refinement calculus [Mor 90]. Refinement calculus is based on the transformational approach, while the refinement method is based on the verification approach. We briefly review these approaches.
2.2.1 Refinement Calculus

The refinement calculus is a notation and set of rules to derive implementations from specification by using Dijkstra's calculus [Dij 75][Dij 76][Gri 81]. The refinement calculus unifies the stepwise refinement and program transformational approaches to program construction. The basis notion in this calculus is the relation of refinement between nondeterministic specifications or programs. The refinement calculus was first introduced by Back [Bac 78][Bac 80][Bac 81b], and was reinvented by Morris [Mor 87] and by Morgan [Mor 88]. Since this calculus concerns the correct refinement, weakest precondition is its central part. The weakest precondition is originated by Dijkstra [Dij 76]. The refinement calculus extends the weakest precondition techniques of Dijkstra's calculus to procedural and data refinement. This method uses the combination of specification statements and Dijkstra's guarded command language. The specification statement is a non-executable construct with the form

\[ w \{pre, post\} \]

where \( w \) is the frame and \( pre \) and \( post \) is precondition and postcondition. This statement is interpreted as follows [Kin 90]:

If the initial state is described by \( pre \), then by changing only the variables listed in the frame \( w \), establish some final state described by the postcondition \( post \).

The Dijkstra's guarded command language is the notation of executable elements of refinement calculus, and can have the following control structures.

- sequential composition
- assignment
- alternation
- iteration
The final product of the refinement process only uses the guarded command language. In refinement calculus, transformation rules are created to transform from abstract specifications to concrete specifications. These rules are generally presented as before_refinement $\subseteq$ after_refinement. No proof obligation is needed in refinement calculus. Refinement calculus is mainly used in development using Z specification. In refinement calculus, the term "procedural refinement" is used for operation refinement.

**OPERATION REFINEMENT** In operation (procedural) refinement, we eliminate specification statements in order to produce programs consisting only of executable statements. There are two kinds of rules: rules for the derivation of language statements and rules for manipulation of the specification. Rules to derive of language statements include:

- **simple assignment**: $[P,Q] \subseteq x := e \iff P \Rightarrow Q [x \ e]$.
  
  The language statement $x := e$ is derived from the specification $[P,Q]$ if the precondition $P$ of specification implies $Q$ with substitution of $x$ with $e$.

- **sequential composition**: $[P,Q] \subseteq [R,S]; [T,U] \iff P \Rightarrow R$, $S \Rightarrow T$, and $U \Rightarrow Q$.
  
  The specification $[P,Q]$ can be developed by $[R,S];[T,U]$ if $P$ implies $R$, $S$ implies $T$ and $U$ implies $Q$.

- **alternate construct**: $[P,Q] \subseteq$ if $B_1 \rightarrow [P \land B_1, Q] | ... | B_n \rightarrow [P \land B_n, Q]$ fi $\iff P \Rightarrow B_1 \lor ... \lor B_n$.
  
  The control structure (if) can be derived if $P$ implies $B_1 \lor ... \lor B_n$.

The following rules are for manipulation of the specification.

- **replacing specification by another specification**: $[P,Q] \subseteq [R,S]$ iff $(P \Rightarrow R \lor S \Rightarrow Q) \lor \neg P$
We can replace \([P,Q]\) by \([R,S]\) if and only if \(P\) implies \(R\) or \(S\) implies \(Q\) or \(P\) is not satisfied.

- **combining specifications:** if \([P,Q]\subseteq T\), and \([R,S]\subseteq T\) then \([P \land R, Q \land S]\subseteq T\) and \([P \lor R, Q \lor S]\subseteq T\).

If two specifications are refined by the same specification, the specification whose precondition is a conjunction (or disjunction) of preconditions of two specifications and postcondition is conjunction (or disjunction) of postconditions of two specifications is also refined by the same specification.

**DATA REFINEMENT** Data refinement replaces abstract data types by concrete data types. A typical sequence of steps in refinement calculus is operation refinement followed by data refinement. The abstract specification is transformed into an abstract program, and the abstract program is transformed into a program on the concrete type. Data refinement in the refinement calculus is considered a special case of operation refinement. It is the process of replacing abstract local variables by concrete local variables such that this replacement does not change the overall effect of the program. Data refinement is characterized as procedural refinement between blocks, i.e. given the block \([[\text{var} \ a: \ T_a; \ A]]\), its replacement \([[\text{var} \ c: \ T_c; \ C]]\) must satisfy

\([[\text{var} \ a: \ T_a; \ A]] \subseteq [[\text{var} \ c: \ T_c; \ C]]\)

The concrete variable \(c\) and concrete program \(C\) must be a concrete refinement of abstract program \(A\) on abstract variable \(a\). The relationship between the abstract and concrete variables is made by defining an *invariant relation* (or *retrieve relation*) which is denoted \(\leq_{f,a}\). \(A \leq_{f,a} C\) means abstract program \(A\) on abstract variable \(a\) is data-refined to concrete program \(C\) under invariant relation. Its formal definition \([\text{Mor } 90][\text{Mor } 90]\), is

\[ A \leq C = (\exists a : I \land wp(A,R)) \Rightarrow wp(C, (\exists a : I \land R)) \]
There are two kinds of the data refinement rules: structure preserving rules and rules for refinement of primitives. Structure preserving rules include:

- **sequential composition:** A; B ≤ C; D if A ≤ C and B ≤ D. The abstract program which has two sequential routines A and B can be converted to the concrete program which has two sequential routines C and D, when A is refined to C and B is refined to D.

- **common local variable:** if A ≤ C then |[var k: Tk; A]| ≤ |[var k: Tk; C]| When an abstract program A is refined a concrete program C, the extended programs which include the common local variables still have refinement relationship.

For the data refinement of primitives, miracle[Mor 90] is used.

### 2.2.2 Refinement Methods

In the refinement method [Jon 90], the structure of a typical refinement step is to write a more concrete version of the entity being refined, and to show that the (more) concrete version meets the requirements by a sequence of formal proofs, often called the discharging of proof obligations. Three refinement methods are widely known: VDM refinement methods[Jon 90], IBM Hursley Park method[Joh 88][Kin 89][Wor 89], and rigorous refinement method for Z[Nei 87]. VDM refinement methods are described in Section 2.3, and the other methods are described briefly here.

- **IBM HURSLEY PARK METHOD** This method uses Z specifications and Dijkstra's guarded command language, developed at IBM UK Laboratories, Hursley Park[Kin 89][Joh 88][Wor 88]. In the further theoretical development[Wor 89], the IBM Hursley method moves closer to the transformational approach of the refinement calculus discussed earlier. High-level design is the term for data refinement and low-level design is the term for algorithm refinement in this method.
HIGH-LEVEL DESIGN  High-level design transforms the abstract, mathematical state into a concrete implementation-oriented state. In data refinement, a concrete state has been selected, and then, the mathematical relation between concrete and abstract state defined with a retrieve schema(relation). The retrieve schema has an abstract state, a concrete state, and a predicate which defines how their components are related. The relationship schema is not a function but a binary relation. Upon recording the retrieve relation, the initial concrete state is defined. A proof obligation for initialization is needed to show that the concrete state corresponds the abstract state. Related operations to the refined data structures should be respecified. The proof obligations of applicability and correctness are required. These proof obligations are the domain rule and result rule in refinement in VDM. These rules will be discussed in Section 2.3.1.

LOW-LEVEL DESIGN  The low-level design is an adapted form of stepwise refinement. The design consists of following steps.

• choose a control structure for the operation

• record it using Dijkstra's guarded command language, and give names to the subcomponents

• write specifications for the subcomponents.

This process is continued until all subcomponents are specified using only primitive statements.

• RIGOROUS REFINEMENT METHOD FOR Z  Neilson developed this method. Like IBM Hursley Park method, this method uses Z as a specification language and guarded command language as a target implementation language. This method anticipates the refinement calculus as one of the rigorous, transformational refinement.
The method uses transformational rules instead of reference rules. It has a feature to handle operators from the Z schema calculus.

**DATA REFINEMENT** In this method, extract function is used for retrieve function. The extract should be total. The weakest concrete operation corresponding to abstract operation can be derived. The weakest concrete operation is the loosest possible specification of an operation on the concrete state, corresponding to an abstract counterpart. We need to introduce some extra constraints to reflect implementation details.

**OPERATIONAL REFINEMENT** The rules are defined in terms of an order relation (⊆). An operation $A$ is refined to operation $B$ (written $A \subseteq B$) if it satisfies the domain condition and the safety condition. The domain condition requires that when $A$ is applicable, $B$ is applicable. The safety condition requires when $A$ is applicable, the results produced by $B$ imply those produced by $A$.

### 2.3 REFINEMENT IN VDM

At each step, a specification closer to an implementation is written and then proved to meet the original specification. The proofs necessary are described by a set of proof obligations [Jon 90]. There are two stages: data refinement (reification) and operation refinement (decomposition).

#### 2.3.1 DATA REFINEMENT

Data refinement translates the abstract, mathematical data type of the specification into the (more) concrete data types which can be implemented. To prove that the implementation state satisfies the specification state, we need a retrieve function which establishes a link between two states. The relation between abstract and representation values is one-to-many, since each value of an abstract type has more than one possible representations. The relationship between ab-
abstract values and their representation is expressed by a retrieve function from the latter to the former. A retrieve function requires two properties: **total** and **adequacy**. A function is **total** if \( \text{dom}(f) = X \cdot f : X \rightarrow Y \). To make a retrieve function total, sometimes it is necessary to tighten an invariant on the representation in order to ensure that the retrieve function is defined for all values which can arise. In the retrieve function, there should be at least one representation for any abstract value. This property is called **adequacy**. It can be written by the following notation: \( \forall a \in \text{Abstract} \cdot \exists r \in \text{Representation} \cdot \text{retr}(r) = a \) for \( \text{retr} : \text{Representation} \rightarrow \text{Abstract} \). After the state is transformed, we have to respectify the operations on the chosen state representation. This is known as **operation modeling**. Representation detail forces operation specifications to be more complex and algorithmic. The proof obligations needed for operations are the domain rule and the result rule. The domain and result rules are:

**domain rule** -
\[
\forall r \in R \cdot \text{pre-A(retr}(r)) \Rightarrow \text{pre-R}(r)
\]

The domain rule requires that the precondition of the operation on the representation is broader than that on the specification. If the specification of the abstract operation is true of a retrieved state, the representation state must satisfy the precondition of the representation operation.

**result rule** -
\[
\forall \overrightarrow{r}, r \in R \cdot \text{pre-A(retr}(r)) \land \text{post-R (} \overrightarrow{r}, r) \\
\Rightarrow \text{post-A(retr}( \overrightarrow{r} ), \text{retr}(r))
\]

where pre-A and post-A is pre- and post-condition of abstract and pre-R and post-R are the pre- and post-condition of representation.
The result rule requires that any pair of states in the post-R relation must satisfy post-A relation so that operation R and A model the same behavior. The first conjunctive of the antecedent of the implication requires states to satisfy the abstract precondition.

2.3.2 OPERATION REFINEMENT

After data refinement, we need operation refinement (decomposition) to get executable operations. The process of operation decomposition develops implementations in terms of the primitives available in the language and support software. The control constructs which are used to link the primitive instructions can be thought of as combinators. Operations in VDM are expressed in pre- and post-conditions. These conditions are logic predicates which use logical connectives in their expression. There are six logical connectives in VDM: negation, disjunction, conjunction, implication, equivalence, and tautologies. We have to convert these logical connectives into control structures to implement the system. There are three control structures in Eiffel (or general programming languages): sequence, alternate and iteration.

We need rules to decompose the operations in VDM. The underlying logics for these rules are Hoare's logic and LPF (logic of partial function). Decomposition rules from abstract VDM to concrete VDM were investigated by Cliff Jones [Jon 90]. Here are some refinement rules.

The decomposition rule for sequential refinement is

\{pre_1\} S_1\{pre_2 \land post_1\};\{pre_2\} S_2\{post_2\};
{pre_1}(S_1; S_2)\{post_1 | post_2\};

where the composition of two post-conditions is defined:

\[ post_1 | post_2 = \exists \sigma_i \in \Sigma \cdot post_1(\widetilde{\sigma}, \sigma_i) \land post_2(\sigma_i, \sigma) \]

To decompose an operation S by the sequential operations S_1; S_2, the following properties should be observed.
- the first operation $S_1$ can be applied in the precondition of $S$: compare $\text{pre-}S_1$ with $\text{pre-}S$.

- the second operation can safely be applied in the states with result from executing the first operation: compare $\text{pre-}S_2$ with $\text{post-}S_1$.

- the composition of the effects of the two operations achieves the required effect of $S$: compare $\text{post-}S_1 \mid \text{post-}S_2$ with $\text{post-}S$.

The decomposition rule for the conditional statement is

$$\{\text{pre} \land \text{test}\} \text{TH} \{\text{post}\}; \{\text{pre} \land \neg \text{test}\} \text{EL} \{\text{post}\}; \text{pre} \Rightarrow \delta(\text{test})$$

$$\{\text{pre}\} \{\text{if test then \text{TH} else \text{EL} end}\} \{\text{post}\}$$

The operation $S$ is refined to a control (if) statement, when the precondition of TH satisfies $\text{test}$ condition and precondition of $S$, and the precondition of ELSE does not satisfy $\text{test}$ condition and satisfies precondition of $S$. The logical expression in the precondition is only valid if they are defined ($\delta$) in the programming language.

The decomposition rule for loops is

$$\{\text{inv} \land \text{test}\} S \{\text{inv} \land \text{sofar}\}; \text{inv} \Rightarrow \sigma_1(\text{test})$$

$$\{\text{inv}\} \{\text{while test do S}\} \{\text{inv} \land \neg \text{test} \land (\text{sofar} \lor \text{idem})\}$$

A loop invariant ($\text{inv} : \Sigma \rightarrow B$) is identified which limits the states which can arise in the computation and that a relation ($\text{sofar} : \Sigma \times \Sigma \rightarrow B$) is given which holds over one or more iteration of the loop; technically the requirement that

$$(\text{sofar} \mid \text{sofar} \Rightarrow \text{sofar})$$

is stated by saying that sofar must be transitive.

It is also necessary to ensure the termination and this can be done by ensuring that the $\text{sofar}$ is well-founded over the set defined by $\text{inv}$.

Since, most specifications do not exactly fit the conditions of decomposition rules, we provide a rule which allows a decomposition to be performed on a 'weaker' specification.

$$\{\text{pre}_w\} \Rightarrow \text{pre}; \{\text{pre}\} S \{\text{post}\}; \text{post} \Rightarrow \text{post}_w$$

$$\{\text{pre}_w\} S \{\text{post}_w\}$$
This rule asserts that anything which satisfies a specification necessarily satisfies a weaker one. Observe that a 'weaker' specification is one with a narrower pre-condition or a wider post-condition. In either case, the implication could be just an equivalence thus changing only the other part of specification.

2.4 OTHER APPROACHES TO CONVERT FROM SPECIFICATION TO PROGRAMMING LANGUAGE

There are other research initiatives that convert a formal specification language to an implementation. Four such initiatives are the conversion from Z to Anna, conversion from Object-Z to C++, conversion from VDM to ADA, and conversion from MooZ to Eiffel.

The conversion from Z to Anna was studied by W. Wood and P. Place [Woo 91]. This research describes a method for the formal development of Ada Programs from a formal specification written in Z. ANNotated Ada (Anna) is used as an intermediate language linking the more abstract Z specifications to the concrete Ada program. The method relies on the notion that successive small conversions of a specification are easier to verify than a few large conversions. Essentially the method uses three notations for the representation of the system: an implementation-independent notation for the specification of the system, an implementation-dependent notation for the representation of a lower level specification of the system, and implementation language.

Conversions from Object-Z to C++ were exploited by W. Johnston and G. Rose [Joh 93]. Since Object-Z and C++ are both object-oriented languages, there is a fairly straightforward mapping at the class level from Object-Z to C++. This research suggested how the facilities for object-oriented system, such as inheritance, class constructor, and polymorphism can be transformed. But their approach is informal. There is no formal proof that their conversion is semantics preserving.
VDM development with ADA as the target language was examined by David O'Neil [O'Ne 88]. A set of generic Ada packages and package generators has been defined to implement VDM domains and domain constructors and their associated operations. Based on [Pre 83], and on some experience in manually translating VDM specification into ADA, the translation of the various expression forms offered by the VDM (i.e., forms of object construction, conditional expression, etc.) has been considered.

In "From MooZ to Eiffel" [CSM 94], a MooZ specification is refined into Eiffel program. The method can be classified as intermediary between a refinement method and a refinement calculus. The approach to data refinement is a kind of refinement method. However, the approach for operation refinement is refinement calculus.

2.5 SUMMARY

This chapter has described related research. The formal development method, refinement method and refinement calculus are described. The two research initiatives most related to this research are the refinement method in VDM and refinement from MooZ to Eiffel.

The research defined here is to apply the refinement method in VDM for object-oriented paradigm. In the original refinement methods in VDM, explicit notation of VDM for implementation is assumed. Since the target language is Eiffel, we have to modify rules for implementation. We also have made rules to convert object-oriented facilities. Previously, there was no research for development VDM in object-oriented paradigm.

The research transformation from MooZ to Eiffel is another closely related research initiative; however, it mixes the refinement method and the refinement calculus and used the guarded command language. The specification language is not based on VDM but Z. Since we use the expanded refinement method in VDM, the
MooZ to Eiffel method is quite different from our research, especially in the operation refinement area. Their research also does not allow translation of mathematical data structures such as set, sequence, and map. They do not transform logical operators.
CHAPTER 3
OBJECT-ORIENTATION IN VDM

3.1 INTRODUCTION

We propose Object-VDM, an object-oriented VDM, which is based on the current VDM standard in Section 3.2. Then, we give an example in Section 3.3 to show how Object-VDM is used to specify a system. We investigate existing object-oriented VDM extensions: Fresco[Wil 92a], QoVDM[LS 93], and VDM++ [DV 92][DP 95]. In Section 3.4, we review these languages. Section 3.5 summarizes this Chapter.

3.2 OBJECT-VDM

There are two limitations to the existing object-oriented VDM extensions. One is that they do not have all necessary facilities to support the object-oriented paradigm as we will point out in section 3.4.4. Another is they are not fully based on the current VDM standard. The most significant feature in VDM standard is the structuring of VDM. The overall structure of the current VDM standard is as follows[Daw 91][And 93][Par 94].

\begin{verbatim}
types definition block
state definition
values definition block
functions definition block
operations definition block
\end{verbatim}

Since VDM has a standard, an object-oriented extension fully based on the VDM standard is needed. The details of the current VDM standard can be obtained from ISO or British Standard Institute [And 93][LLdB 94]. We extend the VDM standard to add object-oriented facilities for classes, objects, inheritance, encapsulation, and polymorphism. Object-VDM has two modules: class modules and type modules. The overall syntactic structure of Object-VDM is given in Figure 3.1.
class class name([generic parameters])
superclass class name list
public/private [operation name list]|[function name list] | ALL
values pattern=expression
state identifier of field list
  inv pattern^ expression
  init pattern^ expression
end
functions
  function name * (parameters)identifier:type
    [ pre expression]
    post expression
operations
  operation name * (parameters)[identifier:type]
    ext rd | wr variable list
    [ pre expression]
    post expression
endclass
[Class Module]
type typename
variable list
supertype type name list
value
  value name : type expression
axiom
  expression
endtype
[Type Module]

Figure 3.1: Structure of an Object-VDM specification
As shown in the Figure 3.1, Object-VDM adds superclass, public/private clauses, and a type module to the VDM standard. The class clause has its name and optional generic parameters. The keyword superclass can be used to define the classes whose variables, methods, class invariants, and initializations are inherited. Multiple superclasses are allowed. The public/private clause defines the operations (methods) that are externally visible or not accessible. If a class has more public definitions than private ones, it is more effective to use the private clause. If all definitions in a clause are considered private (or public), private (public) all is used. If there is no public/private clause, all definitions are public. The values represents values which can not be changed by any operations. The state clause defines the state variables with their attributes. The invariant denotes the class invariants. The invariants should be satisfied for all methods and variables in the class. The init clause is a schema specifying allowable initial instances for the class. The operations clause defines the object behavior and consists of zero or more operations. The operation contains the followings.

1) a heading with the input and output parameters
2) external variables
3) pre- and post-conditions

External variables can be defined using the ext keyword. They can be rds(read only) or wrs(writable). It is the responsibility of the caller of the operation to ensure that the precondition is fulfilled. The postcondition may contain exception parts. The notation of the type module modifies the RAISE notation[Geo 91]. The type module consists of type declarations, supertype, value, and axiom declaration. The supertype defines the existing type whose attributes are inherited. The value defines the name of the function and its type while the axiom defines the restrictions of the functions. The detailed expressions such as predicate expression and type expression.
in object-VDM follow those of the VDM standard. The Object-VDM specification for the triangle example is given in Figure 3.2. The concrete syntax of Object-VDM follows that of the VDM standard. The four clauses including state, value, function, and operation clauses are presented here. Lower level syntactic definitions are given in Appendix A.

- **State Definition**

  state definition = 'state', identifier, 'of', field list, [invariant], [initialization] 'end';
  
invariant = 'inv', invariant initial function;
  initialization = 'init', invariant initial function;
  invariant initial function = pattern, '▹' expression;

- **Value Definitions**

  value definitions = 'values', value definition, { ';' value definition }
  value definition = pattern, [ ';' type '=' ], expression;

- **Function Definitions**

  function definitions = 'functions', function definition, { ';' function definition }
  function definition = explicit function definition
                     | implicit function definition
  explicit function definition = identifier, [ type variable list ], '}', function type,
                             identifier, parameters list,
                             '▹' expression
                             ['pre', expression],
  implicit function definition = identifier, [ type variable list ],
                             parameter types, identifier type pair,
                             ['pre', expression],
                             'post', expression;
type variable list = '[' , type variable identifier , ',' , type variable identifier , ']';

identifier type pair = identifier , ':' , type ;

parameter types = '(' , [ pattern type pair list] , ')';

pattern type pair list = pattern list , ':' , type , {',' , pattern list , ' ;' , type};

parameters list = parameters , parameters ;

parameters = '(' , [ pattern list] , ')';

- Operation Definitions

operation definitions = 'operations' , operation definition , '{' ; , operation definition} ;

operation definition = explicit operation definition  
| implicit operation definition ;

explicit operation definition = identifier , ':' , operation type ,
identifier , parameters ,
'△' , statement
[' pre' , expression],

implicit operation definition = identifier , parameter types , [ identifier type pair]
[externals] ,
[' pre' , expression],
'post' , expression ,
[exceptions];

operation type = discretionary type , '→' , discretionary type ;

externals = 'ext' , var information , { var information} ;

var information = mode , name list , '[' , type];

mode = 'rd' | 'wr';

exceptions = 'errs' , error list ;
3.3 An Example

We present an example of specifying a system using both VDM and Object-VDM. The example system computerizes a university system to manage student data. Every student has his/her name and identification number. Student data also includes earned credit hours and GPA.

The operations, such as adding new students, adding credit hours, changing GPA, and checking the eligibility to graduate are needed. Graduate students have additional requirements. Graduate students must pass the comprehensive exam first, propose the thesis, and finally defend the thesis. Therefore, graduate student data has the following additional attributes: status_exam, status_proposal, thesis_title, status_defence. To handle graduate students, operations such as reporting the pass of exam (exam), controlling the data for proposal (proposal), and defense of thesis (defence) are needed. The VDM standard specification for this system is written in Figure 3.3, 3.4 and 3.5, and the corresponding Object-VDM specification is in Figure 3.6 and 3.7. The student class is described in Figures 3.3 and 3.6, and the graduate student class is specified in Figures 3.4, 3.5 and 3.7.

In the add_credit operation, the earned credit hour and GPA are updated. For graduate requirements, ordinary students should earn 140 credit hours and a GPA greater than 2.0, while graduate students must earn 30 credit hours, have a GPA greater than 3.0, and defend his thesis. In the VDM notation, we have to write all attributes and operations again, even if student and graduate_student have many things in common: attributes in student are used in graduate_student and the operations such as add and add_credit are the same.
class Triangle
  public all
  state Triangle_Class of
    v1,v2,v3 : Vector
    position : Vector
    p1,p2,p3 : Point
  end
  inv  v1 + v2 + v3 = 0 ∧
        p1=position ∧ p2=p1 + v1 ∧ p3=p2 + v2
  operations
    move(v:vector)
      ext wr position
      pre
        post position = position + v
    rotate(θ:angle)
      ext wr v1,v2,v3: Vector
      pre
        post v1=v1 ⊕ θ ∧ v2=v2 ⊕ θ ∧ v3=v3 ⊕ θ
  endclass

class Equilateral_Triangle is
  superclass Triangle;
  operations
    area(s:real)
      pre
      post s=\frac{\sqrt{3}}{4}|v_1|^2
  endclass

Figure 3.2: Object-VDM specification of the triangle manipulation system
types
student:: id: NAT
    name: CHAR
    credit: NAT
    gpa: REAL
    finish: BOOL;
state Student_Class of
ST: student-set
st: student
end
operations
add(new_id:NAT, new_name:CHAR,
    new_credit: NAT, new_gpa:REAL)
ext wr ST:student-set
wr st:student
pre ¬∃ st ∈ ST . st.id=new_id
post st.id=new_id ∧ st.name=new_name ∧
    st.credit=new_credit ∧ st.gpa=new_gpa ∧
    ST=ST ∪ {st}
add_credit(id:NAT, h:NAT, g:REAL)
ext wr ST:student-set
wr st:student
pre ∃ st ∈ ST . st.id=id
post st.gpa=(st.credit * st.gpa + h * g)/(st.credit + h) ∧
    st.credit=st.credit + h ∧
    ST=ST \ {st} ∪ {st}
graduate
ext wr ST:student-set
wr st:student
pre ∀ st ∈ ST . st.credit >=140 ∧
    st.gpa>=2.0
post st.finish=T

Figure 3.3: VDM specification of the student records system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
types
gr_adeuate_student::
    id: NAT
    name: CHAR
    credit: NAT
    gpa: REAL
    status_exam: BOOLEAN
    status_proposal: BOOLEAN
    status_defence: BOOLEAN
    thesis_title: CHAR
    finish: BOOLEAN;
state Graduate_Student_Class of
    G_ST: graduate_student- set
    g_st: graduate_student
end
operations
  add(new_id:NAT, new_name:CHAR, new_credit: NAT, new_gpa:REAL)
    ext wr G_ST: graduate_student- set
    wr g_st: graduate_student
    pre $\exists g_st$ . g_st.id=new_id
    post g_st.id=new_id $\wedge$ g_st.name=new_name $\wedge$
        g_st.credit=new_credit $\wedge$ g_st.gpa=new_gpa $\wedge$
        G_ST= G_ST $\cup \{g_st\}$
  add_credit(id:NAT, h:NAT, g:REAL)
    ext wr G_ST
    wr g_st
    pre $\exists g_st$ $\in$ G_ST $\cdot$ g_st.id=id
    post g_st.gpa=( g_st.credit * g_st.gpa +h*g)/ g_st.credit +h $\wedge$
        g_st.credit$= g_st.credit + h \wedge$
        G_ST= G_ST \{g_st\} $\cup \{g_st\}$

Figure 3.4: (part 1) VDM specification of the graduate student records system
exam(id:NAT)
  ext wr G_ST: graduate_student- set
  wr g_st: graduate_student
  pre \exists g_st \in G_ST \cdot g_st.id=id
  post g_st.status_exam=T

proposal(id:NAT, title:CHAR)
  ext wr G_ST: graduate_student- set
  wr g_st: graduate_student
  pre \exists g_st \in G_ST \cdot g_st.id=id \land g_st.status_exam=T
  post g_st.status_proposal=T \land g_st.thesis_title=title

defence(id:NAT)
  ext wr G_ST: graduate_student- set
  wr g_st: graduate_student
  pre \exists g_st \in G_ST \cdot g_st.id=id \land g_st.status_proposal=T
  post g_st.status_defence=T

graduate
  ext wr G_ST: graduate_student- set
  wr g_st: graduate_student
  pre \exists g_st \in G_ST \cdot g_st.credit>=30 \land g_st.gpa>=3.0 \land
      g_st.status_exam=T \land g_st.status_defence=T
  post g_st.finish=T

Figure 3.5: (part 2) VDM specification of the graduate student records system


```plaintext
type student_type
  id: NAT
  name: CHAR
  credit: NAT
  gpa: REAL
  finish: BOOLEAN;
end type

type graduate_student_type
  supertype student_type
  status_exam: BOOLEAN
  status_proposal: BOOLEAN
  status_defence: BOOLEAN
  thesis_title: CHAR;
endtype

class Student
  state Student_class of
    ST: student_type set
    st: student_type
  end

operations
  add(new.id:NAT, new.name:CHAR, new.credit: NAT, new.gpa:REAL)
    ext wr ST: student_type set
    wr st: student_type
    pre ¬∃ st ∈ ST . st.id=new.id
    post st.id=new.id ∧ st.name=new.name ∧ st.credit=new.credit ∧
           st.gpa=new.gpa ∧ ST=ST \ {st}
  add_credit(id:NAT, h:NAT, g:REAL)
    ext wr ST: student_type set
    wr st: student_type
    pre ∃ st ∈ ST . st.id=id
    post st.gpa = (st.credit * st.gpa + h * g)/st.credit + h) ∧
           st.credit= st.credit + h ∧ ST=ST \ {st} \ {st}
  graduate
    ext wr ST: student_type set
    wr st: student_type
    pre ∀ st ∈ ST . st.credit >=140 ∧ st.gpa>=2.0
    post st.finish=T
endclass
```

Figure 3.6: Object-VDM specification of the student records system
class Graduate_Student
    superclass student
    state G_ST: graduate_student_type- set
        g_st: graduate_student_type
    end
    operations
        exam(id:NAT)
            ext wr G_ST:graduate_student_type- set
                wr g_st: graduate_student_type
                pre \exists g_st \in G_ST \cdot g_st.id=id
                post g_st.status_exam=T

        proposal(id:NAT, title:CHAR)
            ext wr G_ST:graduate_student_type- set
                wr g_st: graduate_student_type
                pre \exists g_st \in G_ST \cdot g_st.id=id \land g_st.status_exam=T
                post g_st.status_proposal=T \land g_st.thesis_title=title

        defence(id:NAT)
            ext wr G_ST:graduate_student_type- set
                wr g_st: graduate_student_type
                pre \exists g_st \in G_ST \cdot g_st.id=id \land g_st.status_proposal=T
                post g_st.status_defence=T

    graduate
        ext wr G_ST:graduate_student_type- set
            wr g_st: graduate_student_type
            pre \exists g_st \in G_ST \cdot g_st.credit>=30 \land
                g_st.gpa>=3.0 \land g_st.status_exam=T \land
                g_st.status_defence=T
            post g_st.finish=T
    end class

Figure 3.7: Object-VDM specification of the graduate student records system

In the Object-VDM example, we use both type modules and class modules. Since
graduate.student_type is a specialization of a student_type, a graduate.student_type
is a subtype of a student_type. Graduate.student class also uses the same operations
such as adding a new student (add), and adding new credit hours (add_credit). Thus, Graduate_S Student class uses operations inherited from the super class, Student class. Graduate requirements for graduate student differ from those of undergraduate students. Therefore, the Graduate_S Student class has a different graduate method.

3.4 SURVEY OF EXISTING OBJECT-ORIENTED VDM

Fresco[Wil92a], OQVDM[LS93], and VDM++ [DV92][DP95] are representative object-oriented extensions of VDM. We briefly introduce these languages and examine their strengths and weaknesses.

3.4.1 FRESCO

Fresco [Wil92a][Wil92b][Wil94] is the object-oriented software system used for rigorous development from specification to implementation. Fresco does not modulize specifications. It specifies program modules. A class can describe a specification in abstract phases and an implementation in a concrete phase. To transform from the abstract phase to a concrete phase, a mixture of specification and implementation is used. Fresco has two hierarchies: 'class hierarchy' and the 'type hierarchy'. The overall structure of a Fresco specification is as in Figure 3.8.

```
Class/TypeName ::= SuperClass/TypeName
visible operation signatures
operation specifications
private model variables and/or
private implementation
```

Figure 3.8: Overall structure of Fresco
An operation specification has the following format:

```
label::variables-[precondition:-postcondition] operation(parameters)
```

The Triangle example is defined by using Fresco in Figure 3.9. In this example, move, rotate and area are declared with op(operation)s, meaning the states are changed. In case the state is not changed, fn(function) is used. The variables and invariants appear in the private partition, since they are not used as part of the interface but are used internally in the model. To specify inheritance, the superclass of Triangle is denoted as Triangle in the Equilateral_Triangle subclass.

```
Triangle
  op move ∈ (Vector)
  op rotate ∈ (Angle)

mv-def: \[ \text{v} \in \text{Vector} \implies \text{position} = \text{position} + \text{v} \] move(\text{v})

rt-def: \[ \text{w} \in \text{Angle} \implies \text{v1} = \text{v1} \odot \theta \land \text{v2} = \text{v2} \odot \theta \land \text{v3} = \text{v3} \odot \theta \] rotate(\text{w})

inv: \[ \text{v1} + \text{v2} + \text{v3} = 0 \land \]
   \[ (\text{p1} = \text{position} \land \text{p2} = \text{p1} + \text{v1} \land \text{p3} = \text{p2} + \text{v2}) \]

Equilateral_Triangle:: Triangle
  op area: ∈ (Real)

a-def: \[ \text{s} \in \text{Real} \implies s = \frac{\sqrt{3}}{4} \text{|v1|^2} \] area(\text{s})
```

Figure 3.9: Fresco specification of the triangle manipulation system

3.4.2 Object-Oriented VDM

Object-Oriented VDM [LS 93] has two types of modules: class modules for incremental inheritance and type modules for subtyping inheritance. Class modules define the internal states and methods. Type modules, which have no state, denote the domain of values. The general form of a class in Object-Oriented VDM is shown in Figure 3.10. A class module begins with the keyword class and ends at the keyword endclass. The class name
is an identification of the class which can have parameters. A subclass is a class whose attributes and methods are inherited from one or more existing classes. The constant construct defines constants in the class. The State schema construct defines the state variables. The initial state construct defines the state when the object is instantiated. The method construct defines the class behavior and consists of zero or more operator schemas. In operator schemas, pre- and post-conditions are used to specify the behavior. A type module consists of type name, supertype, value, and axiom declaration. The attributes of supertype are inherited to the subtype. The value construct specifies the constants and the functions in the type. It gives the name of value and its type. The axiom construct describes the properties of value names.

\[
\text{class class name[parameters]}
\]
\[
\text{inherited class}
\]
\[
\text{constant}
\]
\[
\text{state schema}
\]
\[
\text{initial state}
\]
\[
\text{method}
\]
\[
\text{operator schema*(input, output)}
\]
\[
\text{pre-}
\]
\[
\text{post-}
\]
\[
\text{endclass}
\]

\[
\text{[Class Module]}
\]
\[
\text{type typename}
\]
\[
\text{type declaration}
\]
\[
\text{supertype}
\]
\[
\text{value}
\]
\[
\text{value declaration}
\]
\[
\text{axiom}
\]
\[
\text{axiom declaration}
\]
\[
\text{endtype}
\]

\[
\text{[Type Module]}
\]

Figure 3.10: OOVDM Notation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 3.11 shows how the triangle manipulation system is specified in $\texttt{QVDM}$. In an $\texttt{QVDM}$ specification, attributes are declared in the state schema, and operations such as move, rotate and area are specified in the method clause. To specify inheritance, the inherited class Triangle clause is used in the EquilateralTriangle class. In $\texttt{QVDM}$, there is no way to specify the class invariant. This limitation is one of the disadvantages of $\texttt{QVDM}$.

```plaintext
class Triangle
  state schema
    v1,v2,v3 : Vector
    position : Vector
    p1,p2,p3 : Point
  method
    move(v:Vector)
      pre-
      post-position=position + v
    rotate( $\theta$ :angle)
      pre-
      post-v1 = v1 $\odot \theta$ \land v2 = v2 $\odot \theta$ \land v3 = v3 $\odot \theta$
endclass

class Equilateral_Triangle
  inherited class Triangle
  method
    area(s :real)
      pre-
      post- s = $\sqrt{3} |v_1|^2$
endclass
```

Figure 3.11: $\texttt{QVDM}$ specification of the triangle manipulation system

3.4.3 VDM++

VDM++ [DV 92][DP 95][Lan 94][Lan95] extends VDM with object-oriented facilities and other facilities for concurrency, and real-time processing. VDM++ has been

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
developed as part of the ESPRIT project Afrodite. The overall syntactic structure of VDM++ is:

```
CLASS class-identifier
  Optional-inheritance-clause
  type-definition-part
  value-definition-part
  function-definition-part
  Controlled-inheritance-part
  Instance-variable-part
    invariant-clause
    initialization-clause
  Method-part
  Optional-trace-part
End class-identifier ;
```

Figure 3.12: Structure of VDM++ specification

There are two kinds of inheritance in VDM++: representational inheritance and controlled inheritance. Representational inheritance is the inheritance in which a subclass inherits all data types, variables, methods, invariants and initializations from a superclass. Syntactically, we denote representational inheritance by indicating the superclass in the declaration of the class: IS SUBCLASS OF identifier. Controlled inheritance is the inheritance that a subclass restricts the set of inherited methods from the superclass. The syntax for the controlled inheritance within a class definition is:

```
classified-inheritance- part ::= INHERIT [inheritance-list] +
  | ALLSUPER;
inheritance-list ::= FROM Classname \ ' :: ' method-name-list;
classname ::= SUPER | Class-Id
method-name-list ::= ALL | [method-name] +;
```

VDM++ has optional trace parts. The purpose of the 'trace part' is to restrict the dynamic behavior of an object. Trace parts define allowed sequences of invocation of methods of a class. The whole trace synchronization structure is given below. In the
notation, subtracestr is a trace structure for a subsystem and general tracestr is for a whole system.

\[
\text{trace-synchronization} = \{\text{subtracestr}, \{\text{subtracestr}\}\}, \text{general tracestr}
\]

\[
\text{trace-str} = \langle '<', \text{trace set}, ',', ',', \text{alphabet}, '>' \rangle
\]

For example, subtrace \( tr_1 = \langle \langle \text{init}; (\text{first}; \text{next}^*) \rangle, \{\text{init, first, next}\} \rangle \) where ' ; ' denotes sequential composition and ' * ' denotes repetition zero or more. Figure 3.12 shows a triangle manipulation system using VDM++.

```
Class Triangle

Instance variables
v1,v2,v3 : Vector
position : Vector
p1,p2,p3 : Point
inv v1,v2,v3,p1,p2,p3 == [v1+v2+v3=0 ∧ p1=position∧p2=p1 + v1 ∧ p3=p2 + v2 ]

methods
move(v:Vector)
pre
post position = position + v
rotate( θ : Angle)
pre
post v1 = v1 ⊕ θ ∧ v2 = v2 ⊕ θ ∧ v3 = v3 ⊕ θ

End Triangle;

Class Equilateral_Triangle is
subclass of Triangle;
methods
area(s:real)
pre
post s = \( \sqrt{3}/4 \) |v1|^2

End Equilateral_Triangle;
```

Figure 3.13: VDM++ specification of the triangle manipulation system
In the example, attributes in a class are declared in the instance variables clause. Operations such as move, rotate and area are described using the keyword method. For inheritance, is subclass of Triangle is used in the EquilateralTriangle subclass. Since the Triangle example does not need synchronization constraints, the trace structure is not presented.

3.4.4 Comparison of existing languages.

Fresco, ODVDM, and VDM++ have the basic object-oriented facilities such as class structure, inheritance, and polymorphism. However, to use the object-oriented paradigm effectively, the following additional facilities are desirable: type structure, class invariant, class initialization, class constants, visibility, and parameterized classes [SBC 92].

<table>
<thead>
<tr>
<th></th>
<th>Fresco</th>
<th>ODVDM</th>
<th>VDM++</th>
<th>Object-VDM</th>
</tr>
</thead>
<tbody>
<tr>
<td>class hierarchy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>type hierarchy</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>class invariant</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>state initialization</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>constant</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>based on VDM standard</td>
<td>No</td>
<td>No</td>
<td>Yes(partial)</td>
<td>Yes</td>
</tr>
<tr>
<td>visibility</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>parameterized class</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>concurrency</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 3.14: Comparison of object-oriented VDMs

Type structure is needed to specify subtyping inheritance. Class invariant is a predicate that describes the properties of the internal state. This predicate must be held before and after execution of each operation in the class. Class initialization declares the valid initial state of a newly-created instance of the class. The class constants are fixed values which cannot be changed by any method and are the same.
for all instances of the class. Visibility restricts the access to the listed feature of objects of the class. Parameterized classes or genericity enable us to avoid the need to write many almost identical classes for different types.

Fresco has two hierarchies: the 'class hierarchy' and 'type hierarchy'. Fresco has facilities for visible operations and class invariants; however, Fresco lacks the facilities for constant, initialization, and generic parameters in the class. QVDM also has two hierarchies. QVDM has facilities for constant and initialization, but it lacks the facilities for the visibility of operations and class invariant. VDM++ has two inheritance mechanisms: representational inheritance and controlled inheritance. VDM++ also has trace facilities to specify the restrictions in dynamics behavior. VDM++ has facilities for class invariant, class initialization, and class constant, but lacks the facilities for visibility lists, parameterized class, and type hierarchy.

All existing objected extensions to VDM are not fully based on VDM standard. Fresco's overall structure varies from that of the VDM standard. Its notation also does not follow the VDM standard. QVDM does not distinguish functions and operations, and its keywords are different from those of the VDM standard. Current VDM++ adopts the notation and structures of the VDM standard; however, some keywords are different from those of the VDM standard. For example, the keyword 'instance variables' in VDM++ is used for 'state' in the VDM standard. VDM++ also does not support some types (e.g. function type, type variables), expressions (e.g. iota, lambda), and statements (e.g. nondeterministic, exit, trap) in the VDM standard [DP 95]. The comparisons of the existing object-oriented VDM extensions and Object-VDM are summarized in Figure 3.14.

3.5 SUMMARY

Fresco, QVDM, and VDM++ are well-known object-oriented extensions of VDM. These existing object-oriented extensions of VDM have two limitations. One limita-
tion is that they lack complete facilities for the object-orientation paradigm. Another is that they are not fully based on the current VDM standard. We presented an object-oriented specification language which addresses these limitations. The new object-oriented VDM, Object-VDM, has the following facilities: class hierarchy and type hierarchy, class invariants, state schema, initialization, parameterized class, and visibility of operations.
CHAPTER 4
REFINEMENT IN OBJECT-VDM

To refine Object-VDM to Eiffel code, we used an adapted form of a refinement method in VDM [Jon 90]. A refinement method in VDM was briefly described in Section 2.3. We modified and extended refinement methods in VDM. The original refinement method in VDM has two steps: data refinement and operation refinement. To convert object-oriented facilities, we add structure refinement steps. In data refinement, the mathematical data models in object-VDM such as SET, SEQUENCE, and MAP are converted to Eiffel data structures. We constructed Eiffel libraries to do this. We also proved the correctness of the conversion. In operation refinement, we modify and add rules to the original operational refinement in VDM to obtain Eiffel code. Specifically we add rules to handle quantified predicates. Section 4.1 presents data refinement. In Section 4.2, we explain operation refinement. Section 4.3 describes structure refinement.

4.1 DATA REFINEMENT

4.1.1 INTRODUCTION

Data refinement from the abstract VDM state to the concrete VDM state is described in [Jon 90], and we briefly reviewed it in the Section 2.3.1. We examine the refinement from the Object-VDM state to an Eiffel state. In Section 4.1.2 and 4.1.3, we investigate types and typical operations in Object-VDM and Eiffel. Section 4.1.4 gives the Object-VDM and Section 4.1.5 gives Eiffel assertions for SET, SEQUENCE, and MAP type. Proof obligations are presented in Section 4.1.6. Finally, other considerations for data refinement are described in Section 4.1.7.
4.1.2 Types in Object-VDM and Eiffel

A type in (Object-)VDM can be classified as either a basic type or a compound type.[Daw 91][VDM 93] A basic type is divided by a numeric type (the positive natural-numbers, all natural numbers, integers, the rationals, and the reals), the Boolean type, characters, tokens or a unit type (a singleton set containing NIL as its only element). A compound type is divided by a set type, a sequence type, a mapping type, a composite type, a union type, a function type, a type identifier, an optional type, a type variable, a product type, or a quotation type.

Among these types, we will investigate three types: set type, sequence type, and mapping type, since these are three fundamental data structures in Object-VDM.

SetType = elemtp -set;
SetType is a set type. It consists of a type, elemtp. It is understood as the set of all finite subsets of type of the elemtp.

SeqType = Seq0Type | Seq1Type
SeqType is a sequence type. It is either a possibly empty sequence type or a non-empty sequence type.

Seq0Type = elemtp, "*";
Seq0Type is a possibly empty sequence type. It consists of a type, elemtp. It is understood as the set of all finite sequences with elemtp.

Seq1Type = elemtp, '+'
Seq1Type is a non-empty sequence type. It consists of a type, elemtp. It is understood as the set of all non-empty finite sequences with elemtp.

MapType = GeneralMapType | InjectiveMapType
MapType is a map type. It is either a general map type or a injective map type.

GeneralMapType = dom \rightarrow^n rng
GeneralMapType is a general map type. It consists of a domain type, dom
and a range type, rng. It is understood as the set of all finite mappings from the dom type to the rng type.

InjectiveMap Type:: dom $\xrightarrow{m}$ rng

InjectiveMapType is a injective map type. It consists of a domain type, dom, and a range type, rng. It is understood as the set of all finite injective mappings from the dom type to the rng type.

There are two kinds of types in Eiffel: reference types and expanded types. In a reference type, the possible values are references to potential objects which may be created at run-time. In an expanded type, the possible values are the objects themselves. Expanded types include the basic types: INTEGER, REAL, CHARACTER, BOOLEAN, DOUBLE. Eiffel also supports types such as the array and the linked list through Eiffel libraries. Arrays and linked lists are the data structures which can implement the set type, sequence type, and mapping type in Object-VDM (or VDM). We choose linked lists for set type, sequence type, and mapping types.

4.1.3 TYPICAL OPERATIONS IN SET, SEQUENCE, AND MAP

SETS

Set operations can be classified into three categories: operations to query the state, operations to modify the set. Each category contains the following set operations.

- operations which do not change the state
  

- operations which change the state
  
  AddElmt, SubtractElmt, Intersect, Union, Difference, DistUnion(Distributed Union), DistIntersect(Distributed Intersection)

We will briefly explain each operation.
IsEqual— which, given two sets, returns true if two sets are the same and false otherwise.

IsNotEqual— which, given two sets, returns true if two sets are not the same and false otherwise.

Cardinality— which, given a set, returns the number of elements

IsElmt— which, given a set and value, returns true if the value is an element of the set and false otherwise.

IsNotElmt— which, given a set and value, returns true if the value is not an element of the set and false otherwise.

IsDisjoint— which, given two sets, returns true if two sets do not have any common element and false otherwise.

IsSubset— which, given two sets, returns true if one set includes another set and false otherwise.

IsSubsetProper— which, given two sets, returns true if one set includes another set and two sets are not the same and false otherwise.

IsEmpty— which, given a set, returns true if the set is empty and false otherwise.

AddElmt— which, given a set and an element, returns the set that adds the element to the set

SubtractElmt— which, given a set and an element, returns the set that subtracts the element to the set

Intersect— which, given two sets, returns the set that is their set intersect

Union— which, given two sets, returns the set that is their set union

Difference— which, given two sets, returns the set that is their set difference

DistUnion— which, given a set of sets, returns the set that is their set union

DistIntersect— which, given a set of sets, returns the set that is their set intersection
e.g. Let $S=\{\{1\}, \{1,3\}, \{1,2,3,4\}\}$.

Then DistUnion of $S=\{1,2,3,4\}$ and DistIntersect of $S=\{1\}$

**SEQUENCES**

A sequence is an ordered set of elements. Sequence operations can be classified as follows. concatenates all sequences together

- operations which do not change the state
  
  Head, Tail, Length, Elements, Indices

- operations which change the state
  
  Conc

We will briefly explain each operation.

Head—which, given a sequence, returns the first element of the sequence

Tail—which, given a sequence, returns the sequence with its head removed

Length—which, given a sequence, returns the length of the sequence

Elements—which, given a sequence, returns the set of sequence elements

Indices—which, given a sequence, returns the set of indices

Conc—which, given two sequences, links them together.

**MAPS**

A map is a collection of ordered pairs. Map operations can be classified as follows. concatenates all sequences together

- operations which do not change the state
  
  domain, range, IsEqual, IsNotEqual

- operations which change the state
  
  Inverse, Merge, Composite, DomRestr, DomExcl, RanRestr, RanExcl, Override

We will briefly explain each operation.
Domain—which, given a map, will return the set of elements comprising the domain of the map
Range—which, given a map, will return the set of elements comprising the range of the map
Inverse—which, given a map, will return the map whose domain and range are changed.
Composite—which, given two maps, will return the map that compose two maps.
Merge—which, given two maps, will return the map that has elements of the first map.
Override—which, given two maps, will return the map whose elements are those of the first map except the common elements which is overridden by the second map.
DomRestr—which, given a set and a map, will return the map whose domain is restricted to the elements of the set
DomExcl—which, given a set and a map, will return the map whose domain is restricted by excluding the elements of the set
RanRestr—which, given a set and a map, will return the map whose range is restricted to the elements of the set
RanExcl—which, given a set and a map, will return the map whose range is restricted by excluding the elements of the set
IsEqual—which, given two maps, will return true if two maps have the same domain and range and false otherwise.
IsNotEqual—which, given two maps, will return true if two maps do not have the same domain and range and false otherwise.

4.1.4 OBJECT-VDM SPECIFICATIONS FOR THE SET, SEQUENCE, AND MAP TYPE

SETS

The mathematical notation for set operation is as follows:
∈(membership), ∉(non membership), ⊆(subset), ⊂(proper subset), 
∩ (union), ∪(intersection), \(\setminus\)(difference), \(\cap\)(distributed set intersection), \(\cup\)(distributed set union), card(cardinality)

To convert these operations into Eiffel, we have to make each operation into a VDM operation which has both a pre- and post-condition. Figure 4.1 shows the Object-VDM specification for the class SET and its operations.

SEQUENCES

The mathematical syntax notation for unary sequence operation is as follows:
hd(sequence head), tl(sequence tail), len(sequence length),
elms(sequence elements), inds(sequence indices),
~(sequence concatenate)

Figure 4.2 shows the Object-VDM specification for the class SEQUENCE and its operations.

MAPS

The mathematical syntax notation for unary map operation is as follows:
dom(map domain), rng(map range), expression ‘\(^{-1}\)’(map inverse expression),
\(\langle\)(map domain restriction), \(\langle\)(map domain exclusion),
\(\rangle\)(map range restriction), \(\rangle\)(map range exclusion),
o(map composition), ↑(map iteration),
t(map modify), \(\cup\)(map merge),
=(map equality), ≠(map inequality)

Figure 4.3 shows the Object-VDM specification for the class MAP and its operations.
Class SET(t)
    state Set_Class of
        S:T -set
        SS:T -set -set
    end
    functions
        isEmpty():b:B
            ext rd S
            pre
            post (S={} ∧ b=true) ∨ (S≠{} ∧ b=false)
        IsSubsetProper(S1):b:B
            ext rd S
            pre
            post (S⊂S1 ∧ b=true) ∨ (S⊄S1 ∧ b=false)
        isEqual(S1):b:B
            ext rd S
            pre
            post (S=S1 ∧ b=true) ∨ (S≠S1 ∧ b=false)
        addElmt(e):R:SET(T)
            ext rd S
            pre
            post R=S ∪ {e}
        subtractElmt(e):R:SET(T)
            ext rd S
            pre
            post R=S-{e}
        cardinality():n:N
            ext rd S
            pre
            post n=card(S)
        intersect(S1):R:SET(T)
            ext rd S
            pre
            post R=S ∩ S1
        union(S1):R:SET(T)
            ext rd S
            pre
            post R=S ∪ S1
        difference(S1):R:SET(T)
            ext rd S
            pre
            post R=S \ S1
        isDisjoint(S1):b:B
            ext rd S
            pre
            post (S∩S1={} ∧ b=true) ∨ (S∩S1≠{} ∧ b=false)
            post R=∪ S
        isSubset(S1):b:B
            ext rd S
            pre
            post (S⊆ S1 ∧ b=true) ∨ (S⊄ S1 ∧ b=false)
            post R=∩ S
    End class

Figure 4.1: Object-VDM code for a class SET.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Class SEQUENCE(T)
state Sequence_Class of
  Q:T*
end
functions
  Head():R:T
  ext rd Q
  pre
  post R=hd Q

  Tail():R:SEQUENCE(T)
  ext rd Q
  pre
  post r=tl Q

  Length():R:N
  ext rd Q
  pre
  post R=len Q

  Elements():R:SET(T)
  ext rd Q
  pre
  post R=elms Q

  Indices():R:Set(N)
  ext rd S
  pre
  post R=inds Q

  conc(Q1):R:SEQUENCE(T)
  ext rd Q
  pre
  post R=Q \ Q1
End Class

Figure 4.2: Object-VDM code for a class SEQUENCE.
Class MAP(T1,T2)
  state Map_Class of
    M:T1 \rightarrow T2
  end
  functions
    Domain():R:SET(T1)
      ext rd M
      pre
      post r=dom M
    Range():R:SET(T2)
      ext rd M
      pre
      post R=ran M
    Inverse():R:MAP(T1,T2)
      ext rd M
      pre
      post R=M \neg '1'
    DomRestr(S:SET(T1)):R:MAP(T1,T2)
      ext rd M
      pre
      post R=S \chi M
    DomExcl(S:SET(T1)):R:MAP(T1,T2)
      ext rd M
      pre
      post R=S \downarrow M
    RanRestr(S:SET(T2)):R:MAP(T1,T2)
      ext rd M
      pre
      post R=M \triangleright S
    DomExcl(S:SET(T2)):R:MAP(T1,T2)
      ext rd M
      pre
      post R=M \triangleright S

Figure 4.3: Object-VDM code for a class MAP. (Figure continued)
Override(M1:MAP(T1,T2))R:MAP(T1,T2)
expre
post R=M M1

Composition(M1:MAP(T3,T1))R:MAP(T3,T2)
expre
post R=M M1

Merge(M1:MAP(T1,T2))R:MAP(T1,T2)
expre
post R=M M1

IsEqual(M1:MAP(T1,T2))b:B
expre
post (M=M1 b=true) (M¹M1 b=false)

IsNotEqual(M1:MAP(T1,T2))b:B
expre
post (M¹M1 b=true) (M=M1 b=false)

End Class
4.1.5 **Eiffel Interfaces for Sets, Sequences, and Maps**

In Eiffel [Mey 88], quantifiers cannot be described in the assertion expression. Since many pre and post-conditions need the quantifier for the expression, we introduce syntax to deal with the quantified expressions. These expressions are not included in the regular Eiffel syntax, therefore they are described using comments. The syntax for a quantified expression is:

\[
\text{quantified expression} = \text{quantifier} \ \text{identifier} : \ \text{range}, \ \text{expression}
\]

\[
\text{quantifier} = \text{forall} \ | \ \text{exists} \ | \ \text{not exists}
\]

\[
\text{range} = \text{lower..upper}
\]

\[
\text{lower} = \text{identifier} \ | \ \text{integer} \ | \ \text{constant}
\]

\[
\text{upper} = \text{identifier} \ | \ \text{integer} \ | \ \text{constant}
\]

Here is an example:

\[-\text{forall i: 1..count, Result.i.th(i)=other.i.th(i)}\]

**Sets**

The following shows the Eiffel interface along with the equivalent Object-VDM specification for the Class SET and its operations.
<table>
<thead>
<tr>
<th><strong>Object-VDM</strong></th>
<th><strong>Eiffel</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class SET(T)</strong></td>
<td><strong>Class interface Set[T]</strong></td>
</tr>
<tr>
<td>state Set.Class of</td>
<td>export features</td>
</tr>
<tr>
<td>S:T-set</td>
<td>Init, IsEqual, IsNotEqual,</td>
</tr>
<tr>
<td>SS:T-set-set</td>
<td>Cardinality, IsElmt, IsNotElmt,</td>
</tr>
<tr>
<td></td>
<td>IsDisjoint, IsSubset, IsSubsetProper,</td>
</tr>
<tr>
<td></td>
<td>AddElmt, SubtractElmt,</td>
</tr>
<tr>
<td></td>
<td>Intersect, Union, Difference</td>
</tr>
<tr>
<td></td>
<td>inherit linked_list[T]</td>
</tr>
<tr>
<td><strong>functions</strong></td>
<td><strong>feature specification</strong></td>
</tr>
<tr>
<td>IsEmpty()b:B</td>
<td>IsEmpty</td>
</tr>
<tr>
<td>ext rd S</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post (S={}∧b=true)∨(S≠{}∧b=false)</td>
<td>count=0</td>
</tr>
<tr>
<td>IsEqual(S1)b:B</td>
<td>IsEqual(other):BOOLEAN</td>
</tr>
<tr>
<td>ext rd S</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post (S=S1∧b=true)∨(S≠S1∧b=false)</td>
<td>count=other.count</td>
</tr>
<tr>
<td>- exists i: 1..count,</td>
<td>- exists i: 1..count,</td>
</tr>
<tr>
<td>- forall j: 1..count,</td>
<td>- forall j: 1..other.count,</td>
</tr>
<tr>
<td>- i.th(i)/=other.i.th(j)</td>
<td>- i.th(i)/=other.i.th(j)</td>
</tr>
<tr>
<td>IsNotEqual(S1)b:B</td>
<td>IsNotEqual(other):BOOLEAN</td>
</tr>
<tr>
<td>ext rd S</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post (S ≠ S1 ∧ b=true)∨(S=S1∧b=false)</td>
<td>- exists i: 1..count,</td>
</tr>
<tr>
<td></td>
<td>- forall j: 1..other.count,</td>
</tr>
<tr>
<td></td>
<td>- i.th(i)/=other.i.th(j)</td>
</tr>
<tr>
<td>Cardinality():n:N</td>
<td>Cardinality:INTEGER</td>
</tr>
<tr>
<td>ext rd S</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post n=card(S)</td>
<td>Result=count</td>
</tr>
</tbody>
</table>

Figure 4.4: Comparison of Eiffel code with Object-VDM code for a class SET. (Figure Continued)
IsElmt(e:b): BOOLEAN
ext rd S
pre
post (e ∈ S1 ∧ b = true) ∨
       (e ∉ S1 ∧ b = false)
  - exists i: 1..count,
  - i.th(i) = e

IsNotElmt(e:T): BOOLEAN
ext rd S
pre
post (e ∉ S1 ∧ b = true) ∨
       (e ∈ S1 ∧ b = false)
  - not exists i: 1..count,
  - i.th(i) = e

IsDisjoint(S1): BOOLEAN
ext rd S
pre
post (S ∩ S1 = {} ∧ b = true) ∨
       (S ∩ S1 ≠ {} ∧ b = false)
  - forall i: 1..count,
  - forall j: 1..other.count,
  - i.th(i) ≠ other.i.th(j)

IsSubset(S1): BOOLEAN
ext rd S
pre
post (S ⊆ S1 ∧ b = true) ∨
       (S ⊋ S1 ∧ b = false)
  - forall i: 1..count,
  - exists j: 1..other.count,
  - i.th(i) = other.i.th(j)

IsSubsetProper(S1): BOOLEAN
ext rd S
pre
post (S ⊆ S1 ∧ b = true) ∨
       (S ⊋ S1 ∧ b = false)
  - forall i: 1..count,
  - exists j: 1..other.count,
  - i.th(i) = other.i.th(j)
  - and
  - exists j: 1..other.count,
  - forall i: 1..count,
  - i.th(i) ≠ other.i.th(j)

AddElmt(e:T): like Current
ext rd S
pre
post R = S ∪ {e}
  - count = old count + 1
  - forall i: 1..old count
  - exists j: 1..count,
  - i.th(j) = old.i.th(i)
  - and
  - exists j: 1..count,
  - i.th(j) = e

(Figure Continued)
<table>
<thead>
<tr>
<th>Class</th>
<th>Extends</th>
<th>Pre-condition</th>
<th>Post-condition</th>
<th>Ensure</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubtractElmt(e:T:R)</td>
<td>SET(T)</td>
<td>require</td>
<td>R=S-{e}</td>
<td>count=old count-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ensure</td>
<td></td>
<td>- forall j: lower&lt;=j; j&lt;=upper,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- i.th(j)==e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- forall i: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- exists j: 1..old count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- i.th(i)==old i.th(j)</td>
</tr>
<tr>
<td>Intersect(S1:R)</td>
<td>SET(T)</td>
<td>require</td>
<td>R=S\S1</td>
<td>- forall i: 1..Result.count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ensure</td>
<td></td>
<td>- (exists j: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Result.i.th(i)==i.th(j)) and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- (exists k: 1..other.count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Result.i.th(i)==other.i.th(k))</td>
</tr>
<tr>
<td>Union(S1:R)</td>
<td>SET(T)</td>
<td>require</td>
<td>R=S\S1</td>
<td>- forall i: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ensure</td>
<td></td>
<td>- (exists j: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Result.i.th(i)==i.th(j)) or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- (exists k: 1..other.count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Result.i.th(i)==other.i.th(k))</td>
</tr>
<tr>
<td>Difference(S1:R)</td>
<td>SET(T)</td>
<td>require</td>
<td>R=S\S1</td>
<td>- forall i: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ensure</td>
<td></td>
<td>- (exists j: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Result.i.th(i)==i.th(j))</td>
</tr>
<tr>
<td>DistUnion(R:R)</td>
<td>SET(T)</td>
<td>require</td>
<td>R=U SS</td>
<td>DistUnion: SET[T]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ensure</td>
<td></td>
<td>- forall i: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- forall j: 1..i.th(i).count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- exists k: 1..Result.count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Result.i.th(k)==(i.th(i)).i.th(j)</td>
</tr>
<tr>
<td>DistIntersect(R:R)</td>
<td>SET(T)</td>
<td>require</td>
<td>R=\ SS</td>
<td>DistIntersect: SET[T]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ensure</td>
<td></td>
<td>- forall k: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- forall i: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- exists j: 1..count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Result.i.th(k)==i.th(i).i.th(j)</td>
</tr>
</tbody>
</table>

endclass - SET(T)    | end - Class Set[T] |
SEQUENCES

The following shows the Eiffel interface along with the equivalent Object-VDM specification for the Class SEQUENCE and its operations.

<table>
<thead>
<tr>
<th>Object-VDM</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class SEQUENCE(T)</strong></td>
<td><strong>Class interface SEQUENCE[T]</strong></td>
</tr>
<tr>
<td>state Sequence_Class of</td>
<td>export features</td>
</tr>
<tr>
<td>Q:T-sequence</td>
<td>head,tail,length,elements,indices,conc</td>
</tr>
<tr>
<td>end</td>
<td>inherit linked_list[T]</td>
</tr>
<tr>
<td><strong>functions</strong></td>
<td><strong>feature specification</strong></td>
</tr>
<tr>
<td>Head() r:T</td>
<td>Head:T</td>
</tr>
<tr>
<td>ext rd Q</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post r=hd Q</td>
<td>Result=i.th(1)</td>
</tr>
<tr>
<td>Tail() r:SEQUENCE(T)</td>
<td>Tail:T</td>
</tr>
<tr>
<td>ext rd Q</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post r=tl Q</td>
<td>Result=i.th(count)</td>
</tr>
<tr>
<td>Length() r:N</td>
<td>Length:INTEGER</td>
</tr>
<tr>
<td>ext rd Q</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post r=len Q</td>
<td>Result=count</td>
</tr>
<tr>
<td>Elements() r:SET(T)</td>
<td>Elements:SET[T]</td>
</tr>
<tr>
<td>ext rd Q</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post r=elms Q</td>
<td>- for all i: 1..count,</td>
</tr>
<tr>
<td></td>
<td>- Result.i.th(i)=i.th(i)</td>
</tr>
</tbody>
</table>

Figure 4.5: Comparison of Eiffel code with Object-VDM code for a class SEQUENCE. (Figure Continued)
Indices():SET(T)

ext rd Q
pre
post r=inds Q

Indices:SET[T]

Concern(Q1):SEQUENCE(T)

ext rd Q
pre
post r=Q ~ Q1

Conc(Q1):SEQUENCE(T)

Conc(other: like Current) like Current

require
ensure
forall i: 1..count,
- Result.i.th(i)=i

forall j: 1..other.count,
- Result.i.th(count+j)=other.i.th(j)

endclass -SEQUENCE(T)
end - Class SEQUENCE[T]

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The following shows the Eiffel interface along with the equivalent Object-VDM specification for the Class MAP and its operations.

**Object-VDM**

<table>
<thead>
<tr>
<th>Class MAP(T1,T2)</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>state Map_class of</td>
<td>Class interface MAP[T1,T2]</td>
</tr>
<tr>
<td>M:T1 $\rightarrow^n$ T2</td>
<td>export features</td>
</tr>
<tr>
<td></td>
<td>domain, range, IsEqual, IsNotEqual,</td>
</tr>
<tr>
<td></td>
<td>Inverse, Merge, Composite, DomRestr, DomExcl,</td>
</tr>
<tr>
<td></td>
<td>RanRestr, RanExcl, Override</td>
</tr>
<tr>
<td></td>
<td>inherit linked_list[PAIR[T1,T2]]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>functions</th>
<th>feature specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain():SET(T1)</td>
<td>Domain():SET[T1]</td>
</tr>
<tr>
<td>ext rd M</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post r=dom M</td>
<td>- forall i: 1..count,</td>
</tr>
<tr>
<td></td>
<td>- Result.i.th(i)=i.th(i).first</td>
</tr>
<tr>
<td>Range():SET(T2)</td>
<td>Range():SET[T2]</td>
</tr>
<tr>
<td>ext rd M</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post r=ran M</td>
<td>- forall i: 1..count,</td>
</tr>
<tr>
<td></td>
<td>- Result.i.th(i).first=i.th(i+1).first</td>
</tr>
<tr>
<td></td>
<td>- Result.i.th(i).first=i.th(i+1).second</td>
</tr>
<tr>
<td>Inverse():MAP(T2,T1)</td>
<td>Inverse():MAP[T2,T1]</td>
</tr>
<tr>
<td>ext rd M</td>
<td>require</td>
</tr>
<tr>
<td>pre</td>
<td>ensure</td>
</tr>
<tr>
<td>post R=M '¬1'</td>
<td>- forall i: 1..count,</td>
</tr>
<tr>
<td></td>
<td>- Result.i.th(i).second=i.th(i).first</td>
</tr>
</tbody>
</table>

Figure 4.6: Comparison of Eiffel code with Object-VDM code for a class MAP. (Figure Continued)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Precondition</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomRestr(S:SET(T1)) R:MAP(T1, T2)</td>
<td>Domain Restriction</td>
<td></td>
<td>$R = S \triangle M$</td>
</tr>
<tr>
<td>DomExcl(S:SET(T1)) R:MAP(T1, T2)</td>
<td>Domain Exclusion</td>
<td></td>
<td>$R = S \setminus M$</td>
</tr>
<tr>
<td>RanRestr(S:SET) R:MAP(T1, T2)</td>
<td>Range Restriction</td>
<td></td>
<td>$R = S \times M$</td>
</tr>
<tr>
<td>RanExcl(S:SET(T1)) R:MAP(T1, T2)</td>
<td>Range Exclusion</td>
<td></td>
<td>$R = S \setminus M$</td>
</tr>
</tbody>
</table>

**DomRestr(S:SET(T1)) R:MAP(T1, T2)**
- **require**
  - For all $i$: $1..S.count$,
  - $\exists j$: $1..Result.count$,
  - $\exists k$: $1..count$,
  - $i.th(k).first = S.i.th(i)$
  - $Result.i.th(j).first = S.i.th(i)$
  - $Result.i.th(j).second = i.th(k).second$

**DomExcl(S:SET(T1)) R:MAP(T1, T2)**
- **require**
  - For all $i$: $1..S.count$,
  - For all $j$: $1..Result.count$,
  - $Result.i.th(j).first = i.th(i).first$
  - And
  - For all $i$: $1..S.count$,
  - $\exists j$: $1..Result.count$,
  - $\exists k$: $1..count$,
  - $Result.i.th(j).second = i.th(k).second$
  - $i.th(k).first = S.i.th(i)$

**RanRestr(S:SET) R:MAP(T1, T2)**
- **require**
  - For all $i$: $1..S.count$,
  - $\exists j$: $1..Result.count$,
  - $\exists k$: $1..count$,
  - $i.th(k).second = S.i.th(i)$
  - $Result.i.th(j).second = S.i.th(i)$
  - $Result.i.th(j).first = i.th(k).first$

**RanExcl(S:SET(T1)) R:MAP(T1, T2)**
- **require**
  - For all $i$: $1..S.count$,
  - $\exists j$: $1..Result.count$,
  - $\exists k$: $1..count$,
  - $Result.i.th(j).second = i.th(i).second$
  - $Result.i.th(j).first = i.th(k).first$
Override(M1:MAP(T1,T2))R:MAP(T1,T2)  
\text{ext rd M} \\
\text{pre} \\
\text{post } R=M \uparrow M1 \\
\text{require} \\
\text{ensure} \\
\begin{align*} 
- \text{forall } i: & 1..S.count, \\
- \text{exists } j: & 1..\text{Result.count}, \\
- \text{Result.i_th(j).second}/=S.i_th(i) \\
\textbf{and} \\
- \text{forall } i: & 1..S.count, \\
- \text{exists } j: & 1..\text{count}, \\
- \text{Result.i_th(j).first }=i_th(i).first \\
- \text{Result.i_th(j).second }=i_th(i).second \\
\end{align*}

Composition(M1:MAP(T1,T2))R:MAP(T1,T2)  
\text{ext rd Q} \\
\text{pre} \\
\text{post } R=M \circ M1 \\
\text{require} \\
\text{ensure} \\
\begin{align*} 
- \text{forall } i: & 1..S.count, \\
- \text{exists } j: & 1..\text{Result.count}, \\
- \text{Result.i_th(j).second}/=S.i_th(i) \\
\textbf{and} \\
- \text{forall } i: & 1..S.count, \\
- \text{exists } j: & 1..\text{count}, \\
- \text{Result.i_th(j).first }=i_th(i).first \\
- \text{Result.i_th(j).second }=i_th(i).second \\
\end{align*}

Merge(M1:MAP(T1,T2))R:MAP(T1,T2)  
\text{ext rd M} \\
\text{pre} \\
\text{post } R=M \cup M1 \\
\text{require} \\
\text{ensure} \\
\begin{align*} 
- \text{forall } i: & 1..\text{count}, \\
- \text{exists } j: & 1..\text{M1.count}, \\
- \text{Result}(j).second=M1.i_th(i): \\
- \text{Result}(j).first =i_th(i).first \\
- \text{Result}(j).second=M1.i_th(i).second \\
\end{align*}

IsEqual(M1:MAP(T1,T2))b:B  
\text{ext rd M} \\
\text{pre} \\
\text{post } (M=M1 \land b=true) \lor (M \neq M1 \land b=false) \\
\text{require} \\
\text{ensure} \\
\begin{align*} 
- \text{forall } i: & 1..\text{count}, \\
- \text{exists } j: & 1..\text{Result.count}, \\
- \text{Result}(j).first =i_th(i).first \\
- \text{Result}(j).second=i_th(i).second \\
\end{align*}

IsNotEqual(M1:MAP(T1,T2))b:B  
\text{ext rd M} \\
\text{pre} \\
\text{post } (M \neq M1 \land b=true) \lor (M=M1 \land b=false) \\
\text{require} \\
\text{ensure} \\
\begin{align*} 
- \text{not IsEqual}(M1) \\
\end{align*}

endclass - MAP(T1,T2)
The Eiffel implementation version for SET, SEQUENCE, and MAP class is given in Appendix B.

4.1.6 Proof Obligations

To prove the Eiffel representation is correct, we have to show that the retrieve function from the Eiffel implementation to the Object-VDM is total and adequate. The Eiffel representation of SET, SEQUENCE, and MAP is LINKED LIST. The retrieve function, retr1, for SET is defined as follows: retr1(dr)={dr.i_th(1), dr.i_th(2),..., dr.i_th(count)} where count is the number of items in a LINKED LIST. We show retr1 from LINKED LIST to SET is total and adequate.

Theorem 1 retr1 is total.

PROOF:

Since every element dr in LINKED LIST has dr.i_th(i) where i is between 1 and count, there exists {dr.i_th(1), dr.i_th(2),...,dr.i_th(count)} for every dr. Therefore retr1 is total.

Theorem 2 retr1 is adequate.

PROOF:

This is proved by induction.

from d∈ SET(T)
1. e ∈ LINKED_LIST(T) · e.count=0 empty linked list
2. retr1(e)={}
3. ∃ dr ∈ LINKED_LIST(T) · retr1(dr)={} retr1
4. from d∈ SET(T), w ∈ d
   ∃ dr ∈ LINKED_LIST(T) · retr1(dr)=d
4.1 from dr ∈ LINKED_LIST(T), retr1(dr)=d h4.1, retr1
4.1.1 {dr.i_th(1), dr.i_th(2),..., dr.i_th(count)}=d h4, 4.1.1
4.1.2 w ∈ {dr.i_th(1), dr.i_th(2),..., dr.i_th(count)}
4.1.3 e1 ∈ LINKED_LIST(T) · ∀i: 1 ≤ i ≤ count:
   e1.i_th(i)=dr.i_th(i) ∧ (e1.i_th(count+1)=w)
4.1.4 $\text{retr1}(e1) = d \cup \{ w \} \quad 4.1.1, 4.1.3, \text{retr1}$

$\quad \text{infer } \exists \text{dr1 } \in \text{LINKED\_LIST}(T) \cdot \text{retr1(dr1)} = d \cup \{ w \} \quad \exists-I(4.1.3, 4.1.4)$

$\quad \text{infer } \exists \text{dr1 } \in \text{LINKED\_LIST}(T) \cdot \text{retr1(dr1)} = d \cup \{ w \} \quad \exists-E(h4,4.1)$

$\quad \text{infer } \exists \text{dr } \in \text{LINKED\_LIST}(T) \cdot \text{retr1(dr)} = d \quad h,3,4$

In the above proof, a hypothesis in the induction is denoted $h$ with a numeral such as $h4.1$. We used $\exists-I$ rule and $\exists-E$ rule. These rules are stated as follows:

$\exists-I$ rule:

$s \in X; E(s/x)$

$\exists z \in X \cdot E(x)$

$\exists-E$ rule:

$\exists x \in X \cdot E(x); y \in X, E(y/x) \vdash E1 \quad E1$

The retrieve function $\text{retr2}$ from $\text{LINKED\_LIST}$ to $\text{SEQUENCE}$ is defined as follows:

$\text{retr2(dr)} = [\text{dr.i\_th}(1), \text{dr.i\_th}(2), \ldots, \text{dr.i\_th(count)}]$. Theorem 3 and theorem 4 shows $\text{retr2}$ is total and adequate.

**Theorem 3** $\text{retr2}$ is total.

**PROOF:**

Since every element dr in $\text{LINKED\_LIST}$ has $\text{dr.i\_th(i)}$ where $i$ is between lower and upper, there exists $[\text{dr.i\_th}(1), \text{dr.i\_th}(2), \ldots, \text{dr.i\_th(count)}]$ for every dr. Therefore $\text{retr2}$ is total.

**Theorem 4** $\text{retr2}$ is adequate.

**PROOF:**

This is proved by induction.

from $d \in \text{SEQUENCE}(T)$

1. $e \in \text{LINKED\_LIST}(T) \cdot e.count=0 \quad \text{empty linked list}$

2. $\text{retr2(dr)} = [] \quad \text{retr2}$

3. $\exists \text{dr } \in \text{LINKED\_LIST}(T) \cdot \text{retr2(dr)} = [] \quad \exists-I(1,2)$
4. from \( d \in \text{SEQUENCE}(T) \), \( w \not\in d \)
   \( \exists d_r \in \text{LINKED}_L\text{IST}(T) \cdot \text{retr2}(d_r)=d \)

4.1 from \( d_r \in \text{LINKED}_L\text{IST}(T) \), \( \text{retr2}(d_r)=d \)

4.1.1 \([d_r.i_th(1), d_r.i_th(2),\ldots, d_r.i_th(\text{count})]=d \)

4.1.2 \( w \not\in [d_r.i_th(1), d_r.i_th(2),\ldots, d_r.i_th(\text{count})] \)

4.1.3 \( e_1 \in \text{LINKED}_L\text{IST}(T) \cdot \forall i \leq \text{count}:
   e_1.i_th(i)=d_r.i_th(i) \land (e_1.i_th(\text{count}+1)=w) \)

4.1.4 \( \text{retr2}(e_1)=d \overset{\text{w}}{\sim} \{ \text{w} \} \)

The Eiffel representation for Map is \( \text{LINKED}_L\text{IST} \). The retrieve function, \( \text{retr3} \), for MAP is defined as follows: \( \text{retr3}(d_r)=[d_r.i_th(1).\text{first}->d_r.i_th(1).\text{second},
   d_r.i_th(2).\text{first}->d_r.i_th(2).\text{second},\ldots, d_r.i_th(\text{count}).\text{first}->d_r.i_th(\text{count}).\text{second}] \)

**Theorem 5** \( \text{retr3} \) is total.

**PROOF:**

Since every element \( d_r \) in \( \text{LINKED}_L\text{IST}(\text{PAIR}[T_1,T_2]) \) has \( d_r.i_th(i) \) where \( i \) is between 1 and \( \text{count} \), there exist \( [d_r.i_th(1).\text{first}->d_r.i_th(1).\text{second}, d_r.i_th(2).\text{first}->d_r.i_th(2).\text{second},\ldots, d_r.i_th(\text{count}).\text{first}->d_r.i_th(\text{count}).\text{second}] \) for every \( d_r \). Therefore \( \text{retr3} \) is total.

**Theorem 6** \( \text{retr3} \) is adequate.

**PROOF:**

This is proved by induction.

from \( d \in \text{MAP}(\text{PAIR}[T_1,T_2]) \)
1. \( e \in \text{LINKED}_L\text{IST}(T) \cdot e.\text{count}=0 \) empty linked list
2. \( \text{retr3}(d_r)=\{ \) retr3
3. \( \exists d_r \in \text{LINKED}_L\text{IST}(T) \cdot \text{retr3}(d_r)=\{ \) \( \exists-I(1,2) \)
4. from \( d \in \text{MAP}(\text{PAIR}[T_1,T_2]) \), \( w \not\in d \)
   \( \exists d_r \in \text{LINKED}_L\text{IST}(T) \cdot \text{retr3}(d_r)=d \)
4.1 from \( dr \in \text{LINKED\_LIST(PAIR[T1,T2])} \), \( \text{retr3}(dr)=d \)

4.1.1 \( \{\text{Current.i.th(1).first}\to\text{Current.i.th(1).second},...,\)
\( \text{Current.i.th(2).first}\to\text{Current.i.th(2).second},\)
\( \text{Current.i.th(count).first}\to\text{Current.i.th(count).second}\} = d \)

4.1.2 \( \{\text{Current.i.th(1).first}\to\text{Current.i.th(1).second},\)
\( \text{Current.i.th(2).first}\to\text{Current.i.th(2).second},...,\)
\( \text{Current.i.th(count).first}\to\text{Current.i.th(count).second}\} = d \)

4.1.3 \( el\in\text{LINKED\_LIST(PAIR[T1,T2])} \) \( \forall i \cdot 1 \leq i \leq \text{count}: \)
\( el.i.th(i)=\text{Current.i.th(i)}\wedge \{el.i.th(count+1).first\to el.i.th(count+1).second\}=w \)

4.1.4 \( \text{retr3}(el)=d \cup \{w\} \)

infer \( \exists dr1 \in \text{LINKED\_LIST(PAIR[T1,T2])} \cdot \text{retr3}(dr1)=d \cup \{w\} \)

infer \( \exists dr1 \in \text{LINKED\_LIST(PAIR[T1,T2])} \cdot \text{retr3}(dr1)=d \cup \{w\} \)

infer \( \exists dr \in \text{LINKED\_LIST(PAIR[T1,T2])} \cdot \text{retr3}(dr)=d \)

The proof obligations for operations related to SET, SEQUENCE, MAP types are the domain rule and the result rule. Most proofs are straightforward. In most operations, since the pre-condition of the Object-VDM specification and the Eiffel assertion is TRUE, it is not necessary to prove the domain rule. We will show result obligations are satisfied for some sample operations.

- UNION operation in SET

The result obligation for this operation becomes:

\[
\forall i: 1..\text{Result.count},
\exists j: 1..\text{Current.count},
\text{Result.i.th(i)}=\text{Current.i.th(j)}
\]

\[
\exists k: 1..\text{other.count},
\text{Result.i.th(i)}=\text{other.i.th(k)}
\]

\[
\Rightarrow
\text{retr1(Result)}=\text{retr1(Current)} \cup \text{retr1(other)}
\]

PROOF:

from Current, other, Result \( \in \text{VDM\_SET} \)

where \( \text{retr1(Current)}=\{\text{Current.i.th(1)}, \text{Current.i.th(2)},...,\)
\( \text{Current.i.th(\text{Current.count})}\} \)

\( \text{retr1(other)}=\{\text{other.i.th(1)}, \text{other.i.th(2)},...,\text{other.i.th(\text{other.count})}\} \)

\( \text{retr1(Result)}=\{\text{Result.i.th(1)}, \text{Result.i.th(2)},...,\text{result.i.th(Result.count)}\} \)
1. from forall i: 1..Result.count,
   (exists j: 1..Current.count,
    Result.i.th(i)=Current.i.th(j)) or
   (exists k: 1..other.count,
    Result.i.th(i)=other.i.th(k))

1.1 retrl(Result)=\{Result.i.th(1), Result.i.th(2),...,Result.i.th(Result.count)\}
    =\{Current.i.th(1), Current.i.th(2),...,Current.i.th(Current.count),
        other.i.th(1),other.i.th(2),...,other.i.th(other.count)\}
    =\{Current.i.th(1), Current.i.th(2),...,Current.i.th(Current.count)\} \cup
    \{other.i.th(1),other.i.th(2),...,other.i.th(other.count)\}

   infer retrl(Result)=retrl(Current) \cup retrl(other)

2 δ(forall i: 1..Result.count,
   (exists j: 1..Current.count,
    Result.i.th(i)=Current.i.th(j)) or
   (exists k: 1..other.count,
    Result.i.th(i)=other.i.th(k)))

   infer
   (forall i: 1..Result.count,
    (exists j: 1..count,
     Current.i.th(i)=i.th(j)) or
    (exists k: 1..other.count,
     Current.i.th(i)=other.i.th(k)))
   ⇒ retrl(Result)=retrl(Current) \cup retrl(other)

• HEAD operation in SEQUENCE

   The result obligation for this operation becomes:

   Result=Current.i.th(1)
   ⇒ Result=hd(retrl(Current))

PROOF:

   from Current ∈ VDM SEQUENCE
1. from Result =Current.i_.th(1)

1.1 \[\text{hd(retr2(Current))=}\text{hd}[\text{Current.i_.th(1), Current.i_.th(2),..., Current.i_.th(Current.count)}]\]
\[=\text{Current.i_.th(1)}\]

infer retr2(Result)=hd(retr2(Current))

2. \(\delta(\text{Result }=\text{Current.i_.th(1)})\)

infer Result=Current.i_.th(1)
\[\Rightarrow\]
retr2(Result)=hd(retr2(Current))

- DomRestr operation in MAP

The result obligation for this operation becomes:

\[
\begin{align*}
\forall i: 1..\text{S.count}, \\
\exists j: 1..\text{count}, \\
\exists k: 1..\text{count}, \\
\text{Current.i_.th(k).first=}\text{S.i_.th(i)} \\
\text{Result.i_.th(j).first=}\text{Current.i_.th(k).first} \\
\text{Result.i_.th(j).second=}\text{Current.i_.th(k).second} \\
\Rightarrow \\
\text{retr3(Result)= retr1(S) } < \text{ retr3 (Current)}
\end{align*}
\]

PROOF:

from Current \(\in\) VDM.MAP

1. from \(\forall i: 1..\text{S.count},\)
\(\exists j: 1..\text{count},\)
\(\exists k: 1..\text{count},\)
\(\text{i_.th(k).first=}\text{S.i_.th(i)}\)
\(\text{Result.i_.th(j).first=}\text{Current.i_.th(k).first}\)
\(\text{Result.i_.th(j).second=}\text{Current.i_.th(k).second}\)

1.1 retr1(S) < retr3(Current)={S.i_.th(1), S.i_.th(2),...,Current.i_.th(S.count)} <
\{\text{Current.i_.th(1).first} \rightarrow \text{Current.i_.th(1).second},...,\}
\{\text{Current.i_.th(Current.count).first} \rightarrow \text{Current.i_.th(Current.count).second}\}
\(=\{\text{Result.i_.th(1).first} \rightarrow \text{Result.i_.th(1).second},...,\}
\)
\[ \text{Result}.i\_th(\text{result.count}).\text{first} \rightarrow \text{Result}.i\_th(\text{result.count}).\text{second} \}

\text{infer} \quad \text{= retr3} (\text{Result})

2. \delta (\forall i: 1..\text{S.count},
   \exists j: 1..\text{count},
   \exists k: 1..\text{count},
   \text{i}.\_th(k).\text{first} = \text{S}.\_th(i)
   \text{Result}.i\_th(j).\text{first} = \text{Current}.i\_th(i).\text{first}
   \text{Result}.i\_th(j).\text{second} = \text{Current}.i\_th(k).\text{second}

\text{infer} \quad \forall i: 1..\text{S.count},
   \exists j: 1..\text{count},
   \exists k: 1..\text{count},
   \text{i}.\_th(k).\text{first} = \text{S}.\_th(i)
   \text{Result}.i\_th(j).\text{first} = \text{Current}.i\_th(i).\text{first}
   \text{Result}.i\_th(j).\text{second} = \text{Current}.i\_th(k).\text{second}

\Rightarrow

\text{retr3} (\text{Result}) = \text{retr1} (\text{S}) \circ \text{retr3} (\text{Current})

4.1.7 OTHER CONSIDERATIONS FOR DATA REFINEMENT

The basic type in Object-VDM is easily converted to the corresponding Eiffel code.

The pairs of corresponding keywords are:

<table>
<thead>
<tr>
<th>Object-VDM</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>NAT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>CHAR+</td>
<td>STRING</td>
</tr>
</tbody>
</table>

Values in Object-Eiffel becomes constant expressions in Eiffel. Symbolic constants of simple types (integer, boolean, characters, and real) are treated in Eiffel as class attributes, which simply happen to have a fixed values for all instances of the class. For example, Object-VDM specification "VALUES zero=0" becomes the Eiffel expression "\text{zero:INTEGER is 0}".
4.2 OPERATION REFINEMENT

4.2.1 INTRODUCTION

Operation refinement (decomposition) rules to convert from abstract VDM to concrete VDM were investigated by Cliff Jones [Jon 90], and we briefly reviewed them in Section 2.3.2. Our research concentrates on transforming from concrete VDM to the Eiffel implementation. During this process, the mixture of VDM notation and Eiffel language is used. The final version consists of only Eiffel language notation. In section 4.2.2., we present rules for operation refinement which are adopted from [Jon 90] for the Eiffel syntax. We added a rule to handle quantified predicates. Section 4.3.3. gives an example how to apply these rules.

4.2.2 REFINEMENT RULES

Refinement rules for sequence, conditionals, and loops are already explained in Section 2.3.2. We will briefly review these rules, and introduce new rules for quantified predicates and logical operators.

1. Sequential refinement

\[
\{\text{pre}_1\} S_1 \{\text{pre}_2 \land \text{post}_1\}; \{\text{pre}_2\} S_2 \{\text{post}_2\}; \\
\{\text{pre}_1\} (S_1; S_2) \{\text{post}_1 \mid \text{post}_2\};
\]

where the composition of two post-conditions is defined:

\[
\text{post}_1 \mid \text{post}_2 = \exists \sigma_i \in \Sigma \cdot \text{post}_1(\overline{\sigma}, \sigma_i) \land \text{post}_2(\sigma_i, \sigma)
\]

2. Refinement into conditionals

\[
\{\text{pre} \land \text{test}\} \text{TH}\{\text{post}\}; \{\text{pre} \land \neg \text{test}\} \text{EL}\{\text{post}\}; \text{pre} \Rightarrow \delta_i(\text{test}) \\
\{\text{pre}\} \{\text{if test then TH else EL end}\} \{\text{post}\}
\]
The operation $S$ is refined to control(if) statement, when the precondition of TH satisfies $test$ condition and precondition of $S$, and the precondition of ELSE does not satisfy $test$ condition and satisfies precondition of $S$. The logical expression in the precondition is only valid if it is defined as $\delta$ in the programming language.

3 Refinement into multiple alternations

\[
\begin{align*}
\{\text{pre } test_1\} \text{TH}\{\text{post}\}; \\
\{\text{pre } \neg test_1 \land test_2\} \text{EL}_1 \{\text{post}\}; \\
\{\text{pre } \neg test_1 \land \neg test_2 \land test_3\} \text{EL}_2 \{\text{post}\}; \\
\vdots \\
\{\text{pre } \neg test_1 \land \neg test_2 \ldots \land test_n\} \text{EL}_{n-1} \{\text{post}\}; \\
\{\text{pre}\}\{\text{if test}_1 \text{ then elsif test}_2 \text{ then EL}_1 \\
\ldots \\
\text{elsif test}_n \text{ then EL}_{n-1} \\
\text{else EL}_n \\
\text{end}\} \\
\{\text{post}\}
\end{align*}
\]

This rule is a more general case of rule 2.

4. Weakening triples

\[
\{\text{pre}_s\} \Rightarrow \text{pre}; \{\text{pre}\}S\{\text{post}\}; \text{post} \Rightarrow \text{post}_w \\
\{\text{pre}_s\}S\{\text{post}_w\}
\]

This rule asserts that anything which satisfies a specification necessarily satisfies a weaker one. Observe that a 'weaker' specification is one with a narrower pre-condition or a wide post-condition. In either case, the implication could be just an equivalence thus changing only either a precondition or a post-condition of specification.
5. Introduce blocks

\{\text{pre} \land v = e\}\{\text{post}\};
\{\text{pre}\}\text{local } v; \text{ do } v := e; S \text{ end } \{\exists v \cdot \text{post}\}

A block construct is needed to introduce the local variables.

6. Decompose into loops

\{\text{inv} \land \text{test}\}\{\text{inv} \land \text{sofar}\}; \text{ inv } \Rightarrow \delta_i(\text{test})
\{\text{inv}\}\{\text{from } v := e \text{ invariant } \text{inv} \text{ variant } \text{V} \text{ until } \neg \text{test loop } S \text{ end}\}\{\text{inv} \land \neg \text{test} \land (\text{sofar} \lor \text{iden})\}

A loop invariant \((\text{inv} : \Sigma \rightarrow B)\) is identified which limits the states which can arise in the computation and that a relation \((\text{sofar} : \Sigma \times \Sigma \rightarrow B)\) is given which holds over one or more iteration of the loop; technically the requirement that \((\text{sofar} \mid \text{sofar} \Rightarrow \text{sofar})\) is stated by saying that \(\text{sofar}\) must be transitive.

It is also necessary to ensure the termination and this can be done by ensuring that the \(\text{sofar}\) is well-founded over the set defined by \(\text{inv}\).

7. Decompose a clause which has an existential quantifier.

\{\text{predicates}\}\{\text{predicates} \land \text{post}\}
\{\exists \text{elmt } \in \text{Set} \cdot \text{predicates}\}
local b: \text{boolean}
    temp: \text{type}
    do
    from
        temp := \text{emptyset}
        b := \text{true}
    invariant temp.IsSubsetOf(\text{Set})
    variant \text{Set.card-temp.card}
until temp = set
loop
  elem := set.diff(temp).choose;
  temp := temp.add(elem);
  if (predicates) then S
    b := false
  end
end
if b then
  error message
end
{post}

When a predicate uses quantifiers such as ∀, ∃, or ¬, the predicate is converted to a loop statement in Eiffel code. The initial value of a boolean local variable b is set to true.

8. Decompose a clause which has a universal quantifier.

{b=true;}S{b=true; ∧ post}
{∀ elmt ∈ Set . predicates}
local b:boolean
temp:type
do
  from
  temp := emptyset
  b := true
invariant temp.IsSubsetOf(Set)
variant Set.card-temp.card
until temp = set
loop
  elem := set.diff(temp).choose;
  temp := temp.add(elem);
  if not (predicates) then b := false
  end
end
if b then
  S
9. Decompose a clause which has a negation of existential quantifier.

\[
\{ b = \text{true}; \} S \{ b = \text{true}; \land \text{post} \} \\
\{ \neg \exists \text{elmt} \in \text{Set} \cdot \text{predicates} \}
\]

local b:boolean
  temp:type
  do
    from
      temp := \text{emptyset}
      b := \text{true}
  invariant temp.IsSubsetOf(\text{Set})
  variant \text{Set.card} - temp.card
  until temp = set
  loop
    elem := set.diff(temp).choose;
    temp := temp.add(elem);
    if (predicates) then b := \text{false}
  end
if b then
  S
end
{post}

10. Decompose a clause which has a negation of universal quantifier.

\[
\{ \text{predicates} \} S \{ \text{predicates} \land \text{post} \} \\
\{ \neg \forall \text{elmt} \in \text{Set} \cdot \text{predicates} \} \{ \text{post} \}
\]

local b:boolean
  temp:type
  do
    from
      temp := \text{emptyset}
      b := \text{true}
  invariant temp.IsSubsetOf(\text{Set})
variant Set.card-temp.card
until temp=set
loop
  elem:=set.diff(temp).choose;
  temp:=temp.add(elem);
  if (predicates) then S
    b:=false
  end
end
if b then
  error message
end
end
{post}

11. Refinement into separate statements

{pre}{if test then $TH_1 \land TH_2$ end}{post};
{pre} {if test then $TH_1; TH_2$ end} {post}

The predicates connected by logical expression $\land$ in THEN clause are converted to statements separated by ",".

12. Refinement of $\land$ in test in if or loop statement

{pre}{if ($test_1 \land ... \land test_n$) then TH end}{post}
{pre} {if ($test_1$ and ...and $test_n$) then TH end} {post}
{pre}{from $v:=e$ invariant inv variant V until $test_1 \land ... \land test_n$ loop S end}{post}
{pre}{from $u:=e$ invariant mu variant V until $test_1$ and...and($test_n$ loop S end}{post}

The logical expression $\land$ in the test is converted to the and.

13. Refinement of $\lor$ in test in if or loop statement

{pre}{if ($test_1 \lor ... \lor test_n$) then TH end}{post}
{pre} {if ($test_1$ or $test_2$ ...or $test_n$) then TH end} {post}
The logical expression \( \vee \) in the precondition is converted to the \( \text{or} \).

### 4.2.3 An Example of Operation Refinement

To show how the operation rules are applied, we use an example from [Jon 90], which specifies the multiplication procedure.

\[
\text{MULT} \\
\text{ext wr m,n,r: Z} \\
\text{pre true} \\
\text{post } r = \text{m} \times \text{n}
\]

MULT can be decomposed by the two sequential operation \( \text{COPYPOS;POSMULT} \). COPYPOS copies the variables \( m \) and \( n \) into new variables (usually the same or negative values of the original variables) to make at least one variable is positive. POSMULT assumes one of the variables was definitely positive so that a loop could be designed which counted up to that value. The COPYPOS and POSMULT operations in Object-VDM are specified:

\[
\text{COPYPOS} \\
\text{ext rd m,n :Z} \\
\text{wr mp,nn :Z} \\
\text{pre true} \\
\text{post } 0 \leq mp \land mp \times nn = \text{m} \times \text{n}
\]

\[
\text{POSMULT} \\
\text{ext rd mp,nn :Z} \\
\text{wr r :Z} \\
\text{pre } 0 \leq mp \\
\text{post } r = mp \times nn
\]
COPYPOS is further decomposed using conditionals.

COPYPOS: if \( 0 \leq m \) then TH else EL end

where:

TH

\[
\text{ext } \text{rd } m,n : \mathbb{Z} \\
\text{wr } mp,nn : \mathbb{Z} \\
\text{pre } 0 \leq m \\
\text{post } 0 \leq mp \land mp \cdot nn = \overline{m} \cdot \overline{n}
\]

EL

\[
\text{ext } \text{rd } m,n : \mathbb{Z} \\
\text{wr } mp,nn : \mathbb{Z} \\
\text{pre } m < 0 \\
\text{post } 0 \leq mp \land mp \cdot nn = \overline{m} \cdot \overline{n}
\]

We get a refined specification for these two operations by applying weakening triples rules.

TH

\[
\text{ext } \text{rd } m,n : \mathbb{Z} \\
\text{wr } mp,nn : \mathbb{Z} \\
\text{pre } \text{true} \\
\text{post } mp = \overline{m} \land nn = \overline{n}
\]

EL

\[
\text{ext } \text{rd } m,n : \mathbb{Z} \\
\text{wr } mp,nn : \mathbb{Z} \\
\text{pre } \text{true} \\
\text{post } mp = -\overline{m} \land nn = -\overline{n}
\]
The Eiffel code corresponding to the VDM specification for COPYPOS is:

COPYPOS
m,n: INTEGER
mp,nn : INTEGER
if 0 \leq m then
    mp=m
    nn=n
else
    mp=-m
    nn=-n
end

By introducing blocks, POSMULT can be refined as follows:

POSMULT:
local t:INTEGER do t:=0 LOOP end
Where
LOOP
ext rd mp,nn :Z
wr t,r :Z
pre \ r = t * nn \land t \leq mp
post \ r = \overrightarrow{mp} * \overrightarrow{nn} \land t = \overrightarrow{mp}

LOOP is decomposed using the refinement rule for loop.
LOOP
from t:=0
invariant r=t*nn
variant t
until t ≠ mp
loop
  t:=t+1;
  r:=r+nn
end

The final Eiffel code for POSMULT is:

POSMULT
local t:INTEGER
do
t:=0
from t:=0
invariant r=t*nn
variant t
until t ≠ mp
loop
  t:=t+1;
  r:=r+nn
end
end
4.3 STRUCTURE REFINEMENT

In the structure refinement process, the object-oriented facilities and structure structures of Object-VDM are transformed to those of Eiffel. The basic unit of Object-VDM and Eiffel is a class. Both of them have the same keyword class. The keyword superclass in the Object-VDM is implemented by the inherit clause into Eiffel. The state clause, the functions clause and the operations clause are integrated into feature clause in Eiffel. The public/private clause in Object-VDM specifies the visibility of operations. In Eiffel, the visibility is implemented by the export clause or client lists in the feature facility.

4.3.1 CLASS

The basic unit of Object-VDM and Eiffel is a class. Both of them have the same keyword class.

4.3.2 SUPERCLASS

The keyword superclass in Object-VDM is implemented in the inherit clause in Eiffel. For example, consider the following Object-VDM specification.

```
Class A

end superclass A

endclass
```

The corresponding Eiffel implementation is:

```
Class A

end;

inherir A

end;
```
When an operation in a subclass overrides the operation in a superclass, Eiffel uses the feature redefinition facility.

```
Class A    Class B
----------    ----------
operations    superclass A
opl( ---)    ----------
----------    operations
endclass      op1( ---)
              ----------
endclass
```

The corresponding Eiffel implementation is:

```
Class A    Class B inherit
----------    ----------
feature        A redefine op1
opl( ---) is    feature
----------    op1( ---) is
end;        ----------
end;
```

### 4.3.3 PRIVATE/PUBLIC

The `public/private` clause in the Object-VDM specifies the visibility of operations. In Eiffel, the visibility is implemented by `export` clause or client lists in the `feature` facility. Consider following specification.

```
Class A
----------
public op1
----------
endclass
```

This can be converted to Eiffel code by using `export` clause.

```
Class A
export op1
----------
```
Another way to do the conversion is by using the feature facility. There are four ways of expressing the feature facility. We specify which clients can use the attributes and operations under the feature clause.

- feature---------Any clients
- feature{A,B}-----only class A, and class B
- feature{}--------No class
- feature{NONE}---No class

Since op1 is public to all clients, it can be written as following Eiffel notation.

Class A

feature

end;

4.3.4 STATE

The state clause in Object-VDM has two clauses: invariant clause and initialization clause. Invariant in the state definition is converted to the class invariant. Initialization in the state definition is converted to the make function in Eiffel.

Class A
state
  inv pred1
  init pred2
  -------
endclass

This unit can be converted to the following Eiffel code.

Class A
  -------
  feature
    make
    pred2
    -------
    invariant pred1
  end;

4.4 SUMMARY

In this Chapter, we described the theory of method to Object-VDM to Eiffel. The method is based on the refinement method in VDM by Jones. The method consists of three steps: data refinement, operation refinement, and structure refinement. In data refinement, the mathematical data models in object-VDM such as SET, SEQUENCE, and MAP are converted to Eiffel data structures. We created Eiffel libraries to do this. We also proved that this conversion is correct. In operation refinement, we modified and added rules to the original refinement to obtain Eiffel code. Object-oriented features are converted in the structure refinement step.
We present a case study to illustrate the transformation and the associative proof obligations defined in Chapter 4. The case study is a part of the system to computerize a student records system in a university.

5.1 OBJECT-VDM SPECIFICATION

In the student record system, every student has his/her name and identification number. Student data also includes earned credit hours and GPA. Operations, such as adding new students, adding credit hours, changing GPA, and checking the eligibility to graduate are needed. Graduate students have additional requirements. Graduate students must pass the comprehensive exam first, propose the thesis, and finally defend the thesis. Therefore, graduate student data has the following additional attributes: status_exam, status_proposal, thesis_title, status_defence. To handle graduate students, operations such as reporting the pass of exam (exam), controlling the data for proposal (proposal), and defense of thesis (defence) are needed. The Object-VDM specification for TYPE of student records system is in Figure 5.1. The student class is described in Figure 5.2, and the graduate student class is specified in Figure 5.3.

In the add_credit operation in Figure 5.2, the earned credit hour and GPA are updated. For graduate requirements, ordinary students should earn 140 credit hours and a GPA greater than 2.0, while graduate students must earn 30 credit hours, have a GPA greater than 3.0, and defend their theses. In Figure 5.3, we use both type modules and class modules. Since graduate_student_type is a specialization of a student_type, a graduate_student_type is a subtype of a student_type. Graduate_student class also uses the same operation such as adding a new student (add), and adding
new credit hours (add_credit). Thus, Graduate_Student class uses operations inherited from the super class, Student class. Graduate requirements for graduate students differ from those of undergraduate students. Therefore, the Graduate_Student class has a different graduate method.

```plaintext
type student_type
    id: NAT
    name: CHAR+
    credit: NAT
    gpa: REAL
    final: BOOLEAN
endtype

type graduate_student_type
    supertype student_type
    status_exam: BOOLEAN
    status_proposal: BOOLEAN
    status_defence: BOOLEAN
    thesis_title: CHAR+
endtype
```

Figure 5.1: Object-VDM specification for TYPE of the student records system

5.2 DATA REFINEMENT

In this process, attributes in the TYPE part and data structures in CLASS part of Object-VDM are converted to Eiffel data structures. Especially, the typical types such as sets, maps, and sequences are converted to Eiffel classes. We develop libraries for these types. These library use LINKED LIST data structures in Eiffel. The class names of libraries for sets, sequences, and maps are VDM-Set, VDM-Sequence, and VDM-Map. We convert the data structures first, then we convert operations related to these data structures. Since we use Eiffel libraries, the notation in this step is a mixture of Object-VDM and Eiffel. Figure 5.4 depicts the type refinement to Eiffel and Figure 5.5 and 5.6 describes data refinement.
class Student

state Student_Class of
  ST:student_type - set
  st:student_type
end

operations
  add(id:NAT, name:CHAR+, credit:NAT, gpa:REAL)
    ext wr ST: student_type - set
    wr st: student_type
    pre ¬∃ st ∈ ST : st.id=new_id
    post st.id=new_id ∧ st.name=new_name ∧
      st.credit=new_credit ∧ st.gpa=new_credit ∧
      ST=ST ∪ {st}

add_credit(id:NAT, h:NAT, g:REAL)
  ext wr ST: student_type - set
  wr st: student_type
  pre ∃ st ∈ ST : st.id=id
  post st.gpa = (st.credit * st.gpa + h * g)/(st.credit + h) ∧
    st.credit=st.credit + h ∧
    ST=ST \ {st} ∪ {st}

graduate(id:NAT):BOOLEAN
  ext wr ST: student_type - set
  wr st: student_type
  pre ∃ st ∈ ST : st.id=id
  post if (st.credit >=140 ∧ st.gpa>=2.0 )
    then return true
    else return false
endclass

Figure 5.2: Object-VDM specification of the student records system
class Graduate_Student
    superclass student
    state Graduate_Student_Class of
        G_ST: graduate_student_type-set
        g_st: graduate_student_type
    end
    operations
        exam(id:NAT)
            ext wr G_ST: graduate_student_type-set
            wr g_st: graduate_student_type
            pre ∃ g_st ∈ G_ST . g_st.id = id
            post g_st.status_exam = T
        proposal(id:NAT, title:CHAR+)
            ext wr G_ST: graduate_student_type-set
            wr g_st: graduate_student_type
            pre ∃ g_st ∈ G_ST . g_st.id = id ∧ g_st.status_exam = true
            post g_st.status_proposal = true ∧ g_st.thesis_title = title
        defence(id:NAT):BOOLEAN
            ext wr G_ST: graduate_student_type-set
            wr g_st: graduate_student_type
            pre ∃ g_st ∈ G_ST . g_st.id = id ∧ g_st.status_proposal = true
            post g_st.status_defence = true
        graduate(id:NAT):BOOLEAN
            ext wr G_ST: graduate_student_type-set
            wr g_st: graduate_student_type
            pre ∃ st ∈ ST . st.id = id
            post if (g_st.credit >= 30 ∧ g_st.gpa >= 3.0 ∧
                g_st.status_exam = true ∧ g_st.status_defence = T)
            then return true
            else return false
    endclass

Figure 5.3: Object-VDM specification of the graduate student records system
type student_type
  feature
    id: INTEGER
    name: STRING
    credit: INTEGER
    gpa: REAL
    finish: BOOLEAN
endtype

type graduate_student_type
  supertype student_type
  feature
    status_exam: BOOLEAN
    status_proposal: BOOLEAN
    status_defence: BOOLEAN
    thesis_title: STRING
endtype

Figure 5.4: Type refinement of student records system

In this refinement, NAT and CHAR in Object-VDM data type are converted to INTEGER and STRING in Eiffel data type. The following shows the specific refinement in this step.

<table>
<thead>
<tr>
<th>Object-VDM</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: NAT</td>
<td>id: INTEGER</td>
</tr>
<tr>
<td>name: CHAR</td>
<td>name: STRING</td>
</tr>
<tr>
<td>thesis's title: CHAR</td>
<td>thesis's title: STRING</td>
</tr>
</tbody>
</table>
class Student
    state Student_Class of
        ST: VDM_Set[student_type]
        st: student_type
    end
    operations
        add(new_id: INTEGER, new_name: STRING, new_credit: INTEGER, 
            new_gpa: REAL)
            ext wr ST: VDM_Set[student_type]
            wr st: student_type
            pre ¬∃ ST.IsElmt(st) • st.id=new_id 
            post st.id=new_id ∧ st.name=new_name ∧
                st.credit=new_credit ∧ st.gpa=new_credit ∧ 
                ST=(old ST).add(st)
        add_credit(id: INTEGER, h: INTEGER, g: REAL)
            ext wr ST: VDM_Set[student_type]
            wr st: student_type
            pre ∃ ST.IsElmt(st) • st.id=id 
            post st.id=id ∧ st.gpa=(st.credit * st.gpa + h * g)/(st.credit + h) ∧ 
                st.credit= st.credit + h ∧ 
                ST=((old ST).delete(old st)).add(st)
        graduate(id: INTEGER)
            ext wr ST: VDM_Set[student_type]
            wr st: student_type
            pre ∃ ST.IsElmt(st) • st.id=id 
            post if (st.credit >=140 ∧ st.gpa>=2.0 )
                then return true 
                else return false 
    endclass

Figure 5.5: Data refinement of the student records system

The type student_type-set in Object-VDM is converted to VDM_Set[student_type]. NAT and CHAR+ are converted to INTEGER and STRING respectively. The followings show the specific refinement in this step.

<table>
<thead>
<tr>
<th>Object-VDM</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST:student_type-set</td>
<td>ST:VDM_Set[student_type]</td>
</tr>
<tr>
<td>new_id: NAT</td>
<td>new_id: INTEGER</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
new_name: CHAR$^+$
new_credit: NAT
id,h: NAT

new_name: STRING
new_credit: INTEGER
id,h: INTEGER

class Graduate_Student
 superclass student
 state Graduate_Student_Class of
 G_ST: VDM_Set[graduate_student_type]
g_st: graduate_student_type
end
operations
exam(id:INTEGER)
ext wr G_ST: VDM_Set[graduate_student_type]
wr g_st: graduate_student_type
pre $\exists$ g_st \in G_ST \cdot g_st.id=id
post g_st.status_exam=true \land
       G_ST=((old G_ST).delete(old g_st)).add(g_st)
proposal(id:INTEGER, title:STRING)
ext wr G_ST:
  wr g_st: graduate_student_type
  pre $\exists$ G_ST.IsElmt(g_st) \cdot g_st.id=id \land g_st.status_exam=true
  post g_st.status_proposal=true \land g_st.thesis_title=title \land
       G_ST=((old G_ST).delete(old g_st)).add(g_st)
defence(id:INTEGER)
ext wr G_ST: VDM_Set[graduate_student_type]
wr g_st: graduate_student_type
pre $\exists$ G_ST.IsElmt(g_st) \cdot g_st.id=id \land g_st.status_proposal=true
post g_st.status_defence=true \land
       G_ST=((old G_ST).delete(old g_st)).add(g_st)
graduate(id:INTEGER):BOOLEAN
next wr G_ST: VDM_Set[graduate_student_type]
wr g_st: graduate_student_type
pre $\exists$ st \in S \cdot st.id=id
post if (g_st.credit>=30 \land g_st.gpa>=3.0 \land
          g_st.status_exam=true \land g_st.status_defence=T)
       then return true
       else return false
endclass

Figure 5.6: Data refinement of the graduate student records system
VDM.Set[graduate_student_type] is an Eiffel data structure corresponding to Graduate_student_type-set in Object-VDM. NAT and CHAR+ are converted to INTEGER and STRING respectively. The following shows the specific refinement in this step.

<table>
<thead>
<tr>
<th>Object-VDM</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.ST:graduate_student_type-set</td>
<td>ST:VDM.Set[graduate_student_type]</td>
</tr>
<tr>
<td>id: NAT</td>
<td>id: INTEGER</td>
</tr>
<tr>
<td>title: CHAR+</td>
<td>title: STRING</td>
</tr>
</tbody>
</table>

5.3 OPERATION REFINEMENT

In operation refinement, each operation which is denoted by predicates is converted to Eiffel language control structures. The operation rules in section 5.2 are applied. Logical operators such as negation, disjunction, conjunction, and two quantifiers (i.e., existential quantifier and universal quantifier) are converted to programming language control structures. The notation in this step is a mixture of predicate expression and Eiffel code. We will describe how the add operation in student records system can be transformed.

- Step 1.

Since the precondition of the add operation contains a negation of the existential quantifier, we can apply rule 9. The result after applying rule 9 is in Figure 5.7. The refinement is correct because S satisfies:

\[
\{ b = \text{true} \}
\]
\[
\text{st.id} = \text{new.id} \land \text{st.name} = \text{new.name} \land
\]
\[
\text{st.credit} = \text{new.credit} \land \text{st.gpa} = \text{new.credit} \land
\]
\[
\text{ST} = \text{ST.add(st)};
\]
\[
\{ b = \text{true} \land \text{st.id} = \text{new.id} \land \text{st.name} = \text{new.name} \land
\]
\[
\text{st.credit} = \text{new.credit} \land \text{st.gpa} = \text{new.credit} \land
\]
\[
\text{ST} = \text{ST.add(st)}\}
\]
add(new_id:INTEGER, new_name:STRING, new_credit: INTEGER, new_gpa:REAL) is
local b: BOOLEAN
do from
  TEMP:= emptyset;
  b:=true;
invariant TEMP.isSubsetof(ST)
variant ST.card - TEMP.card
until TEMP=ST or b=false
loop
  st:=(ST.diff(TEMP)).choose;
  TEMP:=TEMP.add(st);
  if st.id=new_id then b:=false
end
if b then
  st.id=new_id ∧ st.name=new_name ∧
  st.credit=new_credit ∧ st.gpa=new_credit ∧
  ST=ST.add(st);
end
end;

Figure 5.7: The First Operation refinement for add operation in the student records system

• Step 2.

To transform ∧ in then part of if statement, we applied rule 11. The result is
in Figure 5.8 and 5.9. The following shows the specific refinement in this step.

if b then
  st.id=new_id ∧ st.name=new_name ∧
  st.credit=new_credit ∧ st.gpa=new_credit ∧
  ST=ST.add(st);
end
⇒
if b then
    st.id=new_id; st.name=new_name;
    st.credit=new_credit; st.gpa=new.credit;
    ST=ST.add(st);
end

Other operations are transformed in the same way by applying adequate rules several times. The final Eiffel code for the operations in student class and graduate_student class are presented in Figure 5.8, 5.9 and Figure 5.10, 5.11 respectively.

```
add(new_id:INTEGER, new_name:STRING, new_credit: INTEGER,
     new_gpa:REAL) is
local b: BOOLEAN
do
  from
    TEMP:=emptyset;
    b:=true;
  invariant TEMP.isSubsetof(ST)
  variant ST.card - TEMP.card
  until TEMP=ST or b=false
  loop
    st:=(ST.diff(TEMP)).choose;
    TEMP:=TEMP.add(st);
    if st.id=new_id then b:=false
  end
  if b then
    st.id=new_id; st.name=new_name;
    st.credit=new_credit; st.gpa=new.credit;
    ST=ST.add(st);
  end
end;
```

Figure 5.8: Operation refinement for the operations in the student records system (part 1)
In student class, we got the following final Eiffel code for other operations.

```
add_credit(id: INTEGER, h:INTEGER, g:REAL) is
  local b: BOOLEAN
  do
    from
      TEMP:= emptyset;
      b:=false;
    invariant TEMP.isSubsetof(ST)
    variant ST.card - TEMP.card
    until TEMP=ST or b=true
    loop
      st:=(ST.diff(TEMP)).choose;
      TEMP:=TEMP.add(st);
      if st.id=new_id then
        ST=ST.delete(st);
        st.gpa=(st.credit*st.gpa+h*g)/(st.credit+h);
        st.credit= st.credit + h;
        ST=ST.add(st)
      end
    end
  end

graduate(id: INTEGER):BOOLEAN is
  do
    from
      TEMP:= emptyset;
    invariant TEMP.isSubsetof(ST)
    variant ST.card - TEMP.card
    until TEMP=ST
    loop
      st:=(ST.diff(TEMP)).choose;
      TEMP:=TEMP.add(st);
      if st.credit >=140 or st.gpa>=2.0 then Result=true
        else Result=false
      end
    end
  end
```

Figure 5.9: Operation refinement for the operations in the student records system (part 2)
In graduate student class, we got Eiffel code for following operations:

```eiffel
exam(id: INTEGER) is
do
from
    TEMP:= emptyset;
invariant TEMP.isSubsetof(G_ST)
variant G_ST.card - TEMP.card
until TEMP=G_ST
loop
    g_st:=(G_ST.diff(TEMP)).choose;
    TEMP:=TEMP.add(g_st);
    if g_st.id=id then
        g_st.status_exam=T
        G_ST=(G_ST.delete(old g_st)).add(g_st)
    end
end
end;

proposal(id: INTEGER, title:string) is
local b: BOOLEAN
do
from
    TEMP:= emptyset;
    b:=false;
invariant TEMP.isSubsetof(ST)
variant ST.card - TEMP.card
until TEMP=ST or b=true
loop
    g_st:=(G_ST.diff(TEMP)).choose;
    TEMP:=TEMP.add(g_st);
    if (g_st.id=id and g_st.status_exam=T) then
        G_ST=G_ST.delete(g_st);
        g_st.status_proposal=T;
        g_st.thesis_title=title;
        G_ST=G_ST.add(g_st)
    end
end
end
```

Figure 5.10: Operation Refinement for operations in graduate student records system (part 1)
defence(id: INTEGER)
  local b: BOOLEAN
  do
    from
      TEMP := emptyset; b := false;
    invariant TEMP.isSubsetof(ST)
    variant ST.card - TEMP.card
    until TEMP=ST or b=true
    loop
      g_st := (G_ST.diff(TEMP)).choose;
      TEMP := TEMP.add(g_st);
      if (g_st.id = id and g_st.status_exam = true) then
        G_ST = G_ST.delete(g_st);
        g_st.status_proposal = true;
        g_st.thesis_title = title;
        G_ST = G_ST.add(g_st)
      end
    end
  end;

graduate(id: INTEGER): BOOLEAN is
  local b: BOOLEAN
  do
    from
      TEMP := emptyset; b := false;
    invariant TEMP.isSubsetof(G_ST)
    variant ST.card - TEMP.card
    until TEMP=ST
    loop
      st := (ST.diff(TEMP)).choose;
      TEMP := TEMP.add(st);
      if g_st.id = id then
        if (g_st.credit >= 30 and g_st.gpa >= 3.0 and 
            g_st.status_exam = true and g_st.status_defence = T) 
        then Result = true
        else Result = false
      end
    end
  end
end

Figure 5.11: Operation Refinement for operations in graduate student records system (part 2)
5.4 Structure Refinement

Structure refinement is the final process. The object-oriented facilities and structures are converted to Eiffel code. All operations are integrated. The notation in this step is purely Eiffel code. The Eiffel code for TYPE is described in Figure 5.12 and the Eiffel implementation for student record system are illustrated in Figure 5.13 and 5.14 and Figure 5.15 and 5.16.

<table>
<thead>
<tr>
<th>Class student_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>export id, name, credit, gpa, finish</td>
</tr>
<tr>
<td>feature</td>
</tr>
<tr>
<td>id: INTEGER</td>
</tr>
<tr>
<td>name: STRING</td>
</tr>
<tr>
<td>credit: INTEGER</td>
</tr>
<tr>
<td>gpa: REAL</td>
</tr>
<tr>
<td>finish: BOOLEAN</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class graduate_student_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>export status_exam, status_proposal, status_defence, thesis_title</td>
</tr>
<tr>
<td>inherit student_type</td>
</tr>
<tr>
<td>feature</td>
</tr>
<tr>
<td>status_exam: BOOLEAN</td>
</tr>
<tr>
<td>status_proposal: BOOLEAN</td>
</tr>
<tr>
<td>status_defence: BOOLEAN</td>
</tr>
<tr>
<td>thesis_title: STRING</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

Figure 5.12: Eiffel code for the TYPE of student records system

The keyword type, supertype in Object-VDM is changed to the keywords class, inherit respectively. All attributes are exported by using keyword export in order that subclasses can use them. The following show the specific refinement in this step.

<table>
<thead>
<tr>
<th>Object-VDM</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>type:(graduate_)student_type</td>
<td>Class:(graduate_)student_type</td>
</tr>
<tr>
<td>supertype student_type</td>
<td>inherit student_type</td>
</tr>
<tr>
<td>export id,name,credit,gpa,finish</td>
<td>export id,name,credit,gpa,finish</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Class student
   export add, add_credit
feature
   ST, TEMP : VDM_Set[student-type]
   st: student_type
add(new_id:INTEGER, new_name:STRING, new_credit: INTEGER,
    new_gpa:REAL) is
   local b: BOOLEAN
   do
      from TEMP:= emptyset;
      b:=true;
      invariant TEMP.isSubsetof(ST)
   variant ST.card - TEMP.card
   until TEMP=ST or b=false
   loop
      st:=(ST.diff(TEMP)).choose;
      TEMP:=TEMP.add(st);
      if st.id=new_id then b:=false
      end
      if b then
         st.id=new_id; st.name=new_name;
         st.credit=new_credit; st.gpa=new_credit;
         ST=ST.add(st)
      end
   end
end;

Figure 5.13: Eiffel code for the student records system (part 1)

Figure 5.13 and 5.14 combine the attributes part of Figure 5.5 and operation part of Figure 5.8 and 5.9. The keyword state and operations are integrated into a keyword feature. TEMP is declared because operations need the variable. The following shows the specific refinement in this step.
add_credit(id: INTEGER, h:INTEGER, g:REAL) is
  local b: BOOLEAN
  do
    from
      TEMP:= emptyset;
      b:=false;
    invariant TEMP.isSubsetof(ST)
    variant ST.card - TEMP.card
  until TEMP=ST or b=true
  loop
    st:=(ST.diff(TEMP)).choose;
    TEMP:=TEMP.add(st);
    if st.id=new_id then
      ST=ST.delete(st)
      st.gpa=(st.credit*st.gpa+h*g)/(st.credit+h)
      st.credit= st.credit + h
      ST=ST.add(st);
    end
  end
end

graduate(id: INTEGER):BOOLEAN is
  do
    from
      TEMP:= emptyset;
    invariant TEMP.isSubsetof(ST)
    variant ST.card - TEMP.card
  until TEMP=ST
  loop
    st:=(ST.diff(TEMP)).choose;
    TEMP:=TEMP.add(st);
    if st.credit >=140 or st.gpa>=2.0
      then Result=true
    else Result=false
    end
  end
end
end

Figure 5.14: Eiffel code for the student records system (part 2)
class Student
    state Student_Class of
        ST: VDM_Set[student_type]
        st: student_type
    end

    operations
        add(new_id: INTEGER, new_name: STRING, new_credit: INTEGER, 
            new_gpa: REAL) is
            local b: BOOLEAN
            do
                from
                    TEMP := emptyset;
                    b := true;
                invariant TEMP.isSubsetof(ST)
                variant ST.card - TEMP.card
                until TEMP = ST or b = false 
                loop
                    st := (ST.diff(TEMP)).choose;
                    TEMP := TEMP.add(st);
                    if st.id = new_id then b := false
                end
                if b then
                    st.id = new_id; st.name = new_name;
                    st.credit = new_credit; st.gpa = new_gpa;
                    ST := ST.add(st);
                end
            end
    endclass

⇒

Class student
    export add, add_credit

    feature
        ST, TEMP : VDM_Set[student-type]
        st: student_type
        add(new_id: INTEGER, new_name: STRING, new_credit: INTEGER, 
            new_gpa: REAL) is
            local b: BOOLEAN
            do
                from
                    TEMP := emptyset;
                    b := true;
                invariant TEMP.isSubsetof(ST)
                variant ST.card - TEMP.card
                until TEMP = ST or b = false 
                loop
                    st := (ST.diff(TEMP)).choose;
                    TEMP := TEMP.add(st);
                    if st.id = new_id then b := false
                end
                if b then
                    st.id = new_id; st.name = new_name;
                    st.credit = new_credit; st.gpa = new_gpa;
                    ST := ST.add(st);
                end
            end
        end
b:=true;
invariant TEMP.isSubsetof(ST)
variant ST.card - TEMP.card
until TEMP=ST or b=false
loop
  st:=(ST.diff(TEMP)).choose;
  TEMP:=TEMP.add(st);
  if st.id=new_id then b:=false
end
if b then
  st.id=new_id; st.name=new_name;
  st.credit=new_credit; st.gpa=new_credit;
  ST=ST.add(st)
end
end

endclass

Figure 5.15 and 5.16 combines Figure 5.6, 5.10 and 5.11 in the same way and describes the final operational refinement of graduate_student class.

5.5 Summary

In this chapter, we applied the refinement method as discussed in Chapter 4 to a students records system. Recall there are three phases in the refinement: Type and data refinement, operation refinement and structure refinement. In type and data refinement, we use Eiffel libraries to convert the mathematical data structures in Object-VDM to Eiffel data structures. For example, student_type-set is converted to VDM_Set[student_type] in Eiffel. In operation refinement, we applied the rules in Section 4.2. We showed step by step refinement, and we proved that each refinement is correct. We converted object-oriented facilities such as classes, superclasses and inheritances in Object-VDM to those of Eiffel in structure refinement.
Class graduate_student

inherit student

feature

G_ST, TEMP : VDM_Set[graduate_student-type]
g_st: graduate_student_type

exam(id: INTEGER) is
do
from
    TEMP:= emptyset;
invariant TEMP.isSubsetof(G_ST)
variant G_ST.card - TEMP.card
until TEMP=G_ST
loop
    g_st:=(G_ST.diff(TEMP)).choose;
    TEMP:=TEMP.add(g_st);
    if g_st.id=id then
        G_ST=G_ST.delete(g_st)
        g_st.status_exam=T
        G_ST=G_ST.add(g_st);
    end
end
end;

proposal(id: INTEGER, title:string) is
local b: BOOLEAN
do
from
    TEMP:= emptyset; b:=false;
invariant TEMP.isSubsetof(ST)
variant ST.card - TEMP.card
until TEMP=ST or b=true
loop
    g_st:=(G_ST.diff(TEMP)).choose;
    TEMP:=TEMP.add(g_st);
    if (g_st.id=id and g_st.status_exam=T) then
        G_ST=G_ST.delete(g_st));
        g_st.status_proposal=T  g_st.thesis_title=title;
        G_ST=G_ST.add(g_st)
    end
end
end

Figure 5.15: Eiffel code for the graduate student records system (part 1)
defence(id: INTEGER)
    local b: BOOLEAN
    do
        from
            TEMP:= emptyset; b:=false;
        invariant TEMP.isSubsetof(ST)
        variant ST.card - TEMP.card
        until TEMP=ST or b=true
        loop
            g_st:=(G_ST.diff(TEMP)).choose;
            TEMP:=TEMP.add(g_st);
            if (g_st.id=id and g_st.status_exam=T) then
                G_ST=G_ST.delete(g_st);
                g_st.status_proposal=T;
                g_st.thesis_title=title;
                G_ST=G_ST.add(g_st)
            end
        end
    end
end

graduate(id: INTEGER): BOOLEAN is
    local b: BOOLEAN
    do
        from
            TEMP:= emptyset; b:=false;
        invariant TEMP.isSubsetof(G_ST)
        variant ST.card - TEMP.card
        until TEMP=ST
        loop
            st:=(ST.diff(TEMP)).choose;
            TEMP:=TEMP.add(st);
            if g_st.id=id then
                if (g_st.credit>=30 and g_st.gpa>=3.0 and
                g_st.status_exam=true and g_st.status_defence=T)
                    then Result=true
                else Result=false
            end
        end
    end
end
end

Figure 5.16: Eiffel code for the graduate student records system (part 2)
CHAPTER 6
CONCLUSION

6.1 SUMMARY

To reduce the distance between the beginning stage (requirement analysis) and the implementation stage in traditional software development life cycle, a formal development method has been recommended. We propose formal development from object-oriented VDM to Eiffel by using a modified refinement method. VDM is one of the widely used formal specification languages and Eiffel is an object-oriented programming language which has many strong facilities such as assertions and genericity.

There are two steps to using formal methods: formal specification and formal development. First the system is specified by using a formal specification language, which can specify a system more accurately, consistently, and completely. The second step of formal development is developing the system from specification to executable code. A refinement method is used to develop the system from the specification to the code.

The object-oriented paradigm is another important method in software engineering. It groups together data structures and the operations performed on them, encapsulates them behind a clean interface, and organizes the entities in a hierarchy based on inheritance. Initially, object-oriented methods were applied primarily during the implementation phase using object-oriented languages. C++, Smalltalk, CLOS, and Eiffel are some of the widely known object-oriented languages. Some researchers have tried to combine the object-oriented paradigm and formal specification languages. Well-known existing object-oriented extensions to VDM are Fresco, QO VDM, and VDM++. We described these languages and discussed the strengths and weaknesses. To solve the problems of existing object-oriented VDMs, we created Object-VDM,
an object-oriented extension to VDM. To develop a system formally from Object-VDM to Eiffel, we used a modified refinement method. There are three stages in this refinement: data refinement, operation refinement, and structure refinement. In data refinement, the mathematical data models in object-VDM such as SET, SEQUENCE, and MAP are converted to Eiffel data structures. We created Eiffel libraries to do this. We also proved that this conversion is correct. In operation refinement, we modified and added some rules to the original refinement to obtain Eiffel code. Object-oriented features are converted in the structure refinement step. We presented a case study to show the refinement process.

6.2 SIGNIFICANCE OF THIS RESEARCH

The primary goal of this research was to extend the original refinement method in VDM to apply to object-oriented environments. The significance of this research is as follows:

- Our adaption of refinement methods in VDM to the object-oriented environment provides the first such extension. We modified and extended the original refinement method by adding structure refinement.

- Object-VDM is fully based on the VDM standard. Existing object-oriented extensions of VDM are not fully based on the VDM standard and do not fully support object-oriented facilities.

- Since the mathematical data models in Object-VDM such as SET, SEQUENCE, and MAP must be converted to Eiffel data structures, we created extensions to Eiffel libraries to systematically refine the Object-VDM data structures. We can automatically convert any three fundamental data structures in Object-VDM into Eiffel by using that Eiffel libraries. We also proved that the conversion is correct.
• We defined operation refinement rules for quantified predicates, thereby extending the original refinement method.

• the refinement method can be used more widely because the refinement methods as derived in this research apply to object-oriented environments.

6.3 Future research

Concurrency, automation, and generalization are topics for future research. Distributed or parallel programming is for concurrency. Formal specification languages and programming language for distributed systems have been studied extensively. Distributed formal languages use temporal logic [Pnu 86], CSP style [Hoa 85], or transition axioms [Lam 83] [Lam 89]. One of these methods could be applied to object-oriented VDM. There exist Eiffel extensions for distributed systems. Two widely known distributed extensions to Eiffel are Eiffel// [Car 89] [Car 93] and Distributed Eiffel [GL 92]. But these languages do not have all possible message passing facilities. Other proposed approaches for concurrent Eiffel languages are found in[Car 93][KB 93][Loh 93][Mey 93].

When the refinement process can be clearly described, semi-automation is possible. If the entire refinement process is automated, we can greatly reduce the effort for the design phase and implementation phase in software life cycle. Although complete automation may not be possible, partial automation reduces effort and time from specification to implementation. Automatic refinement with CASE tools is highly desirable.
BIBLIOGRAPHY


[Bac 78] Back, R. "On the correctness of refinement steps in program development" (Ph.D. thesis), University of Helsingki, 1978

[Bac 80] Back, R. "Correctness preserving program refinements: proof theory and applications" Mathematical Center Tracts 131, Mathematical Center, Amsterdam, 1980


[Cus 94] Cusack, E. “Inheritance In Object Oriented Z”, ECOOP’91, July 1991


[Loh 93] Lohr, K. "Concurrency annotations for reusable software", Commun. ACM 36, 9, Sept. 1993, pp. 81-89


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
118


[ Spi 88a] Spivey, J. "Introducing Z", Cambridge Univ. Press, 1988


[Wie 95] Wiener, R. "Software development using Eiffel: There can be other than C++", Prentice Hall, 1995


[YC 96] Yoo, M. and Carver D."Object-orientation in VDM", Proceedings of the 34th annual ACM conference (Southeast Region), Tuskegee, AL, April 1996
APPENDIX A.
LOW LEVEL SYNTAX OF OBJECT-VDM

- Expressions

expression list = expression, {',', expression};

def expression = bracketed expression
| let expression
| let be expression
| def expression
| if expression
| cases expression
| unary expression
| binary expression
| quantified expression
| iota expression
| set enumeration
| set comprehension
| set range expression
| sequence enumeration
| sequence comprehension
| subsequence
| map enumeration
| map comprehension
| tuple constructor
| record constructor
| record modifier
| apply
| field select
| function type instantiation
| lambda expression
| is expression
| name
| old name
| symbolic literal;

- Bracketed Expressions

  bracketed expression = '(' , expression , ')';

- Local Binding Expressions

  let expression = 'let', local definition, ' in', expression ;
let be expression = 'let', bind, ['be', 'st', expression], 'in', expression;

def expression = 'def', pattern bind, '=', expression, pattern bind, expression, 'in', expression;

- Conditional Expressions

  if expression = 'if', expression, 'then', expression,
  { elseif expression }, 'else', expression

  elseif expression = 'elseif', expression, 'then', expression;

cases expression = 'cases', expression, ':', cases expression alternatives

  [', others expression], 'end';

  cases expression alternatives = cases expression alternative,

  {', 'cases expression alternative};

  cases expression alternative = pattern list,'-', expression;

  others expression = 'others', '->', expression;

- Unary Expressions

  unary expression = prefix expression

  | map inverse expression;

  prefix expression = unary operator, expression;

  unary operator = unary plus

  | unary minus

  | arithmetic abs

  | floor

  | not

  | set cardinality

  | finite power set

  | distributed set union

  | distributed set intersection

  | sequence head

  | sequence tail

  | sequence length

  | sequence elements

  | sequence indices

  | distributed sequence concatenation.
map domain

map range

distributed map merge ;

unary plus = '+';

unary minus = '-';

arithmetic abs = ' abs'

floor = ' floor'

not = '¬';

set cardinality = ' card';

finite power set = ' F';

distributed set union = ' ∪';

distributed set intersection = ' ∩';

sequence head = ' hd';

sequence tail = ' tl';

sequence length = ' len';

sequence elements = ' elems';

sequence indices = ' inds';

distributed sequence concatenation = ' conc';

map domain = ' dom';

map range = ' rng';

distributed map merge = ' merge';

map inverse expression = expression, '¬1';
- Binary Expressions

binary expression = expression, binary operator, expression;

binary operator = arithmetic plus
| arithmetic minus
| arithmetic multiplication
| arithmetic divide
| arithmetic integer division
| arithmetic rem
| arithmetic mod
| less than
| less than or equal
| greater than
| greater than or equal
| equal
| not equal
| or
| and
| imply
| logical equivalence
| in set
| not in set
| subset
| proper subset
| set union
| set difference
| set intersection
| sequence concatenate
| map or sequence modify
| map merge
| map domain restrict to
| map domain restrict by
| map range restrict to
| map range restrict by
| composition
| iterate;

arithmetic plus = '+';
arithmetic minus = '-';
arithmetic multiplication = '×'
arithmetic divide = '/'

arithmetic integer division = ' div'

arithmetic rem = ' rem'

arithmetic mod = ' mod'

less than = '<'

less than or equal = '<='

greater than = '>'

greater than or equal = '>='

equal = '='

not equal = '!='

or = 'V'

and = '\wedge'

imply = ' \Rightarrow'

logical equivalence = ' \Leftrightarrow'

in set = ' \in'

not in set = ' \not \in'

subset = ' \subseteq'

proper subset = ' \subset'

set union = ' \cup'

set difference = ' \setminus'

set intersection = ' \cap'
sequence concatenate = '⊔';
map or sequence modify = '†';
map merge = '∪';
map domain restrict to = '<';
map domain restrict by = '<';
map range restrict to = '>';
map range restrict by = '>';
composition = 'o';
iterate = '↑';

The ↑ infix operator can be replaced by a superscript: m ↑ n can be written as \( m^n \).

- Quantified Expressions
  
  quantified expression = all expression
  | exists expression
  | exists unique expression;

  all expression = '∀', bind list, '.', expression ;

  exists expression = '∃', bind list, '.', expression;

  exists unique expression = '∃!', bind, '.', expression;

- Iota Expression
  
  iota expression = 'i', bind, '.', expression;

- Set Expressions
  
  set enumeration = '{', [expression list], '}';

  set comprehension = '{', expression, ',', bind list, [',', expression], '}';

  set range expression = '{', expression, ',', '...', ',', expression, '}';
Sequence Expressions

sequence enumeration = ‘[’, [ expression list ],’]’;

sequence comprehension = ‘[’, expression, ‘|’, set bind, [‘,’ , expression, ‘]’);

subsequence = expression, ‘(’, expression, ‘;’ , ‘...’ , ’;’, expression, ‘)’;

Map Expressions

map enumeration = ‘{’, maplet, { ‘,’ , maplet },’}’ ‘{’, ‘→’, ‘}’;

maplet = expression, ‘→’, expression

map comprehension = ‘{’, maplet, ‘|’, bind }=list, [‘,’ , expression];

Tuple Constructor Expression

tuple constructor = ‘mk’, ‘(’, expr, ‘,’ , expression list, ‘)’;

Record Expressions

record constructor = name, ‘(’, [ expression list ],’);’;

record modifier = ‘μ’, expression, ‘,’ , record modification, {‘,’ , record modification },’);’;

record modification = identifier, ‘→’, expression

Apply Expressions

apply = expression, ‘(’, [ expression list],’);’;

field select = expression, ‘,’ , identifier

function type instantiation = name, ‘[’, type, type },’]’;

Lambda Expression

lambda expression = ‘λ’, type bind list, ‘.’ , expression;

Is Expressions

is expression = identifier, ‘(’, expression, ‘)’,
| is basic type, ‘(’, expression, ‘)’;

Names

name = identifier;

name list = name, {‘,’ , name };

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
old name = identifier, '-'

An old name such as identifier can also be written as 'identifier'.

- State Designators

state designator = name

  | field reference
  | map or sequence reference;

field reference = state designator, '.', identifier;

map or sequence reference = state designator, '{', expression, '}'

- Statements

statement = let statement

  | let be statement
  | def statement
  | block statement
  | assign statement
  | if statement
  | cases statement
  | sequence for loop
  | set for loop
  | index for loop
  | while loop
  | nondeterministic statement
  | call statement
  | return statement
  | always statement
  | trap statement
  | recursive trap statement
  | exit statement
  | identity statement

- Local Binding Statements

let statement = 'let', local definition, local definition 1, 'in', statement;

  local definition = value definition | function definition;

let be statement = 'let', bind, ['be', 'st', expression], 'in', statement;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
def statement = ' def', equals definition, { ';', equals definition },
    ' in', statement;

equals definition = pattern bind, ' = ', expression
    | pattern bind, ' = ', call statement;

- Block and Assignment Statements

block statement = '(', { dcl statement }, statement, { ';', statement }, ');

dcl statement = ' dcl', assignment definition, ';

assignment definition = identifier, ':', type, [':=', expression]
    | identifier, ':', type, ['=', call statement];

assign statement = state designator, ' := ', expression
    | state designator, ' := ', call statement;

- Conditional Statements

if statement = ' if', expression, ' then', statement,
    { elseif statement }, ' else', statement;

elseif statement = ' elseif', expression, ' then', statement;

cases statement = ' cases', expression, ':', cases statement alternatives,
    [',', others statement ], ' end';

cases statement alternatives = cases statement alternative,
    {',', cases statement alternative };

cases statement alternative = pattern list, ' — > ', statement;

others statement = ' others', ' — > ', statement;

- Loop Statements

sequence for loop = ' for', pattern bind, ' in', [' reverse'], expression,
    ' do', statement;

set for loop = ' for', ' all', pattern, 'E', expression, ' do', statement;

index for loop = ' for', identifier, expression, ' to', expression,
    [' by', expression], ' do', statement;

while loop = ' while', expression, ' do', statement.
- NonDeterministic Statement
  nondeterministic statement = '||', '(', statement, '{', '}', statement, ')';

- Call and Return Statements
  call statement = name, '(', [expression list], ')',
  [using state designator];
  return statement = 'return', [expression ];

- Exception Handling Statements
  always statement = 'always', statement, 'in', statement;
  trap statement = 'trap', pattern bind, 'with', statement,
  'in', statement;
  recursive trap statement = 'tixe', traps, 'in', statement;
  traps = '{', pattern bind, '→', statement, '{', pattern bind,
  statement, '}';
  exit statement = 'exit', [expression ];

- Identity Statement
  identity statement = 'skip'

- Patterns and Bindings
  - Patterns
    pattern = pattern identifier
    | match value
    | set enum pattern
    | set union pattern
    | seq enum pattern
    | seq conc pattern
    | tuple pattern
    | record pattern

    pattern identifier = identifier | '.';

    match value = '(', expression, ')', | symbolic literal

    set enum pattern = '{', pattern list, '}';

    set union pattern = pattern, 'U', pattern;
seq enum pattern = ' [', pattern list, ' ]';
seq conc pattern = pattern, ' ∙ ', pattern;
tuple pattern = ' mk', '(' pattern, ',', pattern, ')';
record pattern = name, ' (', [ pattern list ], ')';
pattern list = pattern, ' ; ', pattern;

- Bindings
  pattern bind = pattern | bind;
  bind = set bind | type bind;
  set bind = pattern, ' ∈ ', expression;
  type bind = pattern, ' ; ', type;
  bind list = multiple bind, { ' ; ', multiple bind };
  multiple bind = multiple set bind | multiple type bind;
  multiple set bind = pattern list, ' ∈ ', expression;
  multiple type bind = pattern list, ' : ', type;
  type bind list = type bind, { ' ; ', type bind };

- Symbols
  keyword = ' as' | ' abs' | ' always' | ' be' | ' bool' | ' by' | ' card' | ' cases' | ' char' | ' comp' |
            ' compose' | ' conc' | ' dcl' | ' def' | ' definitions' | ' dinter' | ' div' | ' dmodule' |
            ' do' | ' dom' | ' dunion' | ' elems' | ' else' | ' elseif' | ' end' | ' error' | ' err' |
            ' exists' | ' exists1' | ' exit' | ' exports' | ' ext' | ' false' | ' floor' | ' for' |
            ' forall' | ' from' | ' functions' | ' hd' | ' if' | ' imports' | ' in' | ' inds' | ' init' |
            ' inmap' | ' instantiation' | ' int' | ' inter' | ' inv' | ' inverse' | ' iota' | ' lambda' |
            ' len' | ' let' | ' map' | ' merge' | ' mod' | ' module' | ' munion' | ' mu' | ' nat' |
            ' nat1' | ' nil' | ' not' | ' of' | ' operations' | ' or' | ' others' | ' parameters' |
            ' post' | ' power' | ' pre' | ' psubset' | ' rat' | ' rd' | ' real' | ' rem' | ' return' |
            ' reverse' | ' rng' | ' seq' | ' seq1' | ' set' | ' skip' | ' st' | ' state' | ' subset'
separator = newline | white space

identifier = ( plain letter | Greek letter ) , { ( plain letter | Greek letter )
            | digit | "" | "."


type variable identifier = 'a', identifier

is basic type = 'is', ( 'bool' | 'nat' | 'nat1' | 'int' | 'rat' | 'real' | 'char' | 'token')

symbolic literal = numeric literal | boolean literal | nil literal | character literal
                   | text literal | quote literal

numeral = digit , { digit }

numeric literal = numeral , [ ".", digit , { digit }] , [ exponent ]

exponent = '× 10 † , [ "+" | "-" ] , numeral

boolean literal = 'true' | 'false'

nil literal = 'nil'

character literal = "", character - newline - multi character , ""

multi character = Greek letter | '<=' | '<' | '=>' | '<>' | '<->' | '+' | '<=' | '==' | '=>'
                  | '||' | '<=>' | '<=' | '<>' | '<' | '< : >' | '< : >' | '<<>' | '<-->' |
                  | '< : >' | '< =>' | '<' | '<-->' | '< =>' | '<-->' | '<-->' | '<-->


text literal = "", { "" } , character - ( "" | newline ) , ""

quote literal = distinguished letter , { "" | distinguished letter | digit }

comment = "--", { character - newline }, newline

• Characters

character = plain letter | key word letter | distinguished letter | Greek letter
           | digit | delimiter character | other character | separator
plain letter

a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

key word letter

a b c d e f g h i j k l m
n o p q r s t u v w x y z

delimiter character

, : ; = ( ) | - [ ]
{ } + / < > <= >= <>
* - > + > == > || => <= > |-> < : >

other character

_ ' , " @ -

digit

0 1 2 3 4 5 6 7 8 9
distinguished letter

The distinguished letters use the corresponding capital and lower-case letters where the whole quote literal is preceded by "<" and followed by ">") (note that quote literals also can use underscores).

Greek letter

The Greek letters can also be used with a number sign "#" followed by the corresponding letter
APPENDIX B.
EIFFEL LIBRARIES FOR VDM-SET, VDM-SEQUENCE AND VDM-MAP

class VDM_SET[T]
export
  isempty, union, intersection, isequal, isnotequal, isdisjoint, issubset, issubsetproper, iselmt, isnotelmt, cardinality, addelmt, subtractelmt, difference, distunion, distintersect
inherirt LINKED_LIST[T]

feature

  isempty():boolean is
    local temp:boolean
    do
      if count=0
        then Result:=true
        else Result:=false
      end
    end;

  isequal(other:like Current):boolean is
    local temp:boolean
    do
      if (IsSubset(other) and Other.IsSubset(Current))
        then Result:=true
        else Result:=false
      end
    end;

  isnotequal(other:like Current):boolean is
    local temp:boolean
    do
      if (IsSubset(other) and Other.IsSubset(Current))
        then Result:=false
        else Result:=true
      end
    end;

  cardinality:integer is
    do
      Result:=count
    end;

133
iselmt(v:item): boolean is
do
  if occurrence(v) /= 0
    then Result := true
    else Result := false
  end
end;

isnotelmt(v:item): boolean is
do
  if occurrence(v) = 0
    then Result := true
    else Result := false
  end
end;

addelmt(v:item): like Current is
do
  Result := extend(v)
end;

subtractelmt(v:item): like Current is
do
  Result := prune_all(v)
end;

isdisjoint(other: like Current): boolean is
  local temp: boolean
  do
    start
    temp := true
    from i := 1
    variant i
    until i = count
    loop
      if other.count(i_th(i)) \= 0
        then temp := false
      end
    end
    Result := temp
  end;
issubset(other:like Current):boolean is
  local temp:boolean
  do
    start
    from i=1
    variant i
    until i=count
    loop
      if others.count(i_th(i))=0
        then temp:=false
      end
    end
    Result:=temp
  end;

issubsetproper(v:item):boolean is
  do
    if (issubset(other) and (isNotEqual(other)))
      then Result:=true
    else Result:=false
    end
  end;

difference(other:Current) like Current is
  do
    start
    from i=1
    variant i
    until i=count
    loop
      if others.count(i_th(i))=0
        then Result.extend(i_th(i))
      end
    end
  end;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
intersect(other:like current) like current is
  do
    start;
    n:=count;
    Result:=duplicate(n);
    from i=1
    variant i
    until i=other.count
    loop
      if others.occurrence(i_th(i))=0
        then Result.extend(i_th(i))
      end
    end
  end;

union(other:like current) like current is
  do
    start;
    n:=count;
    Result:=duplicate(n);
    from i=1
    variant i
    until i=count
    loop
      if others.occurrence(i_th(i))=0
        then Result.extend(i_th(i))
      end
    end
  end;

distunion: like Current is
  do
    start
    Result=empty
    from i=1
    variant i
    until i=count
    loop
      Result=Result.union(i_th(i))
    end
  end;
distintersect: like Current is
do
start
Result=empty
from i=1
variant i
until i=count
loop
   Result=Result.intersection(i_th(i))
end
end

end -- class VDM_SET
class VDM_SEQUENCE[T]
export
    head, tail, length, elms, inds, conc
inherit LINKED_LIST[T]
feature
    head:T is
    do
        Result:=first
    end;

tail:like Current is
    do
        from i=1
        variant i
        until i=count-1
        loop
            Result.i_th(i)=i_th(i+1)
        end
    end;

length:integer is
    do
        Result:=count
    end;

elms:VDM_SET is
    do
        Result:=empty
        from i=1
        variant i
        until i=count
        loop
            Result.addElmt(i_th(i))
        end
    end;
indices: VDM_SET is
do
  Result:=empty
  from i=1
  variant i
  until i=count
  loop
    Result.addElmt(i)
  end
end;

conc(other:like Current):like Current is
do
  finish
  merge_right(other)
end

end -- class VDM_SEQUENCE
class PAIR[T1,T2]
feature
    front:T1
    back:T2

    first:T1 is
    do
        Result:=front
    end;

    second:T2 is
    do
        Result:=back
    end
end
class VDM\_MAP[T1,T2] export
domain, range, inverse, merge, override,
domainrestrict, domainExcl, rangerestrict, rangeexcl,
composite, isequal, isnotequal
inherit LINKED\_LIST[PAIR[T1,T2]]
feature
  domain:VDM\_SET[T1] is
do
    Result:=empty
    from i=1
    variant i
    until i=count
    loop
      Result.addElmt(i\_th(i).first)
    end
  end;

range:VDM\_SET[T2] is
do
  Result:=empty
  from i=1
  variant i
  until i=count
  loop
    Result.addElmt(i\_th(i).second)
  end
end;

inverse: MAP[T2,T1] is
do
  from i=1
  variant i
  until i=count
  loop
    Result.i\_th(i).first=i\_th(i).second;
    Result.i\_th(i).second=i\_th(i).first
  end
end;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
override(M2:like Current): like Current is
do
  from i=1
  variant i
  until i=count
  loop
    k=(other.dom.index_of(i_th(i).first,1))
    if k=0 then
      Result.i_th(i).first=i_th(i).first;
      Result.i_th(i).second=i_th(i).second
    else
      Result.i_th(i).first=other.i_th(i).first;
      Result.i_th(i).second=other.i_th(i).second
    end
  end
end;

DomainRestrict(S:VDM_SET): like Current is
do
  from i=1
  variant i
  until i=count
  loop
    if S.IsElmt(i_th(i).first)
      then Result.put_right(i_th(i).first);
      Result.put_right(i_th(i).second)
    end
  end
end;

DomainExcl(S:VDM_SET): like Current is
do
  from i=1
  variant i
  until i=count
  loop
    if S.IsNotElmt(i_th(i).first)
      then Result.put_right(i_th(i).first);
      Result.put_right(i_th(i).second)
    end
  end
end.
RangeRestrict(S:VDM_SET): like Current is
  do
    from i=1
    variant i
    until i=count
    loop
      if S.IsElmt(i_th(i).second)
        then Result.put_right(i_th(i).first)
      end
      Result.put_right(i_th(i).second)
    end
  end;

RangeExcl(S:VDM_SET): like Current is
  do
    from i=1
    variant i
    until i=count
    loop
      if S.IsElmt(i_th(i).second)
        then Result.put_right(i_th(i).first);
        Result.put_right(i_th(i).second)
      end
    end
  end;

composite(other:MAP[T3,T1]): MAP[T3,T2] is
  do
    from i=1
    variant i
    until i=count
    loop
      k=(other.ran.index_of(i_th(i).first,1));
      Result.put_right(other.i_th(i).first);
      Result.put_right(i_th(i).second)
    end
  end;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
IsEqual(other:like Current): boolean is
  local temp:boolean
  do
    temp=true
    from i=1
    variant i
    until i=count or temp=false
    loop
      k=other.dom.index_of(i_th(i).first,1)
      if k=0 then temp=false
      elseif (other.i_th(k)).second \= i_th(i).first
        then temp=false
      end
    end
  end

isnotequal(other:like Current): boolean is
  local temp:boolean
  do
    if isequal(other)
      then Result:=false
    else Result:=true
    end
  end

end -- class VDM_MAP
VITA

Moonsung Yoo received the bachelor of science degree in Mathematics from Seoul National University in Korea. He worked for Hyundai Construction Company and Korea Development Institute (KDI) for several years as a programmer and systems analyst. He received the master of science degree in Computer Science from Indiana University at Bloomington in 1991. He has been a doctoral student in Computer Science at Louisiana State University since January 1992 and will be awarded his doctor of philosophy degree in December of 1996. His current research interests are formal software development methods and object-oriented paradigm.
Candidate: Moonsung Yoo

Major Field: Computer Science

Title of Dissertation: From Object-Oriented Specification to Implementation: A Formal Refinement Methodology

Approved:

[Signatures]

Major Professor and Chairman
Dean of the Graduate School

EXAMINING COMMITTEE:

[Signatures]

Date of Examination: October 3, 1996