National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

VDM/C++ : A Design and Implementation Framework

Youchen Lou

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

February 1994

ISBN 0-315-90854-8

Canada

ABSTRACT

VDM/C++ : A Design and Implementation Framework

Youchen Lou

This thesis presents a framework for deriving C++ implementations from VDM specifications. The transformation is completed in two steps; from VDM specification to an object-oriented design and from this design to C++ classes. The object-oriented intermediate model used in the development process is language independent and so can be applied to any object-oriented language. Only partial automation is possible in the transformation process. So, an interactive interface is built to support user interaction and provide information available at all stages to assist an integrated software development.

VDM/C++ has been implemented under UNIX system using Motif toolkit and C++ programming language. The result shows that methodologies discussed in this thesis are both feasible and practical. VDM/C++ provides an useful environment supporting the development of software systems from specifications to implementation.

VDM/C++ framework introduced in this thesis can cooperate with the Mural system, a knowledge-based support and tools which support the transformation from a semi-formal description to a VDM specification.

Acknowledgments

I would like to thank my supervisor Professor V.S. Alagar for active discussions and valuable suggestions on the subject of this thesis. This work would not have been possible without his consistent guidance and encouragement. I thank Dr. Alagar for the financial support given to me during my study at Concordia University.

I am grateful to my husband Zhaogong for his hours of beneficial discussions and his patience and moral support.

I wish to express my thanks to all those people who have helped me in the production of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

VDM/C++ environment presented in this thesis is an interactive knowledge assisted environment supporting the development of software components from VDM specifications. Our goal is to develop a practical environment in which software productivity, quality and reliability can all be ensured.

Developing and maintaining large software systems is notoriously difficult and expensive. One source of difficulty is that such projects involve large amounts of disparate knowledge about the application domain, the developing environment, the software architecture, the technical personnel and the resources etc. A critical problem encountered in such situations is that a great deal of relevant knowledge is not usually documented and remains accessible only through human experts. To completely document all knowledge mentioned above is inadequate for three reasons. The first is the *acquisition* and *representation problem*: the amount of such knowledge is so large that capturing it is tedious and time-consuming. It is also unclear how to organize or index the knowledge. The second is the *access problem*: without adequate indexing, the resulting information base is simply too large to be very useful (busy people, including software developers, will not read large documents that are not immediately relevant to their current task). Finally, there is the *maintenance problem*: the existing knowledge will change over time; as a result more detailed knowledge is required as the development of software system. A knowledge-base must be maintained and ensured that it remains useful. We believe that by retaining human interaction in the software development process, many of the problems encountered in the complete

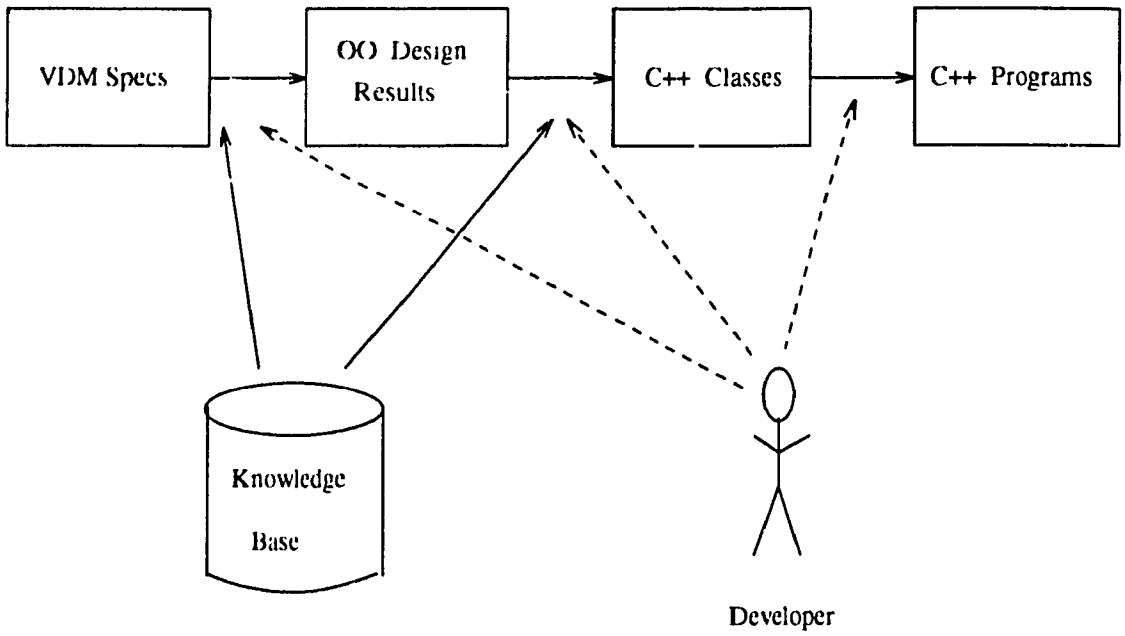


Figure 1.1: The VDM/C++ Environment

automation of software system may be avoided. Figure 1.1 shows how the development process in VDM/C++ can incorporate human interaction and knowledge-based assistance.

Requirements for nontrivial programs can become very complex to state and unmanageable to maintain. Their consistency and adequacy are difficult to determine when stated informally. Formal methods based on mathematical techniques for describing system properties offer a possible solution. The use of formal methods is aimed at controlling the enormous and ever-growing complexity in the development of large software systems.

There are many potential advantages to use formal methods. They allow rigor and precision including unambiguous communication, prediction, evaluation and better understanding of and control over software products and the software development process. Furthermore, the development of a formal foundation of software engineering has the potential to provide criteria for evaluation, a means of comparison, information about theoretical limits and capabilities, and a means of describing and studying underlying rules and structures of software design. Not all of these potential benefits

have yet been realized. The most developed aspect of formal methods are formally-defined specification language.

A formal specification is a language with explicitly and precisely defined syntax and semantics. One virtue of formal specifications is their precision. Precision leaves no room for ambiguity. The process of writing formal specifications can often reveal contradictions, ambiguities and incompleteness in a problem requirement and errors in a program's design. Uncovering bugs early can thus save the cost, compared with uncovering them later during testing and debugging. Precision also implies that we can formally argue the correctness of programs. We believe that using formal specifications at the early stage in the software development process can be especially beneficial. Another virtue of formal specifications is their amenability to machine manipulation. With help from appropriate tools(e.g., theorem provers), we can handle nontrivial specifications, and thus formally reason about specifications and programs which are much lager than these if we had to rely only on pencil and paper.

There is still much to be done before formal methods will be widely used in software development. We need good formal methods and also ways of interfacing them to human abilities for easy use of formal methods. Furthermore, tools which support the development of formal methods should be developed.

Alagar and Periyasamy [2] presented a methodology for deriving an object-oriented design from model-based specifications. The transformations discussed can be formalized and applied to any model-oriented specification to get a corresponding object-oriented design. A tool called GAP is built by Gao [19] which applies the methodology to VDM specification. The development of GAP shows that the methodology is both feasible and practical. This thesis develops the further work based on GAP [19].

Object-oriented paradigm is a relatively new development in software engineering. Compared to the traditional functional design, object-oriented design aims for more robust software that can be easily reused, modified, maintained and extended. The greatest strength of an object-oriented approach to development is that it offers a mechanism that captures a model of the real world. This leads to improved maintainability and understandability of systems whose complexity exceeds the intellectual

capacity of a single developer or a team of developers.

However, the design process for the object-oriented paradigm remains ad hoc. Several efforts have been made to improve the situation. Alabiso [3] introduced the transformation of data flow analysis model to object-oriented design. Ward [37] argued that there is no conflict between structured design and object-oriented design and illustrated how to integrate object-orientation with structured analysis and design. Beck and Cunningham [5] suggested the use of index cards to record initial class designs. Wirfs-Brock, Wilkerson and Wiener [40] provided a coherent model for the design process : responsibility-driven design, in which the index card, hierarchy graph and collaboration graph are used. Rumbaugh, Blaha and Premerlani [34] discussed the analysis, design and implementation, the entire development life cycle, using their own graphical notation and methodology.

In the VDM/C++ environment, an object-oriented design is derived from requirements specification written in VDM. The derived design is language independent which can in principle be implemented later using any object-oriented language. At present, C++ has been chosen as the target language for implementation. Consequently, we create C++ classes and member functions from the object-oriented design derived from the VDM specification of requirements. An interactive user interface is provided to help the developer to comprehend all information available at the requirement specification through implementation levels so that it is possible to derive an implementation faithful to its specification. Such knowledge-based tools may greatly simplify the development process and, above all, the maintainability of the system. Thus, VDM/C++ is the means of achieving a desired end; it provides a specific choice - from VDM through OOD to C++.

Henderson and Edwards [14] argued that it is possible to derive object-oriented designs from both functional specification and object-oriented specification of requirements. However, we claim that the initial requirements will always be functional because users generally describe what the system is to perform; it is hard for the users to describe what objects are involved in the system design and/or implementation and how they are supposed to interact with each other. Moreover, the specifications

of requirements should not bias the design process. For example, formal description in VDM uses mathematical notation to model system components and to describe the functionalities of the system. These specifications naturally lead to function-oriented design: see for example Jones [23].

The significance of our approach lies in the development of an object-oriented design from VDM specification. It builds a bridge between formal specifications (in functional style) and object-oriented paradigm and enjoys the advantages of both of them. The modelling methods built for a function-oriented approach can also be used for deriving an object-oriented design. Furthermore, the major problem for object-oriented design is that existing methods mentioned above are all informal and consequently critical studies in the development process can not be formally verified. In VDM/C++ environment, the verification of VDM specification and the transformation methodology assures the correctness of the design.

The organization of the thesis is as follows. Chapter 1 presents the motivation and significance of the study. Chapter 2 introduces the object-oriented paradigm and discusses the reason for choosing C++ as the implementation language. Some related work is discussed and compared to VDM/C++. Chapter 3 is a brief introduction to VDM specification. It is not a complete description of the VDM specification language but explains the basic notations which are sufficient to understand the specifications and transformation method included in this thesis. System development refinement using VDM is also discussed. Chapter 4 introduces the methodology for deriving an object-oriented design from VDM specification. It also briefly describes a tool in the environment, GAP, which supports direct and automatic transformation from VDM specification of software system into object-oriented design. Chapter 5 is a detailed description of methodology for transforming object-oriented design to C++ header file. Some implementation details are also provided. One example, the banking system, is presented in Chapter 3, Chapter 4 and Chapter 5 to explain the reification process in VDM, the corresponding designs, implementations and their relationships. Chapter 6 illustrates the user interface for the system. Both design consideration and implementation issues are discussed. A complete VDM specification for the user

interface is also provided. And finally Chapter 7 concludes the study and suggests further development directions.

Chapter 2

Formal Approach to OOD and Scope of This Thesis

We present the object-oriented design(OOD) model in this chapter. This is necessary because of the varying concepts and definitions of object-orientation adopted by different practitioners and researchers. Also we describe how the concepts are used and implemented in some popular programming languages, such as C++, Smalltalk, Eiffel and Ada. The reason why we have chosen C++ in VDM/C++ environment is stated. Finally, some work related to the integrating formal methods with object-oriented paradigm is introduced and compared to our work.

2.1 Object-oriented Design Paradigm

Multiple views and definitions for object-orientation are being followed by different practitioners and researchers. We do not intend to argue with the validity of such differing views and concepts; rather, we wish to concretize our notion of object-orientation and use these concepts for the derivation of object-oriented design from the specification. The object-oriented model defined below includes concepts that do not differ much from those available in most popular object-oriented languages such as C++, Eiffel and Smalltalk.

Due to space limitations, we do not give a complete model in this thesis; rather, we provide a list of important concepts and our definitions for those concepts which will be necessary in understanding the derivation process.

Object An object-oriented system consists of a collection of objects, which are the representatives of the real-world entities. The entire system is developed to represent these objects and their interactions.

Class Objects exhibiting similar properties are grouped together into a *class*. Stated otherwise, a class is an abstract collection of objects and an object is a concrete instantiation of that class.

Data member Each object is characterized by a set of variables called *data members*. This is a C++ terminology; Eiffel uses the term *attributes* to denote the structural components of an object. The state of an object is represented by the collection of values associated with its data members. The type of a data member may be a primitive type such as *integer* or *real* or it could be a class in which case the data member is an object by itself. By this way, composite objects can be built from primitive objects. This process is known as *aggregation* which will be defined shortly.

Member Function Each data member of an object can be changed only by a function associated with the object. Consequently, each object has a set of functions called *member functions* defining the behavior of the object. In some object-oriented languages such as C++, the strict condition of accessing the data members only through member functions is not incorporated. Consequently, data members can be accessed directly.

Exported features and Communication Since each data member of an object can be accessed only through its own member functions, the state of an object can be changed only when there is an invocation of one of its member functions. This will happen only when some other object requests for such a service. Such requests are called *messages* and consequently, the object-oriented system provides effective message passing communication between the objects. The interaction between the various objects provides the execution environment of the system.

Each object will provide only a subset of its data members and member functions to be accessible by other objects. These data members and member functions form the interface of the object and are called *exported features* of the object. The interface member functions will access the other member functions which may or may not be in the set of exported features. Thus, it is possible for the object to change its internal behavior without changing its interface. This technique, called *information hiding*, is one of the important principles of object-orientation.

Inheritance Often, it is desirable to predefine relationships among objects so that the interaction of objects between two different classes will be normalized. Inheritance is one such possible relationship in which one class (and hence objects of this class) is made as a specialization of another class (and consequently the objects of the second class). For example, a class *rectangle* can be made as a specialization of a class *polygon*. By inheriting from polygon, the class rectangle acquires all the exported features of polygon and hence need not define them again.

The definition of inheritance differs considerably with respect to several object-oriented techniques and hence we concretely define it here: If class A inherits class B, then

- The features (data members and member functions) of class A become a superset of the features of the class B.
- Class A can add, delete (or restrict) or modify some of the features of class B but, within A itself. This will not affect the behavior of B. Strictly speaking, A must differ from B; otherwise, there is no need for such inheritance.
- Class A should not re-export the features already exported from class B; otherwise confusions and inconsistencies might arise.

Aggregation Also called *part-of* in our model, the aggregation relation is the means

of building composite classes (and hence composite objects instantiated from these classes) from other classes. In our model, class B is part of class A if and only if

- Class B is strictly a component of class A; i.e., there is a data member in A whose type is B. This indicates a strict relationship between A and B in which A cannot survive without B. Stated otherwise, every instance of A relies on an instance of B.
- Classes A and B must be different; otherwise, there is no use of such part-of relationship.
- Class A cannot alter the behavior of the instance of class B inside A. This is the major difference between inheritance and aggregation.

Polymorphism Polymorphism is the ability by which a method can be executed in more than one way, depending on the arguments and the receiver.

2.2 Choice of an Object-oriented Language

In the previous section, we have introduced the essential concepts in object-orientation. An object-oriented language should, at least, support the following[22]:

- Encapsulated objects
- The class and instance concepts
- Inheritance between classes
- Polymorphism

In this section, we discuss how and to what extent these concepts are supported by different languages such as C++, Smalltalk, Eiffel and Ada etc. We will not provide a detailed description of any specific language. Further readings about the languages that we discussed here can be obtained for C++[26, 16]; Smalltalk[20, 28

29]; Eiffel[31]; Ada[4, 6]. Peter Wegner [38] has made a generally accepted classification of different programming languages in the object-based world. He states in this classification that Ada is object-based because it supports the object concept, but not the class concept. C++, Smalltalk, Eiffel, Simula[18] and Objective-C[9] are examples of object-oriented languages since they also support the class and inheritance concept.

Object The most important concept that an object-oriented language must support is the object concept. Thus the language must support the definition of a set of operations for the object, namely the object's interface, and an implementation part for the object(which a user of the object should not know about). The object implementation is thus encapsulated and hidden from the user.

An object is implemented internally as a number of variables which store information and a number of operations which can be performed on those variables. In Smalltalk, the only way to affect the internal variables is to perform an operation on the object. In C++ or Eiffel, it is possible to define whether the user should be able to directly access these variables. Moreover, C++ provides the mechanism called *private*, *protected* and *public* to allow more flexible access to the data members. A *private* member can be accessed only by members of the class. Members that occur in the *protected* section can be accessed by members in the class and in its inherited classes, and a *public* member is accessible from anywhere in the program.

Class and instances In object-oriented language, each object is described by a *class* which is both a module for the source code and a type for the *instances* of the class. The programming language Ada comes close to this approach with the *package* concept, but the package in Ada is not a type. Inside a package, however, types can be defined with associated operations. These types can in turn be used to create objects, where the package's interface may specify the operations that can be performed on the object.

A class defines the operations that can be performed on an instance. It also defines the variables of the instance. A variable associated with a specific instance is often called an *instance variable*. These instance variables store the instance's state. The reason for calling them instance variables, and not just variables, is due to the fact that some languages, such as Smalltalk and, in some senses, C++, have other variables which are only associated with the class. These are called *class variables*. Additionally, variables could also be local to a specific operation. These are called *temporary variables*. Several programming languages do not provide the programmer with the possibility to declare and use class variables. This is the case in Eiffel, where the distinction between the class and the instance is carefully made. The class is viewed as a description, or the program text, while instances are viewed as executions of the program text. Thus instances are the only thing that exists during run-time, and not classes. In C++, the programmer hasn't really the possibility to declare class variables, but can declare an instance variable as 'static', which gives all instances the same value for this variable. In such a way, it can be used as a class variable.

Some object-oriented programming languages also view classes as objects, that is as instances of another class. This is the case in Smalltalk, where each class is an instance of a *meta-class* . An advantage with this is that it is possible to send a message to a class and thus affect all its instances. Languages such as C++ and Eiffel have chosen not to consider classes as object. In this case, the class is regarded as implementing a type, while the object is regarded as an instance of this type. The class is static and is described in the program text, while the instance is dynamic and exists only during execution. Thus, the distinction between description and corresponding execution is made fundamental. The advantage of this approach is that there is a clear distinction between class and instance.

Normally, operations are only performed on instances, but some languages allow operations to be performed directly on the class. Class operations are normally

used to operate on class variables or to create new instances. In Smalltalk, class operations are common, while in C++ and Eiffel, they have been avoided. However, every programming language has operations for creating new instances. This can be considered as a class operation as it operates on the class.

Inheritance Inheritance means that we can develop a new class merely by stating how it differs from another, already existing class. The new class then inherits the existing class. The main advantage with this approach is that the existing class can be reused to a great extent. It is not only the classes which have been designed for the current system that can be reused, but also those designed earlier. Smalltalk, C++ and Eiffel are all delivered with an extensive class library, as such, the programming is based to a large extent on reusing these classes.

The number of inheritance hierarchies in a system varies between different languages. In Smalltalk, there is only one inheritance hierarchy, and thus one root class. Behavior, which is common to all system classes, is collected and stored in a root class which is called *Object* in Smalltalk. In this way, the programmer is forced to work with the inheritance hierarchy. In C++ and Eiffel, there can exist several parallel inheritance hierarchies. One can thus create different hierarchies for different structures in the system. It is therefore possible to create new classes without using the existing one. The experienced programmer can thus make use of the inheritance mechanism to a great extent.

Polymorphism Polymorphism as used in object-oriented paradigm means that the client class which sends a message does not need to know the class of the receiving instance. The client class provides only a request for a specified event, while the receiver knows how to perform this event. The polymorphism characteristic sometimes makes it uncertain at the compiler time, to determine which class an instance belongs to and thus to decide which operation to perform. *Dynamic binding* by which the binding of a message and the actual operation is delayed until the run-time is therefore a way of implementing the polymorphism

characteristic.

Smalltalk is often referred to as being a non-typed language. In Smalltalk, polymorphism is normally not restricted through the use of an inheritance hierarchy, as is often the case with strongly typed languages. In Smalltalk, the referenced instance can be associated to any class in the system. In strongly typed languages, such as C++, Eiffel and Simula, each reference to an instance has a type that specifies the classes to which the reference can refer. But it is unnecessary to specify exactly with which class the receiving instance is associated. Typically we only specify that the instance shall be associated with class A or some of class A's descendants. The operation can be defined in a descendant and we thus cannot bind before execution. C++ implements *limited polymorphism*: all classes within a hierarchy can respond to the same message, but classes outside the hierarchy cannot. In C++, this is compensated by allowing multiple inheritance. We can simply create an abstract superclass that defines the message we want to share and have other classes inherit from it. So, there is no restriction to the use of polymorphism in C++.

Table 2.1 summarizes the different features of languages discussed in this section. The reasons for choosing C++ as the target implementation language in VDM/C++ environment are as follows:

- C++ supports all object-oriented concepts.
- C++ provides flexible way to use these concepts.
- C++ is one of the most popular object-oriented language in the marketplace.

2.3 Related Work

The approach discussed in the thesis is unique in itself. To our knowledge, no other system exists with which this work can be compared. However, some work reported in the literature shares similar concerns in the integration of formal methods in the

	C++	Smalltalk	Eiffel	Ada
Object	yes	yes	yes	yes
Define direct access to internal variables	yes	no	yes	
Class	yes	yes	yes	no
Class variables	static	yes	no	-
Class operation	no	yes	no	
Inheritance	yes	yes	yes	
Polymorphism	yes	yes	yes	

Table 2.1: Comparison of different languages

software development process. All these approaches are concerned with formal specification and object-oriented design but differ among themselves, particularly, with the level at which the formal methods are employed.

Fresco [39] is a Smalltalk-based interactive environment supporting the specification, implementation and proofs. A basic unit in Fresco is called a *capsule* which contains specification, proofs and code. System components are built by putting several capsules together.

Larch/C++ [25] is an interface specification language which applies Larch specifications to C++ programs. Hence the specifications are tied to C++ abstractions. Modules that use common C++ constructs are specified using Larch. Thus, an attempt is made in Larch/C++ to minimize the gap between specification and implementation.

VDM/Ada [33] examines VDM development with Ada as the target language. The development is supported by a semi-automatic tool based on a "rule set" for the process. The author argues that the translation should be introduced early in the development process, leaving the user to fill in the gaps, that is the Ada function body.

Minkowitz and Henderson [32] presented a formal definition of an object-oriented environment using VDM. The principal concepts to be defined are inheritance and message passing. The object-oriented architecture is derived from the Smalltalk ar-

chitecture, which is simpler and has the merit of having been given a concise formal specification. VDM is a tool used to design applications environment, explore a space of alternative environments and choose one that best suited to user's needs. The final architecture can then be exploited to design seemingly complex applications naturally.

Breu [7] provided a framework for the integrated design of object-oriented programs with algebraic specification techniques. The design method pursued relies fundamentally on the structure of systems based on the notion of data types. In the approach presented, data types are described at different levels of abstraction. Problem oriented descriptions in terms of algebraic specification are combined with machine executable descriptions in terms of typed object-oriented programs. Two aspects play a crucial role for the integrated design environment. On the one hand, the use of formal specification techniques provides a framework in which the correctness of a program can be argued. On the other hand, a uniform structuring concept for object-oriented programs and algebraic specifications is developed to provide an integrated design environment. Two sample languages used are the algebraic specification language OS and the kernel of a typed object-oriented programming language OP.

Similar to these approaches, VDM/C++ also incorporates formal specification and object-oriented design, the formal notation being VDM. However, the novelty in VDM/C++ is that the object-oriented design is independent of any object-oriented system or language.

2.4 The Scope of This Thesis

Alagar and Periyasamy[2] provide a methodology for transforming a model-based specification into an object-oriented design. The methodology is independent to any specific model-based specifications. In the paper, VDM specification language has been chosen, and a library management system is used to illustrate the transformation process. Gao [19] presents a tool GAP, which supports direct and automatic transformation from VDM specification of software system into object-oriented de-

sign. GAP accepts as its input a simplified VDM specification and generates an object-oriented design. The design produced is language independent. It can be applied to any object-oriented language in the later stage.

The goal of this thesis is to build an interactive environment which supports the development of software system from VDM specification to C++ implementation. To be specific, our tasks are:

- to implement the VDM variable types in C++ : see Chapter 5 for details.
- to implement the built-in operators associated with the VDM variable types in C++ : see Chapter 5 for details.
- to implement all object-oriented concepts in the design in C++ environment : see Chapter 5 for det ils.
- to built tools supporting the automatic transformation from object-oriented design to C++ classes : see Chapter 5 for details.
- to create a multi-window user interface supporting the user interaction : see Chapter 6 for details.
- to define the syntax for object-oriented design : see Appendix A for details.
- to integrate GAP into VDM/C++ environment : see Chapter 1 and Chapter 4 for details.

Appendix B is a complete example for the development of the library management system under VDM/C++. It includes a VDM specification, it's corresponding object-oriented design and the resulting C++ classes. The specification and design are taken from [2] and revised to meet the syntax in VDM/C++ environment.

Chapter 3

Specification Language VDM – a Brief Outline

Vienna Development Method (VDM)[13] is a model based approach in which the description of a system is given as a series of models. It was developed in the 1970s by the IBM Vienna Research Laboratories. Each model comprises of two components - structure and operations. The structure is given by a set of abstract data types. If variables consistent with these data types are defined, then there exists a state space whose instantaneousness is given by the set of values associated with these state variables. A VDM specification may also contain only abstract data types without such global variables in which case the specification is said to be *property-oriented.* Both property and model-oriented specification styles are useful in describing the functionalities and behavior of software systems and reactive systems. Operations in VDM describe the changes to the values of some variables defined in the state.

3.1 Variables and Types

A variable is part of the internal state of the system being modeled. Each variable has a type which denotes a set of possible values. Table 3.1 lists all built-in types provided by VDM. Except for the **token** type, all others are self explanatory. The **token** type in VDM contains countably infinite number of distinct values, called *tokens.*

Enumerated types similar to those in Pascal are also provided.

Example : Colour = {RED, GREEN, YELLOW}

Z	Integer
N	Natural number
N_1	Natural number excluding zero
R	Real number
Q	Rational number
B	Boolean
char	Character
token	Token type

Table 3.1: Built-in types in VDM

Days = {SUNDAY,MONDAY,TUESDAY,WEDNESDAY,
THURSDAY,FRIDAY,SATURDAY}

It is also possible to introduce "union" types in which the value of a variable may be drawn from either of the two constituent types. The definition

T = T_1 | T_2

states that a variable of type T may assume the values either from type T_1 or T_2.

NIL is a reserved keyword used to represent both the name and its value of a type, specially used to represent undefined values.

A variable in VDM can be declared as

IDNUMBER : Idtype

NAME : String

VDM has the following conventions for its variables and types. A type name starts with an upper case alphabetic followed by lower case alphanumerics. Constants of scalar types and variables are in upper case. Instantiations of variables are represented in lower case. A variable name with an apostrophe at the end is used on the left side of an assignment statement to show that it is being updated.

3.1.1 Powerset Types

It is often useful to introduce into a specification a variable which takes as its value a set of objects of some type. The type of such a variable would be a set of sets. The suffix '-set' appended to a type name creates the power set for that type. For

Operator	Synopsis	Meaning
card	card S	cardinality of the set S
$\in$	x $\in$ S	x is the member of set S
$\cup$	$S_1 \cup S_2$	set union
$\cap$	$S_1 \cap S_2$	set intersection
$-$	$S_1 - S_2$	set difference
$\subseteq$	$S_1 \subseteq S_2$	subset
$\subset$	$S_1 \subset S_2$	proper subset
$=$	$S_1 = S_2$	set equality
$\neq$	$S_1 \neq S_2$	set inequality
$\bigcup$	$\bigcup$SS	distributed union of the sets SS
$\bigcap$	$\bigcap$SS	distributed intersection of the sets SS
{i,...j}	{i,...j}	subset of integers from i to j, both inclusive

Table 3.2: Set Operations in VDM

example, the powerset of type *Colour* given above is the set:

{ {RED,GREEN,YELLOW}, {RED, GREEN},
{RED, YELLOW}, {GREEN, YELLOW}, {RED},
{GREEN}, {YELLOW}, {} }

Thus, any variable which has the type *Colour-set* would take a value from the powerset, that is, a set of colour or the empty set. Empty set is represented by {}.

VDM uses the same notation and operations for sets as in conventional mathematics. Table 3.2 shows all set operators.

Let S_1 = {A, B, C} and S_2 = {C, D, E}

card(S_1) = 3

A $\in$ S_1 is TRUE

$S_1 \cup S_2$ = {A, B, C, D, E}

$S_1 \cap S_2$ = {C}

$S_1 - S_2$ = {A, B}

$S_1 \subseteq S_2$ is FALSE

$S_1 = S_2$ is FALSE

The distributed union is defined as follows:

$$\cup \{ \{A,B\}, \{B,C\}, \{C,D\} \} = \{A,B,C,D\}$$

3.1.2 List Types

It is often useful to introduce into a specification a variable which takes as its value an ordered collection of values. Lists (sometimes referred to as sequences or tuples) provide a suitable data structuring facility in VDM. The suffix '-list' when attached to a type name creates the set of finite lists which can be created from the base type. This is an infinite set. For the type *Colour* given above, Colour-list is as follows:

Colour-list = { <>,
<RED>,
<GREEN>,
<YELLOW>,
<RED, GREEN>,
<RED, YELLOW>,...
.....
/*all other finite lists of Colour*/
}

Thus variables of type Colour-list would have a value either a list of items of colour or the empty list. Empty list is represented by <>. All list operators available are displayed in Table 3.3.

Let $list_1$ = <A,B,A,D>, $list_2$ = <B,C> and $list_3$ = <D, E>
Let SS = < $list_1$, $list_2$, $list_3$ >
len $list_1$ = 4
$list_1$ || $list_2$ = <A,B,A,D,B,C>
conc SS = <A,B,A,D,B,C,D,E>
hd $list_1$ = A
tl $list_1$ = <B,A,D>
inds $list_1$ = {1,2,3,4}
elems $list_1$ = {A,B,D}
$list_1(1,3)$ = <A,B,C>

Operator	Synopsis	Meaning
len	len S	length of the list S
$\|\|$	$S_1 \|\| S_2$	list concatenation
conc	conc SS	distributed concatenation of the list of lists SS
hd	hd S	head of the list S
tl	tl S	tail of the list S
inds	inds S	indices of the list S; returned as a set of positive integers
elems	elems S	set of elements comprising the sequence S
(i,...,j)	S(i,...,j)	subsequence of S from i^{th} element to the j^{th} element, both inclusive.
$=$	$S_1 = S_2$	list equality
$\neq$	$S_1 \neq S_2$	list inequality

Table 3.3: list operations in VDM

$list_1 = list_2$ is FALSE

3.1.3 Record Types

While sets and lists represent structures of same type of elements, composite types from different subtypes can be represented by records. The general format of a record definition is given in the follow example:

Person :: AGE : N_1

SEX : {MALE, FEMALE}

MARRIED : B

The construct operation (denoted as "mk-") on record type is used to create an instance of record type. Thus

mk-Person(30, MALE, FALSE)

will create a new person.

Individual fields of the record can be selected by the names of the fields themselves. Thus

AGE(David) = 30,

SEX(David) = MALE and

MARRIED(David) = FALSE

Records can also be tested for equality and inequality.

3.1.4 Mapping Types

Map is an abstraction for finite function. It provides a mapping from elements of one set(known as domain) to those of another set(known as range), with the restriction that no element of the domain is mapped to more than one element. Maps differ from functions in that the domain of a map must be finite. The empty map is represented as [ ].

The map operators are defined in Table 3.4 and the semantics for some of them are explained in the following example:

Let $M_1 = [1 \rightarrow a, 2 \rightarrow b, 3 \rightarrow c]$, $M_2 = [1 \rightarrow a, 3 \rightarrow a]$,

$M_3 = [a \rightarrow \theta, b \rightarrow \phi, c \rightarrow \varphi]$ and $M_4 = [1 \rightarrow a, 4 \rightarrow d]$

dom $M_1 = \{1, 2, 3\}$

rng $M_1 = \{a, b, c\}$

$M_1^{-1} = [a \rightarrow 1, b \rightarrow 2, c \rightarrow 3]$

Notice that the inverse of a map M will not be a map if M is not bijective; for example M_2^{-1} is not a map

Two maps can be merged only if they have consistent maplets.

$M_1 \cup M_4 = [1 \rightarrow a, 2 \rightarrow b, 3 \rightarrow c, 4 \rightarrow d]$

While applying the overwrite operator, if a domain element appears in both the original mappings, the range element in the second operand takes priority in the resulting mapping.

$M_1 \dagger M_2 = [1 \rightarrow a, 2 \rightarrow b, 3 \rightarrow a]$

Two maps can be composed only if the range of the first map is a subset of the domain of the second map.

$M_1 \circ M_3 = [1 \rightarrow \theta, 2 \rightarrow \phi, 3 \rightarrow \varphi]$

$\{1\} \lhd M_1 = [1 \rightarrow a]$

$\{1\} \lhd\!\!\!\!- M_1 = [2 \rightarrow b, 3 \rightarrow c]$

$M_1 \rhd \{b\} = [2 \rightarrow b]$

Operator	Synopsis	Meaning
dom	dom M	domain of the map M
rng	rng M	range of the map M
M^{-1}	M^{-1}	Inverse of the map M
$\cup$	$M_1 \cup M_2$	map union
$\dagger$	$M_1 \dagger M_2$	map overriding
$\circ$	$M_1 \circ M_2$	map composition
( )	M(a)	map application
$\lhd$	$D \lhd M$	domain restriction
$\lhd\!\!\!-$	$D \lhd\!\!\!- M$	domain subtraction
$\rhd$	$M \rhd R$	range restriction
$\rhd\!\!\!-$	$M \rhd\!\!\!- R$	range subtraction

Table 3.4: Map operations in VDM

$M_1 \rhd\!\!\!- \{b\} = [1 \rightarrow a, 3 \rightarrow c]$

$M_1(1)$=a, $M_2(3)$=a, $M_4(4)$=d

3.2 Operations and Invariants

Operations in VDM describe the behavior of entities described in the state space. Each operation is described by means of pre- and post-conditions. The former asserts the constraints that are to be satisfied before the operation is invoked while the latter specifies the constraints to be met after the operation successfully terminates. Each operation also specifies the set of global variables affected by that operation (and hence the change in that portion of the state space) and the mode of change (read/write) for each such global variable. In case of property-oriented specifications, one could naturally expect only the pre- and post-conditions alone to be specified without any reference to the global variables. In both cases, operations may have parameters whose types are well defined in the structural part of the specification.

There are two types of invariants in VDM - *type invariant* and *state invariant*. A type invariant, as the name implies, is associated with a type. Typically, a type invariant is specified for all user defined composite types since the behavior of all

other types are well defined within the specification language. In a similar way, a state invariant is associated with a state space. Being specified, a type invariant(state invariant) asserts that every instance of that type(state) should respect the properties asserted by the invariant.

Operations and Invariants are usually specified in VDM using predicates. Complex predicates can be constructed by means of the logical connectives and the universal and existential quantifiers. We simply list the predicates operators here. Interested readers can refer to [27] for further explanation.

$\sim$	not
$\wedge$	and
$\vee$	or
$\equiv$	is equivalent to(iff)
$\Rightarrow$	implies
$\forall$	for all...
$\exists$	there exist...
$\exists!$	there exist exactly one...

In addition, for simplication purpose, predicates can be named using the **let** construct. For example:

let p = (a + b) **in**

q = p*p + p + 1

is the simplified form of q = (a+b)(a+b)+(a+b)+1.

To illustrate all concepts introduced in this chapter, we provide a sample VDM specification taken from [8].

Example 3.1

This example specifies the requirements for a simple database system to be used by a marriage bureau. The bureau wishes to record in the database the clients who wish to be introduced to suitable partners, and those ex-clients who have been suitably accommodated.

The VDM specification of the system is given below. The state of the system is defined using two global variables. UNMARRIED represents the set of unmarried people in the database, while MARRIED represents the set of successfully matched clients. There are three operations in the specification. The REGISTER operation is used when a new client is to be registered. The precondition states that the new person must not be already registered as a member of either set of clients. The post-condition defines the effect of the operation in terms of an update to the state, in which the set of unmarried clients will be increased by unioning it with the singleton set containing the new client. The MARRY operation is used when two clients are successfully matched. The precondition states both partners must be members of the set of unmarried clients. The postcondition defines the required state change to be the addition of the couple to the married set and their removal from the unmarried set of clients. A *let* clause is used to clarify the postcondition for the human reader. The INIT operation is used to initialize the system. Its precondition is always *TRUE* so by convention it is omitted. The postcondition gives the effect of the operation as initializing both sets of clients to be empty.

State:: UNMARRIED :Person-set
MARRIED :Person-set
Person= /*some suitable representation*/

REGISTER(P: Person)
ext UNMARRIED :**wr** Person-set /* **wr** represents read and write */
MARRIED : **rd** Person-set /* **rd** represents read only */
pre p $\notin$ unmarried $\wedge$ p $\notin$ married
post *unmarried'* = unmarried $\cup$ {p}

MARRY(M: Person, W: Person)
ext UNMARRIED :**wr** Person-set
MARRIED : **wr** Person-set

pre m ∈ unmarried ∧ w ∈ unmarried

post **let** couple = {m, w} **in**

$married' = married \cup couple \wedge$

$unmarried' = unmarried - couple$

tel

INIT()

ext UNMARRIED :**wr** Person-set

MARRIED : wr Person-set

post $unmarried' = \{\} \wedge married' = \{\}$

It is clear that no individual person should be present in both the set of unmarried clients and the set of married clients. This invariant condition is documented as follows.

inv-State $\triangleq$ unmarried ∩ married = { }

3.3 Using VDM in System Development

VDM methodology provides more than just a specification language. Besides formal notations, it provides rules and procedures to be followed in the various stages of system development. VDM encourages a layered top-down development of systems, by supporting abstract data type at the uppermost levels of description. These abstract data types capture the system concepts at the highest level in order to explain the functionality of the system. Mathematical abstractions such as sets, lists and finite mapping are used at this level and operations and functions are specified by means of pre- and post- conditions. Development of the system towards design and implementation is achieved by refining the abstract data structures through various levels and adding more implementation detail at each stage of refinement.

VDM describes three dimensions of refinement[15]: *data reification*, which corre-

sponds to a choice of data representation; *implementation*, which corresponds to a choice of algorithm; and *operation decomposition*, which corresponds to the implementation of operation by smaller ones or by program statements. We consider only the "reification" in this thesis. At each step of refinement, new operation specifications have to be formulated in terms of the reified data structures to correspond to those at the more abstract level.

The use of "retrieve functions" which map the new data structure onto the old abstract one at each stage makes it possible to verify that the reified specifications correctly model the effects of the abstract specification that they are intended to represent. More specifically, to show that the new representation is sufficient and consistent with respect to the previous specification, we are interested in *adequacy*: everything representable at the abstract level must be representable at the concrete; and in *uniqueness*: no concrete representation can have more than one abstract representation. Obviously it is necessary at the final step that the code produced correctly fulfills the least abstract specification.

Example 3.2

This example explains the reification process discussed in this section. The specification is taken from [1] which specifies a banking system. In the system, all passbook entries are kept in a folder, called a ledger. Each passbook consists of an account number, an account type, the name of the customer and the balance. To simplify the system, we assume that:

- No two customers have the same name.
- Account numbers are unique.
- There are several types of accounts maintained in the system; however, every customer is restricted to hold only one type of account.
- Each type of account has a pre-defined minimum balance which the account

holder has to maintain.

There are four types of transactions provided for the customer:

- OPEN: to open a new account by depositing an amount greater than or equal to the minimum balance, depending on the type of account to be opened.
- DEPOSIT: to deposit an amount into account already opened.
- WITHDRAW: to withdraw an amount from an existing account.
- CLOSE: to close an existing account and withdraw the balance amount.

The most abstract level VDM specification is listed below. And here we only show one operation for opening a new account.

State::
LEDGER: Ledger-leaves
Ledger-leaves = Accno $\rightarrow$ Passbook
Accno = N_1
Passbook :: ACC-NUM : N_1
ACC-TYPE : Typecode
ACC-NAME : Namestring
BALANCE : N_1
MIN-BALS : Mini-balance
Mini-balance = Typecode $\rightarrow$ Minimum
Minimum :: ACC-TYP : Typecode
BAL-MINI : N_1
BANK-BALANCE : N_1
TAKEN-OUT : N_1
PUT-IN : N_1
/* "Typecode" and "Namestring" are to be defined at the lower level */

OPEN(NEWNAME: Namestring; ACCOUNT: Typecode; AMOUNT: N_1)

/* to open a new account of type "ACCOUNT" under the name "NEWNAME" by depositing an amount "AMOUNT"*/

ext

LEDGER : **wr** Ledger-leaves

MIN-BALS : **rd** Mini-balances

BANK-BALANCE : **wr** N_1

PUT-IN : **wr** N_1

pre

/* NEWNAME should not exist in LEDGER before */

$(\forall ps \in$ rng ledger)(newname $\neq$ ACC-NAME(ps))

/* AMOUNT $\geq$ minimum amount required to open an account of type "ACCOUNT"*/

$\wedge$ account $\in$ dom min-bals

$\wedge$ **let** minrec = min-bals(account) **in**

amount $\geq$ BAL-MINI(minrec)

tel

post

/* create a new account and add this into LEDGER */

let new-acno = get-new-acno() **in**

let new-passbook = mk_Passbook(new-acno,account,newname, amount) **in**

ledger' = ledger † [new-acno $\rightarrow$ new-passbook]

tel

$\wedge$ bank-balance' = bank-balance + amount

$\wedge$ put-in' = put-in + amount

tel

get-new-acno(): $\rightarrow N_1$

To refine the specification, the data type *Ledger-leaves* is reified. We use two arrays to represent the abstract type map which is from *Accno* to *Passbook*. The retrieval function is :

retr: (array[1..N] of Accno, array[1..N] of Passbook) → (Accno → Passbook)
retr(A_1, A_2) $\triangleq$ [$A_1[i] \rightarrow A_2[i] / 1 \leq i \leq N$]

The following is the refined specification.

State::
$LEDGER_1$: array [1..N] of Accno
$LEDGER_2$: array [1..N] of Passbook
Accno = N_1
Passbook :: ACC-NUM : N_1
ACC-TYPE : Typecode
ACC-NAME : Namestring
BALANCE : N_1
MIN-BALS : Mini-balance
Mini-balance = Typecode → Minimum
Minimum :: ACC-TYP : Typecode
BAL-MINI : N_1
BANK-BALANCE : N_1
TAKEN-OUT : N_1
PUT-IN : N_1

OPEN(NEWNAME: Namestring; ACCOUNT: Typecode; AMOUNT: N_1)
/* to open a new account of type "ACCOUNT" under the name "NEWNAME" by depositing an amount "AMOUNT"*/

ext

$LEDGER_1$: **wr** array [1..N] of Accno

$LEDGER_2$: **wr** array [1..N] of Passbook

MIN-BALS : **rd** Mini-balances

BANK-BALANCE : **wr** N_1

PUT-IN : **wr** N_1

pre

/* NEWNAME should not exist in LEDGER before */

$(\forall ps \in \text{rng}(\text{retr}(ledger_1, ledger_2)))(\text{newname} \neq \text{ACC-NAME(ps)})$

/* AMOUNT $\geq$ minimum amount required to open an account of type "ACCOUNT"*/

$\wedge$ account $\in$ dom min-bals

$\wedge$ **let** minrec = min-bals(account) **in**

amount $\geq$ BAL-MINI(minrec)

tel

post

/* create a new account and add this into LEDGER */

let new-acno = get-new-acno() **in**

let new-passbook=mk_Passbook(new-acno,account,newname, amount) **in**

$\text{retr}(ledger_1', ledger_2') = \text{retr}(ledger_1, ledger_2)\dagger[\text{new-acno} \rightarrow \text{new-passbook}]$

tel

tel $\wedge$

bank-balance' = bank-balance + amount $\wedge$

put-in' = put-in + amount

get-new-acno(): $\rightarrow N_1$

Since the data type is reified, the pre- and postconditions are modified to match the new data type. It is obvious that new representation is more specified and yet remains consistent with respect to the old one. So we believe that these two specifications

are equivalent, that is, they specify the same system. We will bring this example to Chapter 4 and Chapter 5 where two corresponding designs and implementations will be presented.

3.4 The Choice of VDM

VDM specification has been evolving for the last several years. A standard for VDM semantics is currently being defined by a joint committee of the International Standards Organization, ISO SC22/WG19-VDM, and the British Standards Institute IST/5/50. A reference guild to the standard notation [13] which is the notation used in this thesis has been published in advance of the standard.

VDM is a model-based technique which is close to the user's way of conceiving the problem. A variable type in the model is an abstraction of a physical object. We can extract classes of objects from variable types. When an operation in VDM is specified, we focus on only a portion of the state space affected by the operation. The related step in object-oriented paradigm is to identify all functions affecting an object in one class. These facts makes it possible to derive an object-oriented design from VDM specification. As we will also see from the following chapters, VDM specification provides not only notations to support data abstraction and system modelling, but also clues to implementations. As we mentioned above, the development of a system using VDM consists of a number of refinement steps from specification to implementation. The transformation methodology from VDM to OOD presented in [2] is suitable for all refinement stages. Some methodologies [12] on the transformation of a semi-formal software description to a VDM specification is under the development. Moreover, automatic proof checking systems exist to verify a VDM specification. The Mural System [24] is one of them. We will give a more detailed introduction to the Mural System in the last chapter.

Chapter 4

From Functional Specification to Object-Oriented Design

This chapter briefly reviews the methodology for deriving an object-oriented design from functional specification of requirements [2]. Also we present the system architecture for GAP which is a tool built to implement the methodology.

4.1 Methodology

In VDM specifications, a variable and its type are abstractions of the physical object. The relationship between a variable and its type is the same as that of an object and its class in object-oriented design. However in object-oriented design, a class is more than merely an abstraction of physical object. The access to data of a class is restricted to a specific set of functions known as member functions. Member functions should be contained as part of a class declaration. At specification stage, when an operation is specified, we focus on only a portion of state space affecting one or more objects in the problem domain. The related step in object-oriented paradigm is to identify all member functions affecting an object. To derive an object-oriented design from a given VDM specification, the methodology tries to link classes, their attributes and member functions in design with data types, variables and operations in the specification. More specifically, there are four stages in this transformation. These are:

- identifying the classes in the design.
- identifying the attributes within each class.
- deriving member functions for each class.
- deriving *inheritance* and *part-of* relationship between classes.

4.1.1 Identifying Classes

Since the state of the system in VDM is represented by a set of global variables, these variables form the structural components (i.e., the objects) of the object-oriented design. Hence, the first task is to transform the data types defined in the specification into classes in the object-oriented design.

For the built-in types **Z, N, R, B, char** in VDM, we may assume that there exists associated built-in types in the object-oriented language chosen for implementation. **Q**, N_1 are considered to be primitive classes of the object-oriented environment. Hence, for all these basic types separate classes in the object-oriented environment need not be created. Every other data type in the state space that model an entity can be mapped into a unique class in the design.

For each simple type, a unique class is created in the design. For each composite type in the specification, a new class is created in the design corresponding to the composite type, and a unique class is created for each component type of this composite type, making sure that redundant classes are not created and the component type is not basic types. This way, the class corresponding to a composite type will have data member whose type is another class, thus creating the part-of relationship between classes.

Example 4.1

Consider a VDM state space specification, in which a composite type called "Circle" is defined as

Circle :: CENTRE : Point
RADIUS : R
PLANE-OF-CIRCLE : Plane

This VDM specification represents a 3-D circle by three components - centre, radius and plane in which the circle lies.

In the design, there is a class corresponding to "Circle"; in addition, classes corresponding to "Point" and "Plane" are also created. We do not create a corresponding class for "R", since it is considered to be a basic type. The syntax for classes is given in Appendix A.

Having generated classes for all data types in the specification, a root class is generated to include all other classes which corresponds to the state of the system in the specification.

4.1.2 Identifying Attributes of Classes

Each global variable in the specification becomes the attributes of the root class, thus ensuring the consistency between the root class and the state of the system in the specification. Each composite type in the specification is defined by a set of variables called *fields* and their associated types. These variables become the attributes of the class corresponding to the composite type. The process is repeated for every composite type in the specification.

Example 4.2

Consider the VDM specification fragment below.
LEDGER : Ledger-leaves
Ledger-leaves = Accno $\rightarrow$ Passbook
Accno = N_1
Passbook :: ACC-NUM : N_1
ACC-TYPE : Typecode

ACC-NAME : String

BALANCE : N_1

The classes and the attributes within each class obtained by the transformation are :

class Bank-System (*Root Class*)

attributes

LEDGER :Ledger-leaves

class Ledger-leaves

attributes

class Passbook

attributes

ACC-NUM : N_1

ACC-TYPE : Typecode

ACC-NAME : String

BALANCE : N_1

class Typecode

attributes

class String

attributes

LEDGER is a global variable which becomes one attribute of the root class. All field variables in the composite type *Passbook*, ACC-NUM, ACC-TYPE, ACC-NAME and BALANCE are transformed to the corresponding class *Passbook*. Attributes for the class *Ledger-leaves*, *Typecode* and *String* are not dealt with in the specification and are introduced in either detailed design or the implementation.

4.1.3 Deriving Member Functions

Since member functions of an object define the behavior of the object similar to the way that operations in the specification define the behavior of the state space entities, there exists a correspondence between the operations in the specification and the member functions in the design. The process of deriving these member functions of objects from VDM specifications is explained in [2]. A brief summary of the derivation of member functions is given below:

> An operation *Op* in the specification will be transformed into a member function in a class C (and hence assigned as a member function of C) if one or more of the variables accessed in this operation are already transformed as attributes of C. The justification of this process comes from the fact that the set of variables accessed in *Op* determines the portion of the state space affected by *Op*. Consequently, their mapping into attributes of C confirms that this portion of the state space corresponds to an isolated object belonging to C.
>
> Since the variables might have been transformed into several classes, the same operation might be assigned as member functions of all these classes with the same name and parameters. Renaming of these member functions, if necessary, will be done later depending on the semantics of the operation *Op*.

Example 4.3

Consider the following VDM operation in Appendix B.

ADD-BOOK(T: String; AU: String)
ext LIB-SYSTEM: **wr** Library
post
($\exists cn \in CNtype$)
(($\forall b \in$ COLLECTION(lib-system))

((CALLNUMBER(b) $\neq$ cn) $\wedge$

(**let** new-book = mk_Lib-books(t, au, cn, "inshelf") **in**

COLLECTION(lib-system)' = COLLECTION(lib-system)$\cup$ {new-book}

tel)))

Since the operation *ADD-BOOK* accesses variables *COLLECTION* and *CALL-NUMBER* which are transformed to attributes in class *Library* and *Lib-books* respectively, we create one member function in each class corresponding to this operation. Later the member function in class *Lib-books* may be renamed as "new-book" which returns a new call number. The information for this redefinition of function names is available from the corresponding pre- or post-condition clause in operation. And the redefinition can be completed through the user interaction.

The type invariants and state invariants of the VDM specification can be transformed into invariants of classes in the design by a procedure which is similar to the derivation procedure for member functions. Invariants are important in providing semantic information to derive the relationships among the classes.

4.1.4 Deriving the Relationships among the Classes

The logical relationships between the pre- and post-conditions of an operation and between the operations themselves give rise to the relationships between the objects. The data type invariants and state space invariants in the specification will be helpful in identifying such relationships. The method explained in [2] can be briefly stated as "weakening the pre- and strengthening the post-" conditions. The class inheritance relationship is determined by a thorough analysis of such logical relationships among the stated pre- and postconditions in VDM specification.

Example 4.4

This example provides two object-oriented designs which are derived from specifications in Example 3.2 using the methodology presented in this section.

OOD I(derived from the abstract VDM specification):

class banking-system1

attributes

LEDGER : Ledger-leaves;

MIN-BALS : Mini-balances;

BANK-BALANCE : N_1;

TAKEN-OUT : N_1;

PUT-IN : N_1;

operations

OPEN(NEWNAME: Namestring; ACCOUNT: Typecode; AMOUNT: N_1)

class Ledger-leaves

attributes

class Passbook

attributes

ACC-NUM : N_1;

ACC-TYPE : Typecode;

ACC-NAME : Namestring;

BALANCE : N_1;

operations

GET-ACC-NAME() : Namestring /* redefined from open */

class Typecode

attributes

class Namestring

attributes

class Mini-balances

attributes

class Minimum

attributes

ACC-TYP : Typecode;

BAL-MINI : N_1;

operations

GET-BAL-MINI() : N_1 /* redefined from open */

OOD II(derived from the refined VDM specification):

class banking-system2

attributes

$LEDGER_1$: array [1..N] of Accno;

$LEDGER_2$: array [1..N] of Passbook;

MIN-BALS : Mini-balances;

BANK-BALANCE : N_1;

TAKEN-OUT : N_1;

PUT-IN : N_1;

operations

OPEN(NEWNAME: Namestring; ACCOUNT: Typecode; AMOUNT: N_1)

class Passbook

attributes

ACC-NUM : N_1;

ACC-TYPE : Typecode;

ACC-NAME : Namestring;

BALANCE : N_1;

operations

GET-ACC-NAME() : Namestring /* redefined from open */

class Typecode
attributes

class Namestring
attributes

class Mini-balances
attributes

class Minimum
attributes
ACC-TYP : Typecode;
BAL-MINI : N_1;
operations
GET-BAL-MINI() : N_1 /* redefined from open */

The number of classes in these two design is different. The attributes and their types in class *banking-system1* and *bank-system2* are quite different. It is difficult to see that these two designs are equivalent by just checking them. But from the fact that they are derived from two equivalent specifications and the correctness of the transformation methodology we believe that these two designs are equivalent.

The design obtained using the methodology discussed in this section can be improved further in later stages through user interaction. In particular, during the user interaction:

- The names of member functions can be redefined so as to remain meaningful in the contexts. The information for this redefinition is available from the

corresponding predicate clause in VDM operation.

- Redundant member functions within a class must be deleted from the class.
- One member function may be split to two or more to implement the correspond ing VDM clause.
- Some of the member functions within one class may be merged.
- Some new member functions may be created to implement the existing member function.

A detailed discussion about some of issues listed above will be given in the next chapter.

4.2 System Architecture for GAP

GAP[19] is one of the system tools which was built to implement the transformation method discussed in the previous section. GAP serves two purposes: first as a *syntax parser* and *partial semantics checker* for the underlying VDM specification and second as *an automatic transformer* to generate classes and their member functions in the design. A complete description of GAP and its functionalities are given in [19]. The GAP architecture is shown in Figure 4.1.

The syntax checker is composed of a scanner and a parser. The scanner reads the input of VDM source specification and generates internal tokens to feed the parser. The parser checks the input of specification for syntactic correctness, recognizes the variables, operators and key words. An important task of the parser is the generation of the symbol table.

The transformer consists of the class generator and operation handler.

The class generator takes information from symbol table and generates classes. Attributes within each class and relationships between classes are also created by the class generator.

The operation handler can be divided further into three functional parts:

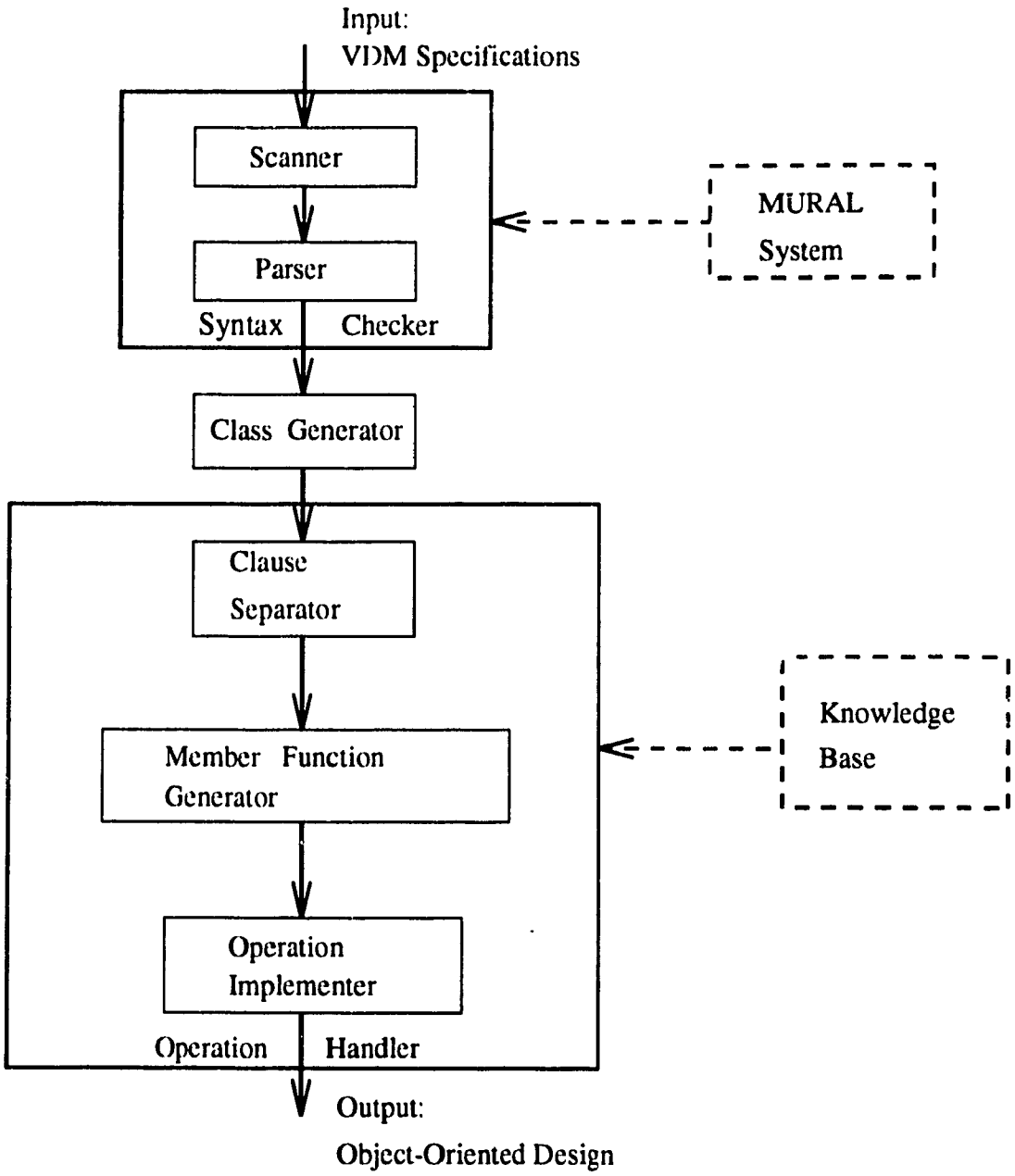


Figure 4.1: System Architecture of GAP

- the clause separator.
- the member function generator. and
- the operation implementor.

The clause separator splits the pre- and postconditions into clauses according to the syntactic analysis. The member function generator categorizes each clause into different classes according to the variables they affect and generates member functions corresponding to each clause. It also reorganizes member functions generated so far to eliminate some duplication. The operation implementor determines the primary member functions for the VDM specified clauses. The primary member function is the function which will perform the operation specified by the clause. The implementation of all primary member functions for a VDM operation implys the implementation of the operation.

The input to the system is a VDM specification file.

There are four output files generated for each VDM specification input.

- Listing file(with extension .lst): The original source of input is listed with line numbers. At the end of the file, there is a summary of errors. This file is useful for error correction.
- Symbol table file(with extension .tab): This file saves the symbol table built by GAP for the underlying source input.
- Object-oriented design file(with extension .ood): The object-oriented design for the underlying VDM specification.
- Member function file(with extension .mf): Each line in the file displays a member function name in the design and its corresponding predicate clause in VDM.

The third file is the most important one which shows the class information in the design. For each class, it provides :

- class name

- inheritance and part_of relationship
- attributes and their types
- suggested member functions and their parameters

The syntax for the OOD file is shown in the Appendix A. Example 4.5 lists some pieces taken from Appendix B to show the layout of OOD.

Example 4.5

class Lib-books
part_of Library
attributes
TITLE: String;
AUTHORS: String;
CALLNUMBER: CNtype;
BSTATUS: Book-status:
operations
INITIALIZE-BOOK()
NEW-BOOK() : CNtype

class Borrowers
part_of Library
attributes
IDNUMBER: IDtype
NAME: String
USTATUS: String
operations
INITIALIZE-USER()
NEW-USER() : IDtype

The last file reveals the possible relationship between the suggested member functions and their corresponding predicate clauses in VDM. The information in the file explains the semantics of each member function which will become important during the implementation of member functions. Through the user interaction which will be discussed in Chapter 6, the developer can access both the object-oriented design file and the member function file. Information in these two files is essential in the further development of the underlying system.

Chapter 5

Methodology for Transforming OOD to C++

Issues discussed in the previous chapters are language independent. To implement the design, a specific language is needed. We chose C++, since it provides sufficient features to implement object-oriented concepts such as *abstract data type, inheritance, polymorphism* and *dynamic binding.* An implementation contains much more detail than its corresponding design, and consequently may require information that is not stated in the design document. This additional information can be obtained through user interaction. In our discussions below, we indicate the particular points where additional information may become necessary for the implementation.

5.1 Classes

There are two interpretations of the class concept in object-oriented paradigm[35]. As we have discussed in Chapter 2, a class in C++ is considered as a user defined type. Ideally, a user-defined type should not differ from built-in types in the way it is used, except in the way it is created. In VDM/C++ environment, we have built general classes set, list and map. A detailed discussion about these classes will be given in the next section. If a class in the design represents a set, list or map, it is unnecessary to convert it to a C++ class. For example, consider the OOD class *Loanmap* which is a map from *IDtype* to *Duemap* in Library Management System in Appendix B. *Loanmap* can be defined as *typedef map<IDtype, Duemap> Loanmap*

OOD Attributes types	C++ data member types
Z	int
N	unsigned int
R	float
Char	char
B	boolean
Q	Rational
set of type	set<type>
list of type	list<type>
map from dom to ran	map<dom, ran>
N_1	unchanged
class name	unchanged
user defined type	unchanged

Table 5.1: Type processing

in C++ which is an instantiation of the general class *map*. Every other class in the design is mapped into a C++ class with the same name.

5.2 Attributes and their Types

All attributes in a class are transformed into *private* data members in the associated C++ class. Some of them may become *public* during the implementation of member functions through user interaction. Most of them will remain in the private part of the class.

Table 5.1 shows how the types of attributes are processed.

The type *boolean* in OOD can be defined as an enumeration type in C++ as given below:

```
typedef enum {false, true} boolean ;
```

Actually, *typedef* is used to declare a boolean type as an equivalent type to integer type in some C++ environment, and *false* is defined as zero and *true* is defined as one. C++ treats an enumeration type as an integer type. In the above enumeration

definition, *false* and *true* also be assumed the value zero and one respectively. The only difference is that if a variable is declared as an enumeration type, it's value can not be any integer other than zero and one.

Rational should be a defined class in most C++ environment.

It is easy to note that the **Record** and **Union** types in VDM will not exist in the design according to the transformation methodology introduced in the previous chapter.

Another possible implementation of data types **N** and N_1 is that we create two classes *ClassN* and *ClassN1*. The only difference between these two classes is that objects of class *ClassN1* can not be assigned the value zero. Other features and operations are identical. So, we can define *ClassN1* as a derived class from base class *ClassN*. The non-zero restriction can be ensured by initialization and member function implementations. A member function can be created to check this invariant condition.

Since *set*, *list* and *map* are defined as generic types in OOD, the C++ *template* is used to represent them. Thus, a set of integers can be defined as set<int> and map<int, char> defines a map with its domain type *integer* and range type *char*. All possible operations on these types are provided by their member function. Listing 5.1, 5.2, 5.3 given below, show C++ interfaces in VDM/C++ for set, list and map types in VDM. The corresponding operations in VDM can be found in Tables 3.2, 3.3 and 3.4.

We remark that there must be a 1-1 correspondence between VDM operations and the corresponding class interface member functions. The reason is that VDM clauses will be used as the definition for the implementation of corresponding member functions in C++ in later stage. VDM operations for set, list and map appear frequently in these clauses. If their corresponding member functions are available, the implementation of member functions is greatly simplified. Some of the C++ library classes may not have this 1-1 property and hence may be inadequate.

▷ Listing 5.1 ◁ Interface of class template set

```
template<class type>
class snode {
  type * obj;
  snode* next;
public:
  snode(type * o, snode *n) : obj(o), next(n) {};
  friend class set<type>;
};

template<class type>
class set {
  snode<type>* head;
 public:
  set() : head(0) {}//initialize the set to be empty
  int card();       //return the cardinality
  boolean mbship(type); //check the membership
  set uni(set);     //return the union of two sets
  set inters(set);  //return the intersection of two sets
  set diff(set);    //return the difference of two sets
  boolean subset(set);  //check the subset relationship
  boolean psubset(set);  //check the proper subset relationship
  boolean equal(set); //check the equality
  boolean nequal(set);//check the inequality
  set<int> isubset(int, int);
           //subset of integers from the first to the second

};
```

▷ Listing 5.2 ◁ Interface of class template list

```
template<class type>
```

```
class lnode {
  type * obj;
  lnode * next;
public:
  lnode(type * o, lnode *n) : obj(o), next(n) {};
  friend class list<type>;
};

template<class type>
class list {
private:
lnode<type> *head;

public:
list(): head(0) {} //initialize the list to be empty
int length();  //return the length of the list
list concat(list);    //concatenate two lists
type* hd();       //return the head of the list
type* tail();     //return the tail of the list
type* operator[](int);//return the element in the given position
set<int> inds();      //return the indices of the list
set<type> elems();    //return all elements in the list as a set
type* sublist(int, int);
          //return the sublist from the first integer given to
          //the second
boolean equal(list); //check the equality
boolean nequal(list);//check the inequality
};
```

▷ Listing 5.3 ◁ Interface of class template map

```
template<class domt, class rant>
class node {
  domt * obj1;
  rant * obj2;
  node * next;
public:
  node(domt* o1, rant* o2, node* n) : obj1(o1), obj2(o2), next(n) {};
  friend class map<domt, rant>;
};

template<class domt, class rant>
class map {
  node<domt, rant>* head;
public:
  map(): head(0) {} //initialize the map to be empty
  set<domt>  DomM();
       //return a set of domain elements which are in the map
  set<rant> RangeM();
       //return a set of range elements which are in the map
  map<rant,domt> ReverseM(); //reverse the map
  map munion(map);           //map union
  map concat(map);           //map overriding
  map mcomposition(map);     //map composition
  rant* m(domt);
       //return the range element for a given domain element
  map RestrictBy(set<domt>);
       //restrict the map by elements in the set
```

```
  map RestrictTo(set<domt>);
       //restrict the map to elements in the set
  map RestrictBy(set<rang>);
       //restrict the map by elements in the set
  map RestrictTo(set<rang>);
       //restrict the map to elements in the set
};
```

We use a link list to implement the data type set, list and map. The auxiliary class *node* represents one element in the type. The implementation details are omitted here.

For some data types in OOD, there is no detailed definition in VDM, these data types can only be defined and implemented through user interaction.

5.3 Member functions

Each member function in the object-oriented design is mapped to a unique member function in the associated C++ class, with all its parameters transformed and their types mapped according to the description in Section 5.2. The transformation process can be automated; however, user interaction will improve the detailed design. See Examples 5.1, 5.2, 5.3 and 5.4.

As mentioned earlier, every member function in design is related to one or more predicate clauses in VDM. The predicate clause is to be thought of as the specification for the member function.

Example 5.1:

Consider a VDM clause

$p = q + r$

representing a relation between three points p, q and r. Let p, q and r be mapped with type *point* into the data members of the class C in the design. In this case, the

clause p = q + r will be transformed into a member function Mf in class C. In order to implement the member function in C++, the two operations '+' and '=' should be included in the class *Point*, if they are not already defined in *Point*.

Example 5.2:

Consider an operation *Return_book* to return a borrowed book in a library environment (see the example on library management system given in Appendix B). The pre-condition for this operation might assert that every book returned must actually belong to the library, loaned to a valid or authorized user of the library and the status of the book in the database of the library should indicate that it is loaned. The predicate clause for such pre-condition is given below:

(∃!b∈COLLECTION(lib-system))
((CALLNUMBER(b)=cn) ∧
(BSTATUS(b)="loaned_out") ∧
(cn ∈ dom LOANS(lib-system)(id)))

According to the OOD derivation process described in Chapter 4, *callnumber* and *bstatus* are defined as data members of a class *Lib_book* (the type corresponds to the book) and there is only one member function for this class. It is clear from the pre-condition given above that there is a need to match the call number of the book and its status which are stored as part of the book. Hence, the only member function in the design must be split into two member functions in the implementation - one to check whether the call number matches and the other to check the status match. One possible C++ code is given below:

```
int same-call-number (int cn)
  if (cn = this.callnumber) return 1 else return 0
```

```
int status-match-loaned (string stat)
  if (stat = "loaned-out") return 1 else return 0
```

Example 5.3

See the example in Appendix B where the operation 'BORROW-BOOK' in specification is mapped to one member function in class 'Borrower' to implement the predicate 'IDNUMBER(u) = id'. During the automatic transformation processes, either from VDM to OOD or from OOD to C++, it is difficult to create meaningful member function name. The name of the member function can be redefined as 'Is-Member' through the user interaction. We claim that the information for this redefinition is available from the corresponding clause, that is 'IDNUMBER(u) = id'.

Example 5.4

Referring to the same example in Appendix B. The predicate 'IDNUMBER(u) = id' appears in both operation 'BORROW-BOOK' and 'RETURN-BOOK'. There is a redundancy. All redundant member functions can be merged. Suppose that there is another member function 'Is-Not-Member' in the class 'Borrowers'. It is possible to merge 'Is-Member' and 'Is-Not-Member' to become a single operation.

Member functions between object-oriented design and C++ are not necessarily one-to-one. During the implementation for one member function, we create some new member functions, merge existing member functions and redefine member functions according to the meaning of the corresponding clause. The implementation of a member function depends on the semantics of the clause as given in the VDM specification. In the status matching operation shown in Example 5.2, instead of writing one member function for every possible status of the book, one could write just two functions - *status-match* and *return-status* which will check the validity of status and

return the current status respectively.

5.4 Inheritance

Inheritance relationship between classes can be achieved by the mechanism of class *derivation* in C++. We show the derivation of inheritance relationship and C++ coding through an example.

Example 5.5:

Consider the following VDM specification fragment:

COURSES :	Courses-set
Courses	:: NAME: String REGISTERED: Student-**set**
Student	:: Grad | Undergrad
Grad	:: ID: **N** NAME: String TAKEN: Course-**set** SUPERVISOR: String
Undergrad	:: ID: **N** NAME: String TAKEN: Course-**set**

A course contains the name of the course and the set of students registered for the course. A student record contains the id number, name of the student and the set of courses taken by this student. If the student is a graduate student(grad type), then the supervisor's name is also included in the graduate student's record. The type Course is left undefined at this stage.

Consider an operation for a student to withdraw from a course. Assume that the undergraduate students cannot withdraw from a course and a graduate student can

do so only with the permission of the supervisor. Then the operation specification is as follows:

WITHDRAW(s: Student; c: Courses)

ext COURSES: **wr** Courses-**set**

pre

$c \in courses \wedge s \in REGISTERED(c) \wedge$

$s \notin Undergrad \wedge s \in Grad \Rightarrow permitted\ (s, SUPERVISOR(s))$

post

$courses' = (courses - \{c\}) \cup \{mk\_Courses\ (NAME(c), REGISTERED(c) - \{s\})\}$

Applying the OOD transformation procedures, we can derive the following object-oriented design for this example:

class System

attributes

COURSES: set of Courses;

operations

WITHDRAW(s: Student; c: Course)

class Courses

attributes

NAME: String;

REGISTERED: set of Student;

operations

GET-REGISTERED(): set of Student /* redefined from withdraw */

class Student

attributes

ID: **N**;

```
    NAME: String;
    TAKEN: set of Course;
operations
    CHECK-TYPE (): B      /* redefined from 'withdraw' */


class Undergrad
inherits Student

class Grad
inherits Student
attributes
    SUPERVISOR: String;
```

The C++ implementation of this design will be a straightforward task which involves the syntactic changes from the OOD notation to that of C++. Assuming the class *String* is given and the mapping of **N** to *unsigned int*, we get the following C++ implementation:

```
class System {
private:
    set<Courses> courses;
public:
    void withdraw(Student s, Courses c)
}


class Courses {
```

```
private:
    String name;
    set< Student > registered;
public:
    set<Student> get-registered();
}

class Student {
private:
    unsigned int id;
    String name;
    set < Course > taken;
public:
    boolean check-type ();
}

class Undergrad: public Student{
}

class Grad: public Student {
private:
    supervisor: String;
}
```

Notice that the class *Undergrad* inherits everything from *Student* but does not add, delete or change any of the inherited features. Such an inheritance is of no use. However, we have included it here to show the transformation. In actual implementation, the class *Undergrad* will be modified.

Having transformed into C++, the developer may redefine, split or merge some of the member functions or add some more data members and/or functions. For

example, since all the data members are kept private in this example, it would be appropriate to add a *get* function and a *set* function to read and write the data members respectively. The refined version of the class *Student* is given below:

```
class Student {
private:
    unsigned int id
    String name
    Set <Course> taken
public:
    unsigned int get_id();
    void set_id(unsigned int);
    boolean Is_student () { return 1;}
}
```

The function *Is_student* is redefined from *check_type* and returns 1 for an undergrad student. The same function will be once again redefined in *Grad* to return 0 in order to distinguish between undergraduate students and graduate students.

Implementation Strategy: Since the derived class will inherit some features from its base class and the client class may send message to its component class, we adopt the following strategies in the implementation of C++ classes:

- If there is an inheritance relationship between classes A and B, where A is a base class and B is a derived class, we implement class A before class B.
- If class A is a component of class B, that is, there is a part_of relationship between A and B, member functions in class A should be implemented first.

Only partial automation is possible in the transformation from object-oriented design into C++ programs. At several points, user interaction is required. The user

interface is discussed in the next chapter.

Example 5.6

As the last example in this chapter, we present two C++ implementations corresponding to the two designs shown in Example 4.4.

Implementation I (C++ classes derived from OOD I):

```
class banking-system1 {
private:
    map<Accno, Passbook> LEDGER;
    map<Typecode, Minimum> MIN-BALS;
    N1 BANK-BALANCE;
    N1 TAKEN-OUT;
    N1 PUT-IN;
public:
    OPEN(Namestring NEWNAME, Typecode ACCOUNT, N1 AMOUNT)
}

class Passbook{
private:
    N1 ACC-NUM;
    Typecode ACC-TYPE;
    Namestring ACC-NAME;
    N1 BALANCE;
public:
    Namestring GET_ACC_NAME();
}
```

```
class Typecode{
private:
}

class Namestring{
private:
}

class Minimum{
private:
    Typecode ACC-TYP;
    N1 BAL-MINI;
public:
    N1 GET_BAL_MINI()
}
```

Implementation II (C++ classes derived from OOD II):

```
class banking-system2 {
private:
    Accno LEDGER1[N];
    Passbook LEDGER2[N];
    map<Typecode, Minimum> MIN-BALS;
    N1 BANK-BALANCE;
    N1 TAKEN-OUT;
    N1 PUT-IN;
public:
    OPEN(Namestring NEWNAME, Typecode ACCOUNT, N1 AMOUNT)
}
```

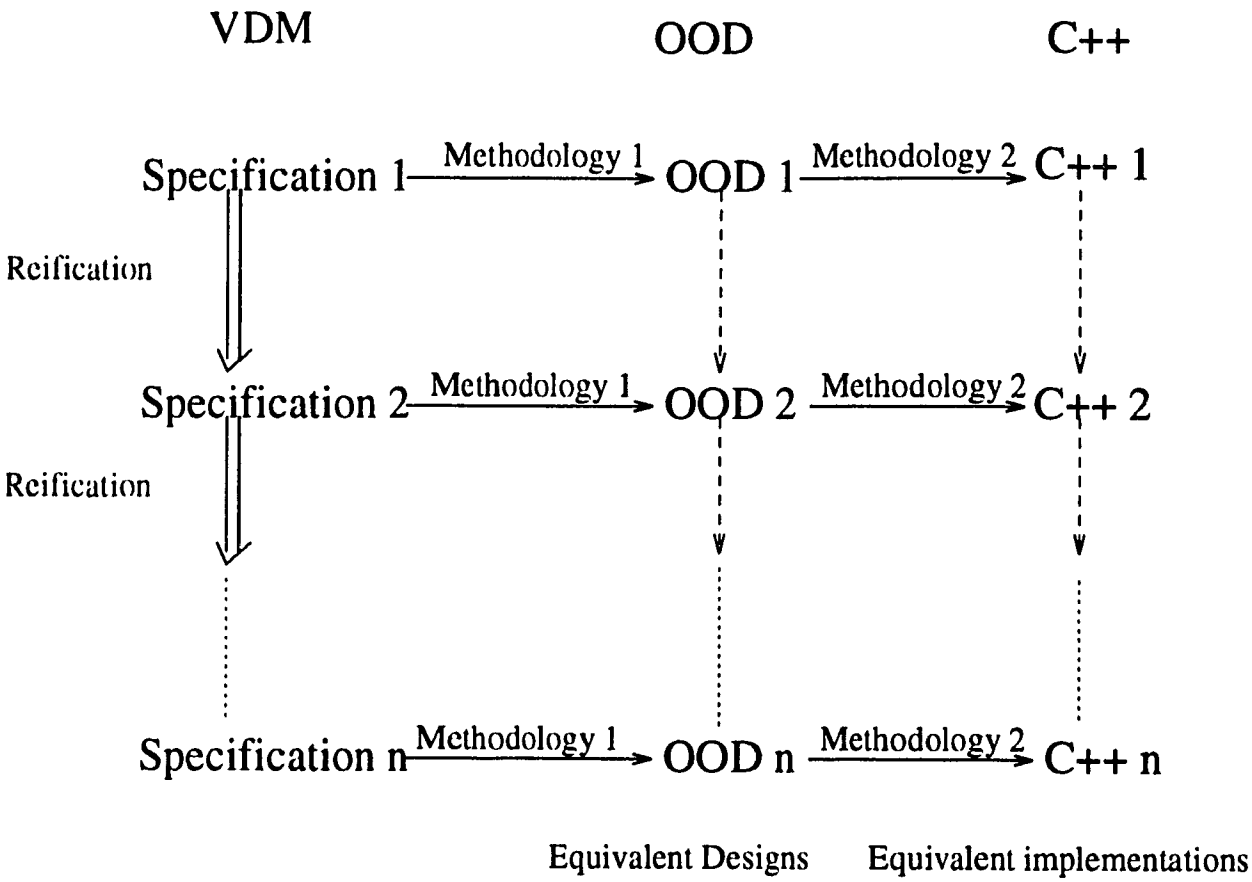
```
class Passbook{
private:
    N1 ACC-NUM;
    Typecode ACC-TYPE;
    Namestring ACC-NAME;
    N1 BALANCE;
public:
    Namestring GET-ACC-NAME();
}

class Typecode{
private:
}

class Namestring{
private:
}

class Minimum{
private:
    Typecode ACC-TYP;
    N1 BAL-MINI;
public:
    N1 GET-BAL-MINI();
}
```

The data members and their types within classes *banking-system1* and *banking-system2* are different. These differences mandate different algorithms for the implementation of the member function *OPEN*. So the final implementations are quite different. However, we believe that these two implementations are equivalent from

the fact that the original VDM specifications are equivalent(one is a refinement of another) and the transformation methodologies produce equivalent designs and implementations.

From Example 3.2, Example 4.1 and Example 5.6 we conclude the following:

- Using methodologies presented in this thesis we can get a series of equivalent OODs and C++ implementations from equivalent specifications. See Figure 5.1.
- The methodologies presented in this thesis not only provide a way to derive OOD and then C++ classes from VDM specification but also reveal equivalent relationship among different object-oriented designs and implementations.
- The methodologies are suitable for different levels of VDM specification.
- If the methodologies are applied to higher level specification, the developer has more choice for data structures and algorithms in the implementation stage.



Methodology1 : From VDM specification to OOD

Methodology2 : From OOD to C++ classes

Figure 5.1: Equivalent Designs and Classes

Chapter 6

User Interface Design

As mentioned in the previous chapters, human interaction is necessary at some points in the development of software systems. A window-based interactive interface is built to support user interaction between the developer and the system. The goal of user interface design is to provide the user an interactive interface through which the developer can provide useful knowledge to improve the design during the transformation, grasp all necessary information, readjust the specification if necessary and finally generate code for the member functions.

6.1 The Significant Aspects of Design Issues

The most fundamental principle in user interface design is that the interface must be designed to suit the needs and abilities of the individual users of the system[36]. The users of our system are software developers with some familiarity with VDM specification, OOD and C++ programming. The important features in the design of user interface in our application are : 1. Different software components, such as specification, design and code that are being developed, can be viewed individually or simultaneously; 2. Multiple windows support the specific needs of the user; 3. Multiple windows provide good system interaction. That is, switching from one window to another does not necessarily mean the complete loss of information generated on the first window.

The user can visualize the following information through several windows:

- An overview of the software system (hierarchy graph) which is under the development.
- The relationship (inheritance and part_of) among the classes derived.
- Member functions and their corresponding operation specification clauses.
- VDM state space.
- C++ programs.

A discussion of detailed design principles of user interface systems is outside the scope of this thesis. Interested readers can refer to [30] for more information. The design principles may often conflict with one another. For example, increasing ease of use might require sacrificing product compatibility across releases. Offering maximum user control may require sacrificing ease of learning. Simplicity may require decreasing power and ease of use. Making these trade-off intelligently requires a thorough understanding of the intended user population. Our design efforts focused on the following aspects :

Ease of Learning and Ease of Use Menu buttons and scrolled lists are two dialog styles mainly used within windows in the system. The major advantage of these two dialog styles are ease of learning and remembering and simplicity of input. Well-designed menu buttons and scrolled lists make the system easy to learn because they make both the *semantics* and *syntax* of the system explicit. That is, they make clear both what can be done (semantics) and how to do it (syntax). The learning burden is decreased and the need for manuals and training programs is reduced. Moreover, since menu and scrolled lists interface are potentially self-explanatory, little human memory is required. It is unnecessary to memorize and retain semantic or syntactic knowledge. For example, as shown in Figure 6.2, the labels on the menu buttons of the main window explain the functionalities of the system. The users can choose a specific function by clicking the associated button without learning and remembering of any specific

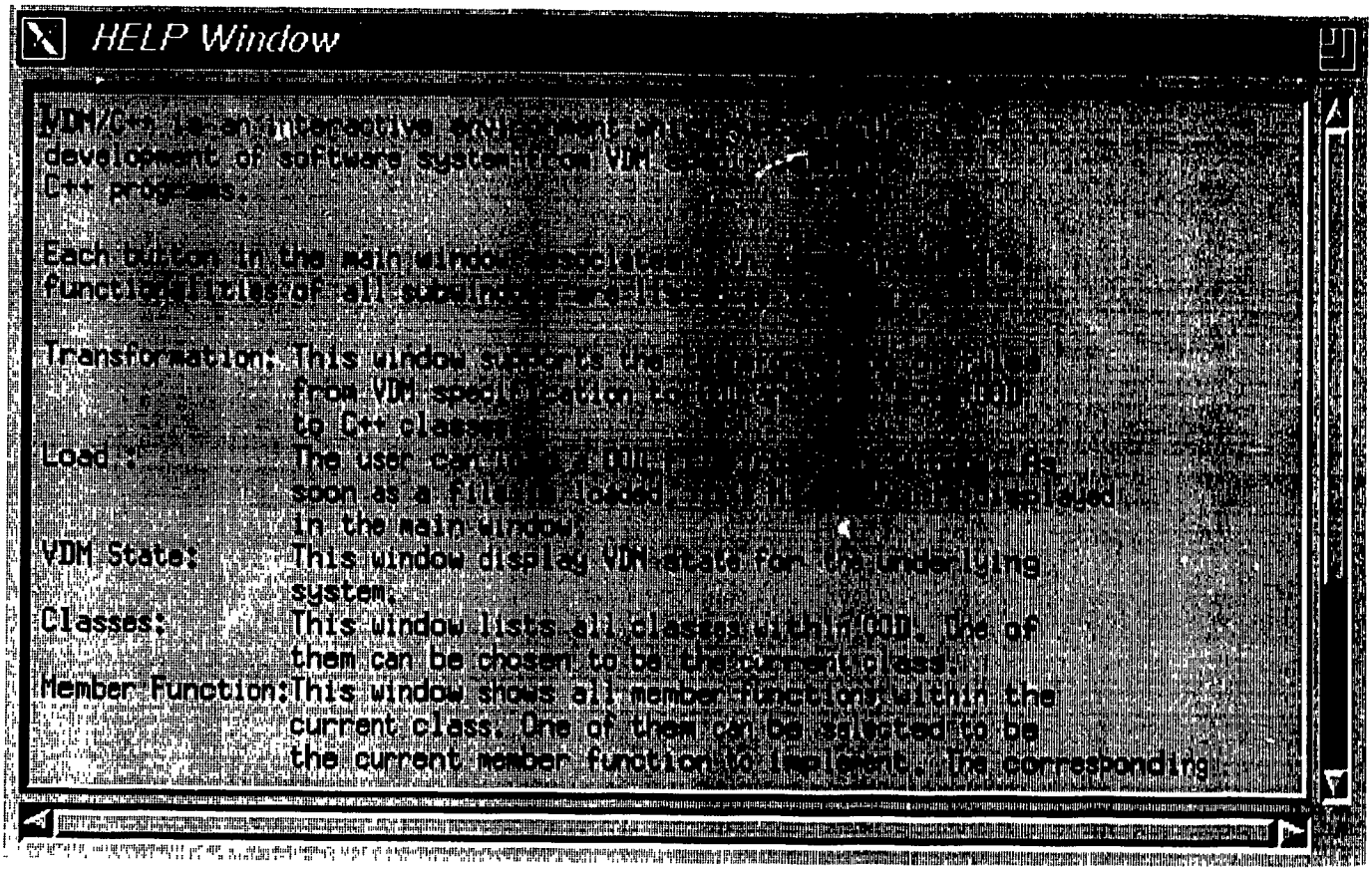


Figure 6.1: On-line help message

semantics or syntax. Furthermore, there is a help button in some windows to provide the on-line help message.The help message presents a detailed description of what functionalities the window has and how to use these functionalities. Figure 6.1 shows the help message for the main window.

Consistency There are two kinds of consistency : the consistency within the system and the consistency across different systems. In our system, some conventions are established for window design and are applied consistently on all windows. For example, window titles, menu buttons and scrollbar etc. always appear in the same location in the window. Colors are used in a consistent way. There is a specific color for any specific window and error message is always displayed in red. All message is consistent in grammatical style. For example, a noun

is used for each menu button. The editor window is exactly the same as the Emacs window under X Window system to make our system to be consistent with other systems.

Familiarity Concepts and terminology that the user is already familiar with are incorporated into the interface, such as some menu button names, *class*, *member function* etc. The hierarchy graph displayed in the main window is also familiar with most of the users. The graph is helpful for the developer to obtain an overall understanding of the system modules which is under development.

Control All messages are positive, polite and concise to make the user feel a sense of mastery and control over the system. Instead of saying "Invalid file name", we indicate "the file can not be found". The former sound like reprimands from the computer. The latter impart more a sense that the computer is a willing slave with limited abilities. User prefer to feel a sense of mastery and control over any tool at their disposal, and computer is no exception.

Simplicity Some default values are set to simplify the system input. In transformation window, the current directory is the default value for directory. If the file name is under the current directory, the user can simply indicate a file name other than the whole path.

Protection There is always a notice for the user to confirm some dangerous actions, such as exit of the system.

6.2 Functionality of Windows

The main window of the system is shown in Figure 6.2. Given a VDM Specification, we can transform it to object-oriented design and then to the C++ header file through the *transformation* window. A specific OOD file can be loaded from the *load* window to implement. As soon as a file is loaded, its hierarchy graph is displayed in the *main* window. From the hierarchy graph, the developer can obtain an overall

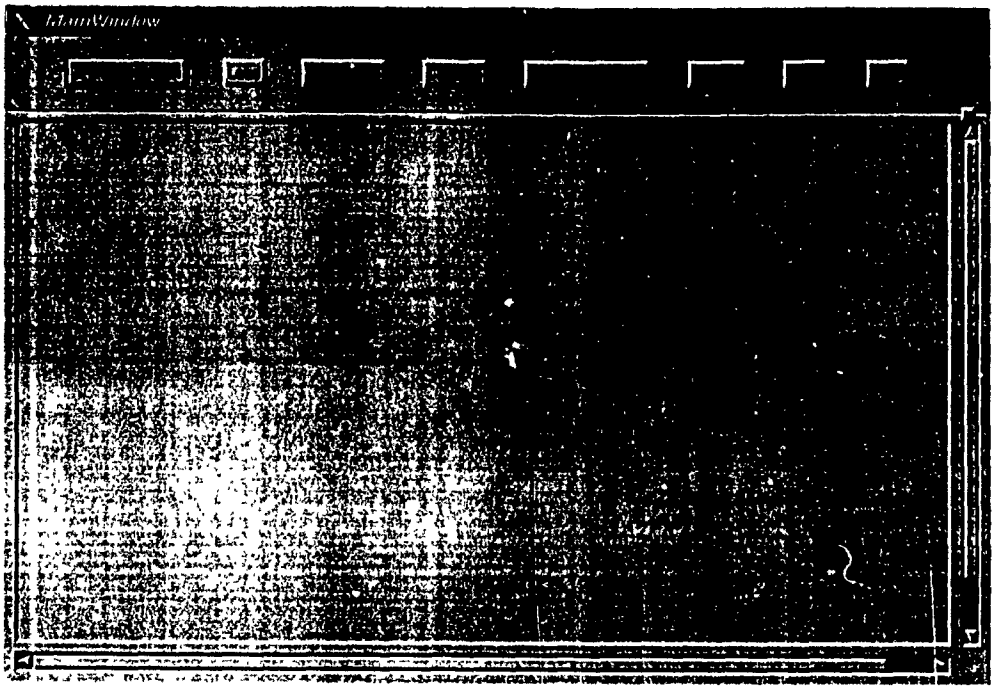


Figure 6.2: Main Window

understanding of the system. We implement the whole system by the implementation of individual member functions within each class. At this stage, the developer has sufficient information to decide which class to implement currently. A current class and then a current member function can be set to implement through the *class* and *member function* window respectively. The *editor* window can be used to edit the code of current member function. The *VDM state* window is helpful for further definition of data types in C++ header file. The user can open a specific window by using the menu buttons. Figure 6.3 to Figure 6.7 display all sub-windows in the system.

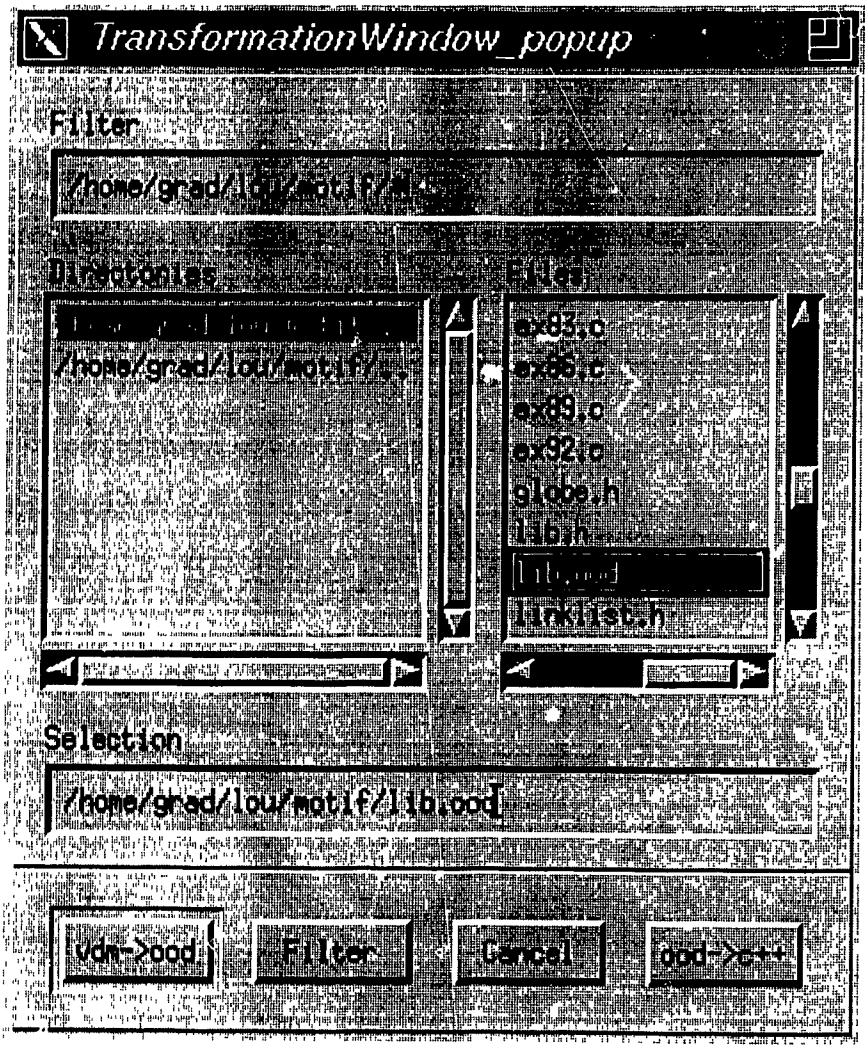


Figure 6.3: Transformation Window

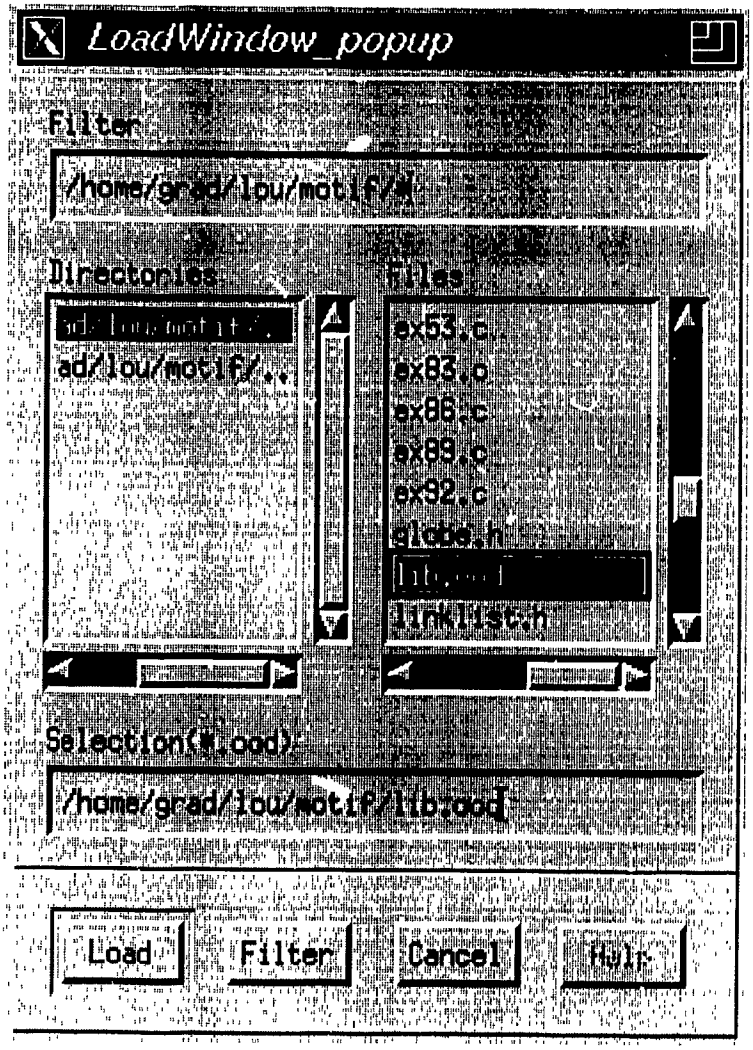


Figure 6.4: Load Window

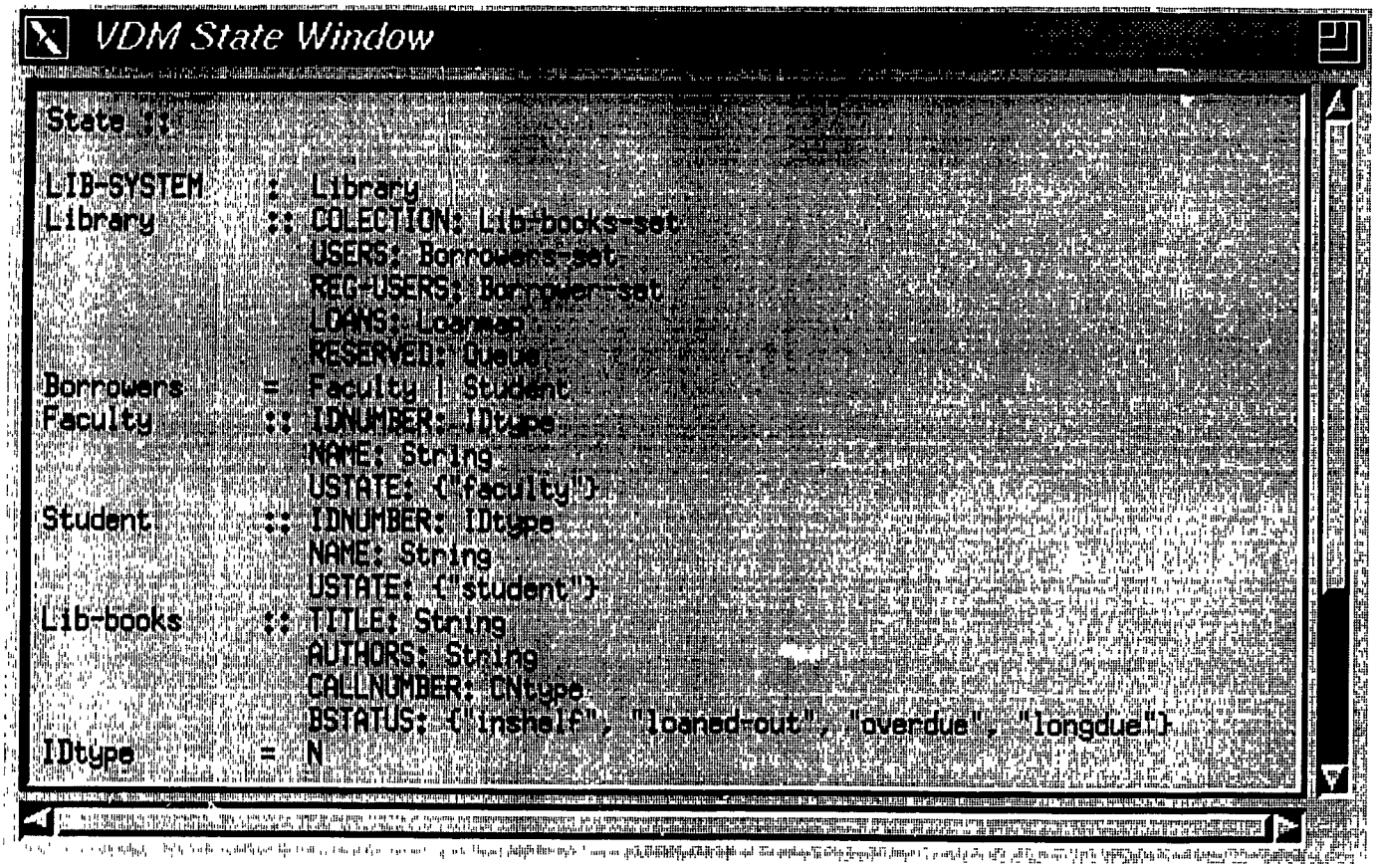


Figure 6.5: VDM State Window

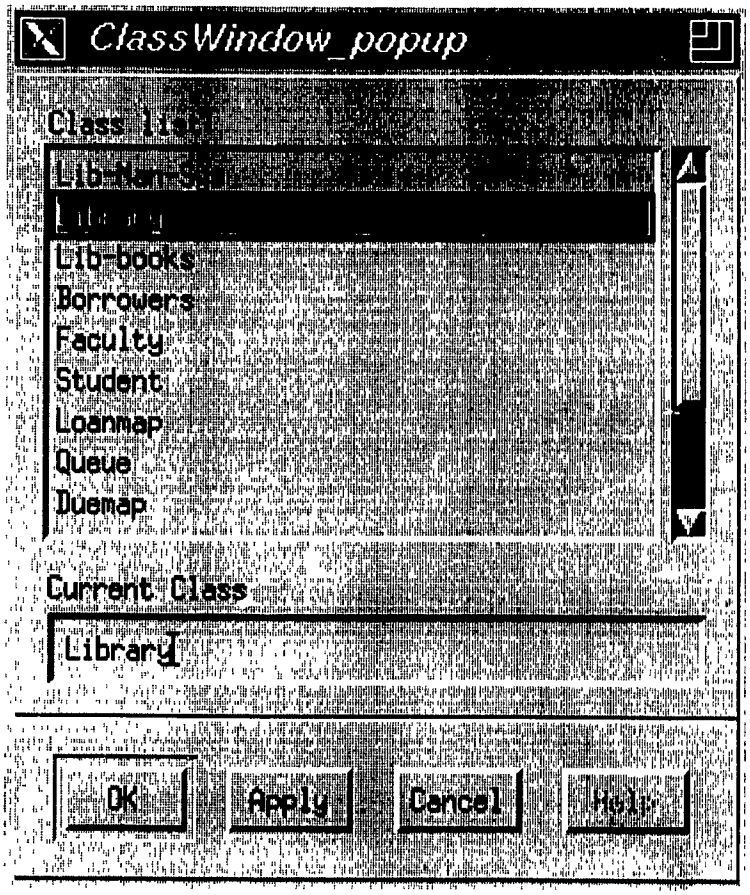


Figure 6.6: Class Window

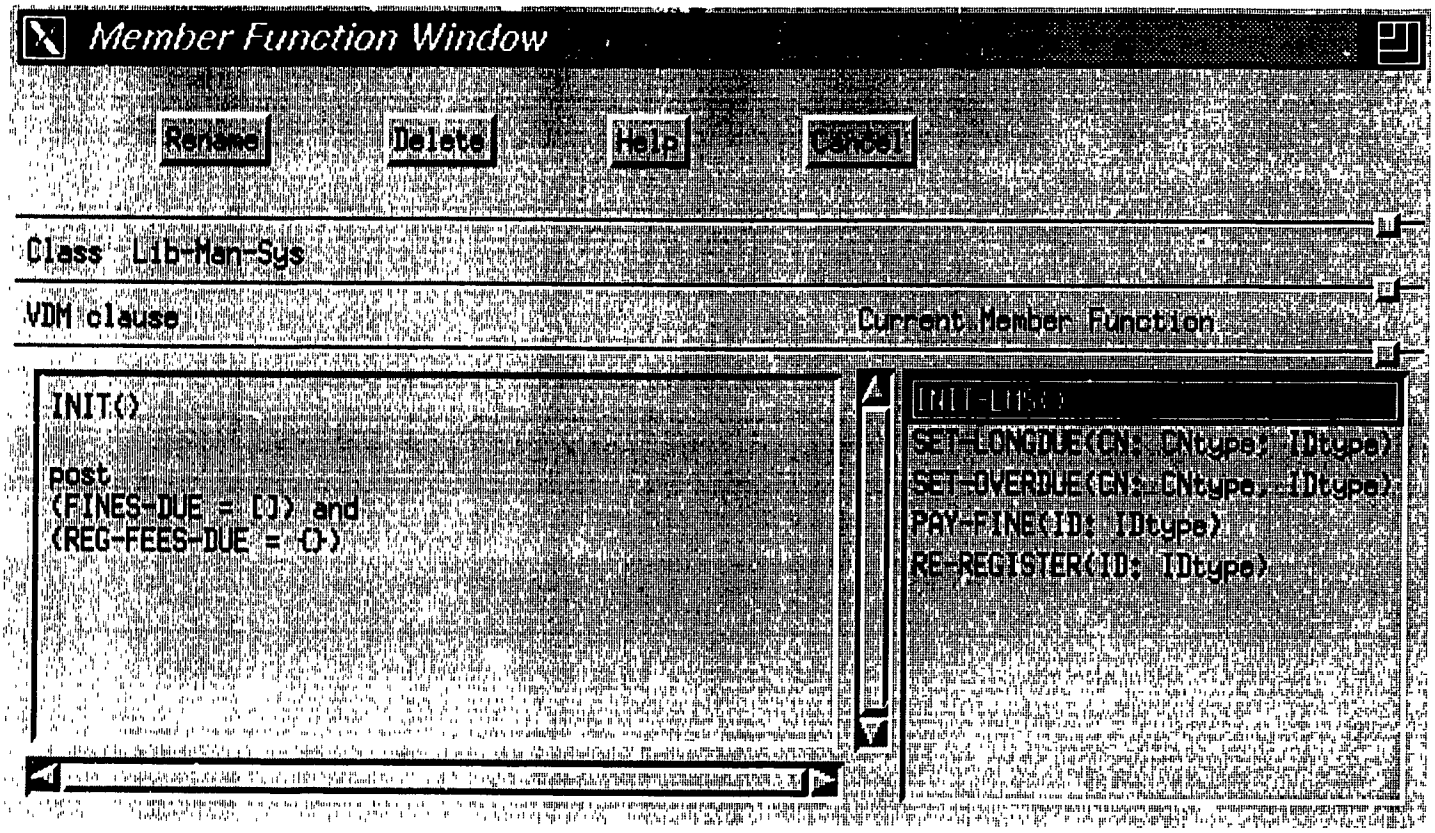


Figure 6.7: Member Function Window

Transformation Window All file transformations, from VDM specification to OOD and from OOD to C++ classes, are completed through this window. As shown in Figure 6.3, the user can select a file name from the file list. The current directory is the default value for the directory. If the file is not under the current directory, both the directory and file name should be given. As soon as the source file name is provided, an associated button may be clicked for the transformation. If the source file doesn't exist, an error message will indicate the error and the user can input another file or push the help button for the on-line help message. The suffix for VDM, OOD and C++ files are .vdm, .ood and .h respectively.

Load Window The Load Window displays all files under the current directory. The user can choose a file from the list or provide another file name. As soon as a file is loaded, its hierarchy graph which reveals the relationship among the classes is displayed in the main window. The current directory is also the default directory. After the indication of file name, the *Load* button may be clicked to load the file. If the file does not exist, an error message will indicate the error and the user can input another file name or push the help button for on line help. There is a scrollbar in the window which can be used to scroll the list of file names on the screen.

VDM State Window The State Window shows the corresponding VDM state in the specification. The scrollbar in the window may be used to scroll the information on the screen. This window can only be opened after a specific OOD file has been loaded. Otherwise, an error message will guide the user to load the OOD file first.

Class Window The Class Window lists all classes in the loaded OOD file. One of them can be chosen to be the current class to implement which is highlighted. The first class in the list is the default current class. The scrollbar can move the class list forward or backward. If there is no OOD file loaded, the creation of the class window will lead to an error message.

Member Function Window The Member Function Window displays all member functions within the current class. The parameters of member functions and their types are all displayed. The user can highlight a current member function to implement. As soon as a current member function is selected, the corresponding VDM predicate clauses is displayed. If the current member function is changed, the VDM clauses will be changed accordingly. The current member function can be renamed or deleted through the *rename or delete* button in the window. If this is done, all related parties will be informed.

Figure 6.8 is a typical screen during the development of the library management system described in Appendix B. Given a VDM specification file(lib.vdm), we first create the transformation window to transform the VDM file to OOD file(lib.ood), next the OOD file is transformed to C++ header file (lib.h) by indicating the source file name, and clicking the corresponding menu button. After the transformation, the window may be closed to save the screen space. The next step is to load the OOD file (lib.ood) for implementation through the load window. As soon as the file is loaded, the user can open the class window to get a list of all classes in the design and select a current class to implement. In the example, *Library* is the current class chosen. To implement a class, we implement all member functions within the class. The member function list can be obtained by clicking the member function button in the main window. From this window, the developer may select a current member function (ADD-BOOK() in the example) to implement. Figure 6.8 shows all windows on the screen at this stage. The development of the current member function can be completed through the Editor window later.

As shown in Figure 6.8, the VDM clauses displayed actually define the meaning of current member function; that is, they provide the semantics of the member function. The clauses in this example means that the member function *ADD-BOOK()* should get a new call number for the book and then add the book to the library collection. Through the Editor window, the developer can load a

file (lib.c) to implement the current member function according to the meaning of those clauses. The C++ code for member function *ADD-BOOK()* can be as follows.

```
void ADD_BOOK(String TIT, String AUT){
CNtype cn;
Lib_books *book;
cn = (*book).NEW_BOOK(); /* get a new call number */
book = new Lib_books(TIT, AUT, cn, inshelf); /* create a new book */
collection.add(book); /* insert the book into library collection */
}
```

NEW_BOOK() is one member function in class *Lib-books* which returns a new call number. And *add* inserts a new element to the set *collection*.

It is clear from this example that given a member function and it's corresponding VDM clauses, the developer can implement the member function without much difficulty.

Editor Window The developer implements all member functions through the Editor Window. The C++ classes file which is obtained from OOD can be loaded and all member functions in the classes can be implemented according to the information in the specification window.

As seen so far, it is possible for the developer to simultaneously view the various phases of the software development such as the specification, design and code. The ultimate goal of this user interface is to synchronize the activities in various windows so that the developer can instantaneously monitor the changes in one window (say, in the design) due to a change in the other window (say, in the specification).

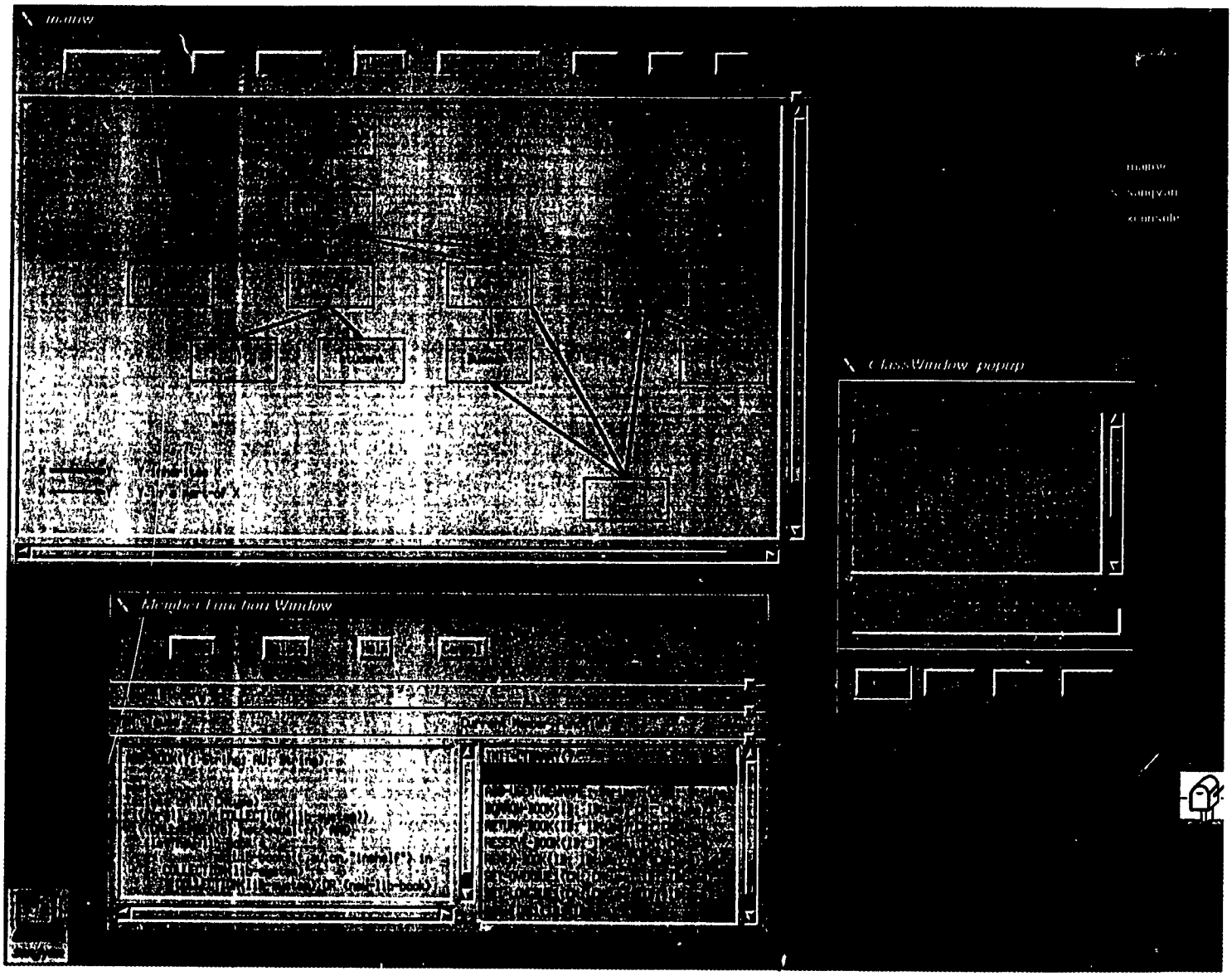


Figure 6.8: A Typical Screen

6.3 VDM Specification of User Interface Design

In this section, we present a VDM specification for the user interface described in the previous section. The concrete notation and terminology used in VDM has evolved continuously and is expected to change further. All notations used here are introduced from [1]. The notation X' in the post-condition denotes the value of globe variable X after the operation; the value of the same variable before the operation is written without the prime.

6.3.1 Global Variables

The user interface maintains all windows currently on the screen; VDM, OOD and C++ files; all classes within an OOD; the current class which is chosen to implement; a map from member function to VDM specification clauses; and a map from OOD file name to the corresponding hierarchy graph. The following global variables are used to model the system. The functionalities of windows can be stated formally by the description of all operations on these global variables.

CLASSES This variable represents all classes in an OOD which is loaded currently. Every class consists of a class name which is unique, all member functions and member variables in the class and the name of the current member function.

CURRENT This variable describes the class in which member functions are chosen for implementation.

VDM This variable is the collection of all maps that associates with each member function in a OOD class to the VDM specification from which it was derived.

HIERARCHY This variable maps OOD file names to their corresponding hierarchy graphs which reveal the inheritance and part-of relationship between classes in the design.

WINDOWS This variable records all windows on the screen. Each window is composed of a window name which is unique, the position on the screen, the color

and all buttons within the window.

VDMSTATE This variable represents the corresponding VDM state space for the loaded OOD.

VDMFILE This variable denotes specification files. Each of them specifies a individual system which may be developed under VDM/C++.

OODFILE This variable represents all OOD files which are converted from a VDM file through the transformation window.

C++FILE This variable records all C++ header files which are obtained from a OOD file through the transformation window.

The formal VDM state definition is given next.

6.3.2 VDM State Space

STATE ::

CLASSES : Cmap

Cmap = String $\rightarrow$ Class

/*string is the name of class. every class has a unique name*/

CURRENT : String /* the current class */

Class ::

NAME : String /*the name of the class*/

MFS : String $\rightarrow$ Member_function

/*for each member function in the class, there is a map from the member function name to itself*/

MVS : Variable-set /* all member variables in the class */

CURRENT_MF : String

/*the current member function which is chosen for implementation*/

Member_function ::

NAME : String /* member function name */

TYPE : Types /* the return type of member function */

PARAMETER : Variable-set/*all parameters for the member function

Variable ::

NAME : String /*the name of the variable*/

TYPE : Types /*the type of the variable*/

VDM : String → String-list

/*mapping from name of member function to the corresponding VDM specification */

String = /*set of legal strings which may be defined in the later stage*/

Types = /* set of all types which may be defined in the later stage*/

HIERARCHY : String → Graphs-set

/* from OOD file name to the corresponding hierarchy graph*/

Graphs ::

NODE : Rectangle /* one node in the hierarchy*/

LS : Line → Rectangle

/* lines from the described node to all it's child nodes */

Line ::

P1 : Point

P2 : Point

Rectangle ::

P1 : Point /*the low left corner*/

P2 : Point /*the upper right corner*/

NAME : String /* the string in rectangle */

WINDOWS : Wmap /* all windows on the screen */

Wmap = Wname → Window

/* mapping from the name of the window to the window*/

Window ::

NAME : Wname /*the name of the window*/

POSITION : Position /*the position of window */

COLOR : Color /*the color of the window*/

BUTTON : Button-set /* all buttons within the window */

Wname = {main_w, transformation_w, load_w, state_w, classes_w, member_function_w, editor_w, help_w}

/* the set of all window names */

Position ::

LLC : Point /*left low corner*/

URC : Point /*right upper corner*/

Point ::

X : integer /* x axis*/

Y : integer /* y axis*/

Color = /* set of all colors which may be defined in later stage */

Button = /* set of all buttons which may be defined in later stage */

VDMFILE : String-set

OODFILE : String-set

C++FILE : String-set

VDMSTATE : String-list

6.3.3 Operations

This subsection presents the specification of all operations in VDM. Each operation implements a menu button or a specific function for a window described in Section 6.2. The implementation of all operations described here will fulfill the functionalities of windows mentioned in Section 6.2.

```
CREATE_MAIN(POSI : Position; COL : Color)
/* create the main window on the screen */
ext
      WINDOWS : wr Wmap
pre
      /*the main window should be created first*/
      dom windows = Φ
post
      /*the main window is created*/
     let button={transformation,load,state,classes,member_function,editor,help,quit} in
        let w=mk_Window(main_w,posi,col,button) in
            windows' = windows †[main_w → w]
        tel
     tel
err
      dom windows ≠ Φ → windows' = windows
```

```
CREATE_TRANSFORMATION(POSI : Position; COL : Color)
/*create the transformation window */
ext
      WINDOWS : wr Wmap
pre
      /* the main window has been created */
       main_w ∈ dom windows
post
      /* the transformation window is created */
      let w=mk_Window(transformation_w,posi,col,
                  {vdm->ood, ood->c++, Filter, Cancel}) in
          windows' = windows †[transformation_w → w]
      tel
err
      ¬ (main_w ∈ dom windows) →
      windows' = windows
```

```
VDM_TO_OOD(NAME1: String; NAME2 : String)
/* transform a vdm file to ood file */
/* name1 is the VDM file name and name2 is the OOD file name */
ext
      VDMFILE : rd String-set
      OODFILE : wr String-set
      WINDOWS : rd Wmap
pre
      /*the VDM file exists and the transformation window has been created*/
      (name1 ∈ vdmfile)∧ (transformation_w ∈ dom windows)
post
      /*the OOD file is created*/
      oodfile' = oodfile ∪ {name2}
err
      ¬(name1 ∈ vdmfile) ∨ ¬(transformation_w ∈ domwindows) →
      oodfile' = oodfile
```

OOD_TO_C++(NAME1: String; NAME2 : String)
/* transform a OOD file to C++ file */
/* name1 is the OOD file name and name2 is the C++ file name */
ext
OODFILE : **rd** String-set
C++FILE : **wr** String-set
WINDOWS : **rd** Wmap
pre
/*the OOD file exists and the transformation window has been created*/
(name1 ∈ oodfile)∧ (transformation_w ∈ dom windows)
post
/*the C++ file has been created*/
$c++file'$ = c++file ∪ {name2}
err
$\neg(name1 \in oodfile) \vee \neg(transformation\_w \in dom windows) \rightarrow$
$c++file'$ = c++file

CREATE_LOAD(POSI : Position; COL : Color) F: String-set
/*create the load window and display all ood files */
ext
WINDOWS : **wr** Wmap
OODFILE : **rd** String-set
pre
/* the main window has been created */
main_w ∈ dom windows
post
/*the load window is created and all ood files is displayed*/
let w=mk_Window(load_w,posi,col,{Load,Filter,Cancel}) **in**
$windows'$ = windows †[$load\_w \rightarrow w$]∧
f = oodfile
tel
err
$\neg(main\_w \in$ dom windows$) \rightarrow$
$(windows' = windows) \wedge (f = nil)$

```
CREATE_STATE(POSI : Position; COL : Color)S : String-list
/*create the VDM state window and display VDM state space*/
ext
      WINDOWS : wr Wmap
      VDMSTATE : rd String-list
pre
     /* the main window has been created */
     main_w ∈ dom windows
post
     /*the state window is created and the VDM state is displayed*/
     let w=mk_Window(state_w,posi,col,{ }) in
         windows' = windows †[state_w → w]∧
         s = vdmstate
     tel
err
     ¬(main_w ∈ dom windows) →
     (windows' = windows) ∧ (s = nil)
```

CREATE_CLASS(POSI : Position; COL : Color)CLA : String-set
/*create the classes window and display all classes in the system*/
ext
WINDOWS : **wr** Wmap
CLASSES : **rd** Cmap
pre
/* the main window has been created */
main_w ∈ dom windows
post
/*the class window is created and all classes is displayed*/
let w=mk_Window(classes_w,posi,col,{OK, Apply, Cancel}) **in**
windows' = windows †[*classes_w* → *w*]∧
cla = dom classes
tel
err
¬(*main_w* ∈ dom windows) →
(*windows'* = *windows*) ∧ (*cla* = *nil*)

SET_CURRENT_CLASS(NAME : String)
/* set the current class to implement*/
ext
CURRENT : **wr** String
CLASSES : **rd** Cmap
pre
/*the class window has been created and the name is a class name in OOD*/
(*name* ∈ dom classes) ∧(*classes_w* ∈ dom windows)
post
/*the given class becomes the current class*/
current' = name
err
¬(*name* ∈ dom classes) ∨¬(*class_w* ∈ dom windows) →
current' = *current*

CREATE_MEMBER_FUNCTION(POSI:Position; COL:Color)MF: Member function set
/*create member function window and display all member functions of current class */
ext
WINDOWS : **wr** Wmap
CURRENT : **rd** Class
pre
/*the main window has been created,and the current class has been set */
(main_w ∈ dom windows) ∧ (*current* ≠ *nil*)
post
/*the member function window is created */
/*and all member functions in the current class is displayed*/
let w=mk_Window(member_function_w,posi,col,{Rename,Delete,Help,Cancel}) **in**
windows' = *windows* † [*member_function_w* ⟶ *w*]∧
mf = rng(MFS(classes(current)))
tel
err
¬(*main_w* ∈ dom windows) ∨ (current = nil) ⟶
(*windows'* = *windows*) ∧ (*mf* = *nil*)

```
SET_CURRENT_MEMBER_FUNCTION(N : String)VMS-S: String-list
/' set the current member function */
ext
      CURRENT : rd String
      CLASSES : wr Cmap
pre
      /'n is a member function in current class, but not the current one */
      (n ∈ dom MFS(classes(current))) ∧(n ≠ CURRENT_MF(classes(current)))
post
      /'the current member function is set*/
      let new-class = mk_class(NAME(classes(current)),
                        MFS(classes(current)), MVS(classes(current)), n) in
         classes' = classes † [current → new-class] ∧
         vdm-s = vdm(n)
      tel
err
      ¬(n ∈ dom MFS(classes(current))) ∨ (n = CURRENT_MF(classes(current)))→
      classes' = classes∧ vdm-s = nil
```

```
RENAME(OLDNAME : String; NEWNAME : String)
/* rename the current member function */
ext
      CLASSES : wr String → Class
      CURRENT : rd String
pre
      /*old name is the current member function in current class*/
      /*and newname is not a member function name in current class*/
      ¬(newname ∈ dom MFS(classes(current))) ∧
      (oldname = CURRENT_MF(classes(current)))
post
   /*rename the member function in current class and revise the class set accordingly*/
   let old_mf=MFS(classes(current))(oldname) in
      let new_mf=mk_Member_function(newname,TYPE(old_mf),
                        PARAMETER(old_mf)) in
         let new_mfs=MFS(classes(current))†[newname→new_mf] in
            let new-class = mk_Class(NAME(classes(current)), new_mfs,
                        MVS(classes(current)), newname) in
               classes' = classes † [NAME(classes(current)) → new-class]
            tel
         tel
      tel
   tel
err
      (newname ∈ dom MFS(classes(current))) ∨
      (oldname ≠ CURRENT_MF(classes(current))) →
      classes' = classes
```

```
DELETE_MF(NAME : String)
/* delete the current member function */
ext
      CLASSES : wr String → Class
      CURRENT : rd String
pre
      /*the given name is the current member function name*/
      name = CURRENT_MF(classes(current))
post
      /*delete the member function from the current class*/
      /*and revise the class set accordingly*/
      let old_mfs = MFS(classes(current)) in
         let new_mfs = {name} ⩤ old_mfs in
            let new_class = mk_Class(NAME(classes(current)), new_mfs,
                        MVS(classes(current)), nil) in
               classes' = classes † [NAME(classes(current)) → new_class]
            tel
         tel
      tel
err
      name ≠ CURRENT_MF(classes(current)) →
      classes' = classes
```

```
SHOW_HIERARCHY(NAME : String)H : Graphs-set
/* show the hierarchy for the loaded OOD */
ext
      WINDOWS : rd Wmap
      HIERARCHY : rd String → Graphs-set
pre
      /*the main and load window are all created*/
      (load_w ∈ dom windows) ∧ (main_w ∈ dom windows )
post
      /*the hierarchy graph is displayed*/
      h = hierarchy(name)
err
      ¬(load_w ∈ dom windows ) ∨¬ (main_w ∈ dom windows )  →
      h=nil
```

```
CREATE_EDITOR(POSI:Position; COL:Color )
/* create the editor window */
ext
      WINDOWS : wr Wmap
pre
      /* the main window has been created */
      main_w ∈ dom windows
post
      /*the editor window is created*/
      let w=mk_Window(editor_w,posi,col,{Buffers,File,Edit,Help }) in
          windows' = windows† [editor_w→ w]
      tel
err
      ¬(main_w ∈ dom windows) →
      windows' = windows
```

CLOSE(N : Wname)
/* close a given window */
ext
WINDOWS : **wr** Wmap
pre
/* main window should be closed at last */
if n=main_w then (n ∈ dom windows) ∧ (card (dom windows)=1)
else n ∈ dom windows
post
/*the window is closed*/
$windows' = \{n\} \lhd windows$
err
$\neg(n \in$ dom windows$) \vee (n = main_w \wedge$ card (dom windows) $\neq 1) \rightarrow$
$windows' = windows$

INIT()
/*initialize global variables*/
ext
WINDOWS : **wr** Wmap
CURRENT : **wr** String
OODFILE : **wr** String-set
C++FILE : **wr** String-set
post
$windows' = [\,] \wedge$
$current' =$ nil $\wedge$
$oodfile' = \{\} \wedge$
$c++file' = \{\}$

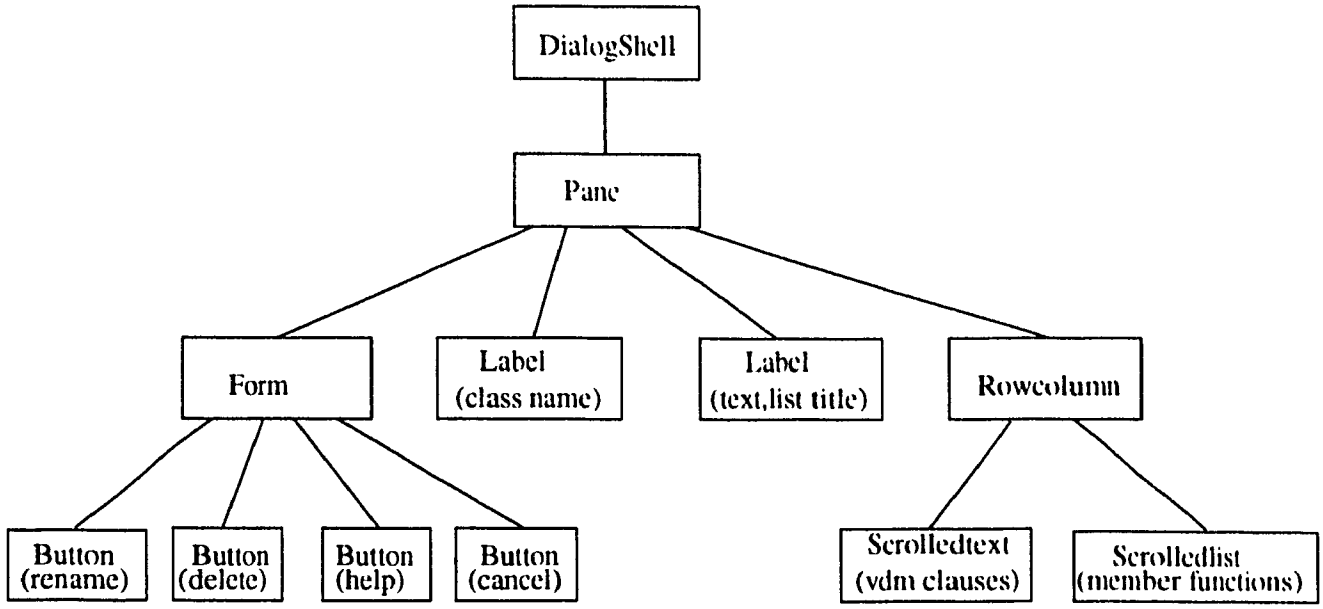


Figure 6.1: Widgets in Member Function Window

6.4 Implementing the User Interface Design

The VDM specifications presented in the previous section have been implemented using Motif toolkit from the Open Software Foundation(OSF). The Motif toolkit is developed using X as window system and X toolkit Intrinsics (also known as Xt) as the platform for the application programmer's interface. Motif provides a complete set of widgets designed to implement applications. We use these motif widgets as building blocks to construct individual windows described in Section 6.2. For example, Figure 6.1 shows all widgets used to implement the *member function window* and the relationship among these widgets. Some Xlib functions are used to draw the hierarchy in the main window.

We believe that the implementation is faithful with respect to the specification, although it has not been proved formally. The reasons are as follows.

- All operations in the specification are implemented.
- All pre- and postconditions have been checked manually.

Example 6.1

Consider the VDM operation *SET-CURRENT-CLASS*. The following procedure is used to implement the operation.

```
void SetCurrentClass(w, data, cbs)
Widget w;
XtPointer data;
XmSelectionBoxCallbackStruct *cbs;
{
char *value;
/* get the current class from CallbackStruct */
XmStringGetLtoR(cbs->value, XmSTRING_DEFAULT_CHARSET, &value);
/* set the current class */
CurrentClass = value;
}
```

The preconditions are ensured by the fact that the current class is selected from a list in class window (the class window should exist at this time) and the class window only lists those classes in the design (the new selected class should be one of them). The postcondition is ensured by the statement

```
CurrentClass = value;
```

The user interface can be easily extended to incorporate some other tools. Suppose we want to include a new tool which transforms a semi-formal specification to VDM specification. We can simply add a button in the transformation window to support this new functionality. We only revise the procedure which deals with the transformation window. All other parts remain unchanged. A detailed discussion about such extensions to this thesis work is provided in the next chapter.

Chapter 7

Conclusion and Further Work

In this thesis we presented an interactive integrated software development environment which supports the transformation from specifications to implementations. The methodologies provided are both feasible and practical. VDM specification not only provides formal language to support for data abstraction and system modeling, but also provides clues to implementations. We discussed some strategies on how to exploit these information throughout the development process. The *reification* concept in VDM reveals the equivalent relationship among designs and implementations.

VDM/C++ supports object-oriented design paradigms and provides C++ design details, classes and their data member, member functions as well as relationships among the classes. It is always a challenging problem for researchers to demonstrate links between formal specifications and programming language implementations. VDM/C++ provides a partial solution to this problem. Besides, VDM/C++ also supports many features that traditional techniques do not provide, such as object-orientedness and reusability. We have not given formal semantics for VDM/C++; however, the experience gained in this work sets the stage for developing formal semantics.

The user interface for the system supports human interaction to the development process; such a human interaction is unavoidable for the transformation process explained in this thesis. Through the user interface, the developer provides his or her own knowledge which is combined with the system's knowledge to achieve higher quality software production.

Both the object-oriented design and user interface discussed in this thesis are language independent. They can be used to work with some other object-oriented language such as Smalltalk. There are some directions in which further work can be done to improve the system:

- Integrating VDM/C++ framework with Mural [24] system to ensure the correctness of specification.
- Providing a knowledge-based support to assist user interaction.
- Incorporating tools which support the transformation of a semi-formal software description to a VDM specification to automate the formation of specification.
- Extending the user interface to support new tools.

The Mural System [24] consists of two parts, a VDM support tool and a proof assistant. Both components work together to provide support for the construction and refinement of VDM specifications and for the proof of the associated proof obligation. Besides, the Mural System interface provides a window-based environment to access both VDM support tool and the proof assistant.

The proof assistant provides a way of creating and storing mathematical *theories* hierarchically such that information stored in one theory can be inherited by other theories. Each theory consists of a *signature*, a set of *axioms* and a set of rules. The signature records the declarations of the symbols which can be used to build valid mathematical expressions in the theory, whilst the axioms record the 'primitive' properties of these symbols, that is those properties which are accepted as being true without proof. All theory's rules present additional properties of the symbols which require proof. These properties may be proved using axioms and other rules. The reasoning power of Mural System can be extended either by adding new theories or by adding new rules to existing theories.

The VDM support tool provides facilities for creating and storing specifications and reifications between specifications in VDM. Related specifications and their as-

sociated reifications are grouped together as *developments*, which can be added, accessed, renamed and removed via the interface. In Mural System, a specification or a reification in VDM support tool can be translated automatically into a theory supporting reasoning about it in the proof assistant.

D'Almeida [12] presents a semi-formal description mechanism which contains characteristics of both informal and formal method and a transformation system. The semi-formal specification is described by a Modified Entity-Relationship(MER) model and the Keyword-based Formatted Description(KFD). The MER model, which is a modification of the well-known Entity-Relationship approach to data modeling [11], describes the different system entities and the relationship among them. The KFDs describe functional requirements of the intended software system as textual descriptions using keywords. The transformation system is composed of a set of rules which transform the MER and KFDs into a formal VDM specification.

The output from transformation system can be the input to the VDM/C++ environment discussed in this thesis. A syntax checker, a semantics analyzer and a proof assistant should be interfaced to ensure that only correct VDM specification are input to VDM/C++ environment. The knowledge base in the system will be helpful at each stage of software development. And the user interface discussed in Chapter 6 can be extended to support human interaction in all phases. Figure 7.1 shows the system architecture of the whole integrated development from stating informal requirements to implementation.

Incorporated with all these suggested enhancement, VDM/C++ will provide an environment supporting the development of software system from semi-formal specification, which is easy to learn and understand, all the way to implementation. The quality and reliability of the final products can be assured only by the correctness of all transformation methodologies and the correctness of VDM specification which is in turn assured by the Mural System. All tools in the environment will help the system developer in all phases of the software development life-cycle to diminish the burden of such tasks and therefore increase the productivity of software development.

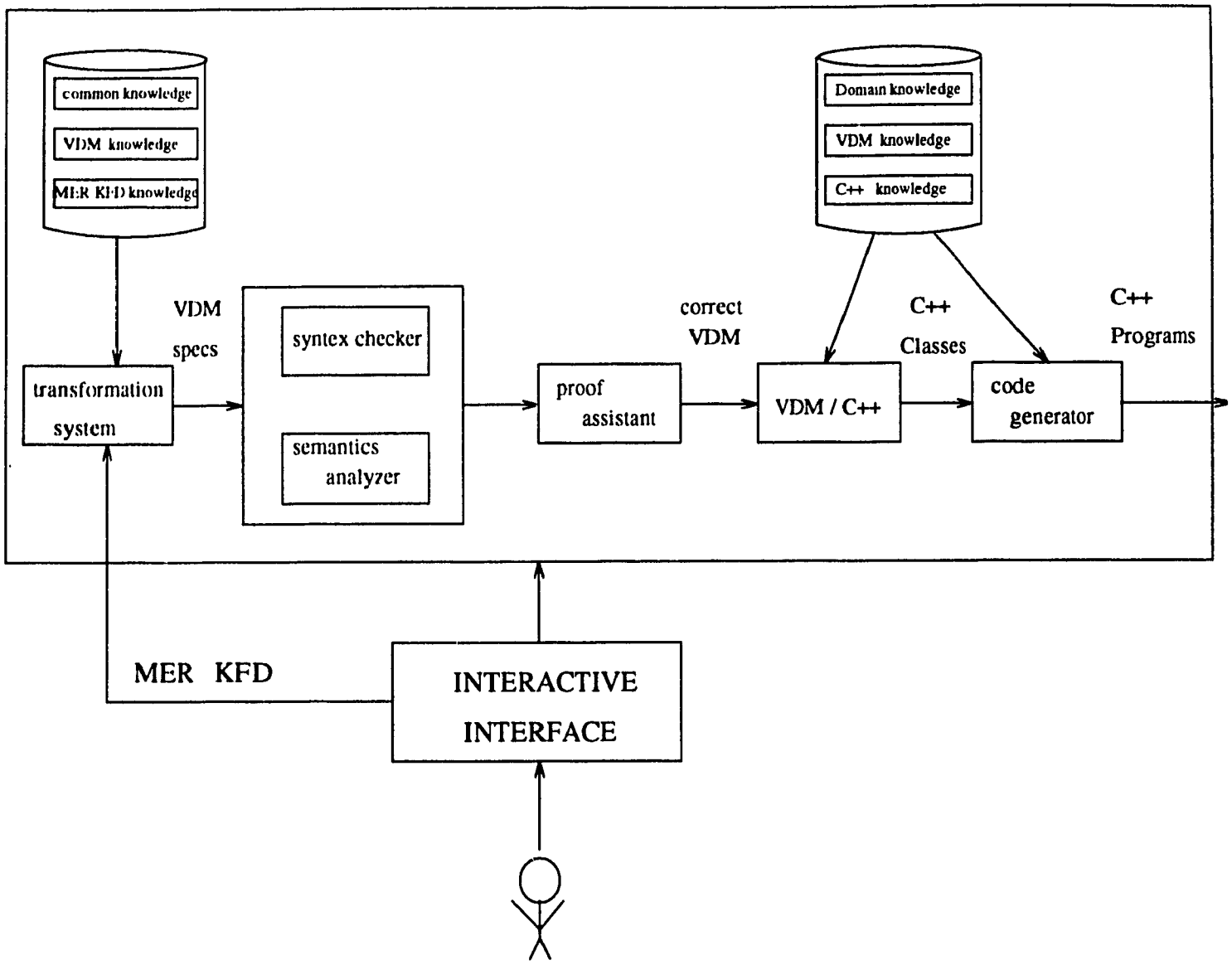


Figure 7.1: System Architecture for the Integrated Software Development

Bibliography

[1] V.S. Alagar, "Specification of Software Systems", *Lecture Notes*, Department of Computer Science, Concordia University, Montreal, Canada, 1991, Revised 1993.

[2] V.S. Alagar and K. Periyasamy, "A Methodology for Deriving an Object-Oriented Design from Functional Specifications", *Software Engineering Journal*, Vol 7, No 4, July 1992, pp. 247–263.

[3] B. Alabiso, "Transformation of Data Flow Analysis Models to Object-Oriented Design", *ACM SIGPLAN Notice*, **23**, 11, pp. 335-353, 1988.

[4] J.G.P. Barnes, *Programming in Ada*, 4th Edition, International Computer Science Series, 1994.

[5] K. Beck and H. Cunningham, "A Laboratory for Teaching. Object-Oriented Thinking", In *Proceedings of OOPSLA 1989*, New Orleans, pp.1-6, 1989.

[6] G. Booch, *Software Engineering with Ada*, 2nd ed. Benjamin/Cummings, 1987b.

[7] R. Breu, "Algebraic Specification Techniques in Object Oriented Programming Environment", *Lecture Notes in Computer Science - 562*, Springer-Verlag, 1991.

[8] B.Cohen, W.T.Harwood and M.I.Jackson, "The Specification of Complex Systems", Addison-Wesley Publishing Company, 1986.

[9] B.J. Cox, *Object Oriented Programming - An evolutionary Approach.*, Reading, MA: Addison-Wesley, 1986.

[10] O.J. Dahl, "Object Orientation and Formal Techniques", In *VDM'90 VDM and Z Formal Methods in software Development*, Bjoner, D., Hoare, C.A.R. and Langmaack, H. (Eds.), *Springer-Verlag*, pp.1-11, 1990.

[11] B. C. Desai, *An Introduction to Database System*, West Publishing co. 1990.

[12] J. D'Almeida, "On the Transformation of a Semi-formal Software Description to a VDM Specification", Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1992.

[13] J. Dawes, *The VDM-SL Reference Guide*, Pitman, 1991.

[14] B. Henderson and J.M. Edwards, "The object-oriented systems life-cycle", *Communications of the ACM*, Vol 33, No 9, Sep. 1990, pp. 142-159.

[15] J. Dick, "VDM Methodology Guide", Bull System Products, 1992.

[16] B. Eckel, *C++ inside & out*, McGraw-Hill, 1993.

[17] P.M. Ferguson, *Motif Reference Manual*, O'Reilly & Associates, Inc. 1993

[18] G. Eriksson and P. Holm, *Programming in Simula*, Lund(Sweden): KF-Sigma, 1984.

[19] J.M. Gao, "GAP: a tool for transformation from VDM specification to object-oriented design", Master's Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1992.

[20] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Reading, MA: Addison-Wesley, 1983.

[21] D.Heller, *Motif Programming Manual*, O'Reilly & Associates, Inc. 1991

[22] I. Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley Publish Company, 1992.

[23] C.B. Jones, *Systematic software development using VDM*, (Second Edition) Prentice Hall, 1989.

[24] C.B. Jones, K.D. Jones, P.A. Lindsay, R. Moore, *Mural: A Formal Development Support System*, Springer-Verlag, 1991.

[25] G.T. Leavens and Y. Cheon, "Preliminary Design of Larch/C++",*Proceedings of the First International Workshop on Larch*, 1992.

[26] S.B. Lippman, *C++ Primer* (Second Edition), Addison-Wesley Publishing Company, 1992.

[27] S. Lipschutz, *Theory and Problems of Set Theory and Related Topics*, Schaum Outline Series, McGraw-Hill, 1964.

[28] W.R. LaLonde and J.R. Pugh, *Inside Smalltalk*, vol I. Englewood Cliffs, NJ: Prentice Hall, 1990.

[29] W.R. LaLonde and J.R. Pugh, *Inside Smalltalk*, vol II. Englewood Cliffs, NJ: Prentice Hall, 1991a.

[30] D.J. Mayhew, *Principles and Guidelines in Software User Interface Design*, Prentice Hall, 1990.

[31] B. Meyer, *Object-Oriented Software Construction*, Englewood Cliffs, NJ: Prentice Hall, 1988.

[32] C. Minkowitz and P. Henderson, "A Formal Description of Object-Oriented Programming using VDM", *Lecture Notes in Computer Science - 252*, Springer-Verlag, 1987.

[33] D. O'Neill, "VDM Development with Ada as the Target Language", *Lecture Notes in Computer Science - 328*, Springer-Verlag, 1991.

[34] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modelling and Design*, Prentice-Hall Inc., 1991.

[35] B. Stroustrap, *The C++ Programming Language* (Second Edition), Addison-Wesley, 1991.

[36] H. Thimbleby, *User Interface Design*, acm press, 1990.

[37] P.T. Ward, "How to Integrate Object-Orientation with Structured Analysis and Design", *IEEE Software*, pp. 74-82, 1989.

[38] P. Wegner, "Dimensions of Object-Based Language Design", *Proceeding of OOPSLA '87*, Orland Special Issue of SIGPLAN Notices, **22**(12), pp.168-182.

[39] A. Wills, "Capsules and types in Fresco", *ECOOP 91: European Conference on Object-Oriented Programming, Lecture Notes in Computer Science - 512*, 1991, pp. 59 -76.

[40] R. Wirfs-Brock, B. Wilkerson and L. Wiener, *Design Object Oriented Software*, Prentice-Hall, 1990.

Appendix A

A.1 Syntax of Object-Oriented Design

Ood → Class_list

Class_list → Class Class_list | ε

Class → **class** ID
Inheritance
Part_of
Attributes
Operations

Attributes → **attributes** Declaration_list | ε

Inheritance → **inherits** ID Identifier_list **redefines** ID Identifier_list | ε

Part_of → **part_of** ID Identifier_list | ε

Operations → **operations** Operation_list | ε

Declarations_list → ID Identifier_list : Type; Declarations_list | ε

Identifier_list → ,ID Identifier_list | ε

Type → Map | Set | Sequence | ID

Operation_list → Ope Operation_list | ε

Ope → ID(Declarations_list) Return

Return → : Type | ϵ

Map → **map from** Type **to** Type

Set → **set of** Type

Sequence → **sequence of** Type

A.2 Tokens

1. Keywords
    - class
    - attributes
    - inherits
    - redefines
    - operations
    - map
    - set
    - sequence
    - from
    - to
    - of
    - part_of
2. Identifier

3. Punctuation

- :
- ;
- ,
- (
-)

Appendix B

Library Management System

B.1 Requirements and Assumptions

1. A library management system deals with the two major entities *users* or *borrowers* and *books*.

2. All books must have been entered in the database. A book can have multiple copies; however, every copy has a unique call number.

3. It is assumed that every book acquired by the library is never lost, meaning that it will be either in stack or loaned out.

4. All users must have been registered in order to use the facilities of the library. Every user has a unique ID number.

5. Users are classified into two categories, i.e. faculty users and student users. A faculty user can borrow a book for a maximum of 30 days, and a student can borrow a book for a maximum of 10 days.

6. A faculty borrower can borrow a maximum of 20 books, whereas the maximum limit for a student user is 10.

7. Users can reserve books. If more than one user reserves the book, the user's requests are queued. All users have equal privileges in reserving a book.

8. A book not returned on its due date is said to be 'overdue'.

9. If a user does not return a book within three days of its due date, the user loses borrowing privileges. This is done by temporarily canceling the user's registration; however, the user is entitled to hold the ID number. After paying the fine for overdue books and registration fees, the user is re-registered.

10. A user can renew a book if it is already loaned to that user and not requested by any other user.

11. Whenever a book is returned, the user who has first requested this book is informed of the return of the book and the book is placed in the stack.

B.2 VDM Specifications

B.2.1 VDM State Space

State ::

```
LIB-SYSTEM    : Library
Library       ::COLLECTION: Lib-books-set
                USERS: Borrowers-set
                REG-USERS: Borrowers-set
                LOANS: Loanmap
                RESERVED: Queue
Borrowers     = Faculty | Student
Faculty       ::IDNUMBER: IDtype
                NAME: String
                USTATUS: {"faculty"}
Student       ::IDNUMBER: IDtype
                NAME: String
                USTATUS: {"student"}
Lib-books     ::TITLE: String
                AUTHORS: String
                CALLNUMBER: CNtype
                BSTATUS: {"inshelf","loaned-out","overdue","longdue"}
IDtype        = N
```

CNtype = N

Loanmap =IDtype → Duemap

Duemap = CNtype → Borrow-date

Queue = CNtype → IDtype-list

FINES-DUE : IDtype → CNtype-set

REG-FEES-DUE : IDtype-set

The library system consists of a COLLECTION of library books and a set of USERS. The set of users also includes a subset called REG-USERS, who are entitled to use the facilities of the library. The remaining users in the set USERS have already registered with library, but their registrations are canceled because of 'longdue' on some books they borrowed. LOANS is a two-level map identifying the users who have borrowed books from the library, the books that are loaned out and their respective due dates. RESERVED is a map from books to the list of users identifying the books that are reserved and the respective users for each reserved book. A book has a title, a list of authors, a unique call number and a status. Each user belongs to either of the categories *Faculty* or *Student*. Each user has a unique ID number and a name. The map FINES-DUE identifies the set of users who have to pay fines for overdue and longdue books and the set of books for which the users own the fine. REG-FEES-DUE is a set identifying those users who have to pay the fees for re-registration.

B.2.2 Operations

1 Initialize the library system

INIT()

ext

LIB-SYSTEM: **wr** Library

FINES-DUE: **wr** IDtype → CNtype-set

REG-FEES-DUE: **wr** IDtype-set

post

(COLLECTION(lib-system)={}) ∧

(USERS(lib-system)={}) ∧

(REG-USERS(lib-system)={}) ∧

(LOANS(lib-system)= []) ∧

(RESERVED(lib-system)= []) ∧

(FINES-DUE = []) ∧

(REG-FEES-DUE = {})

2 Add a book to the library

ADD-BOOK(T: String; AU: String)

ext LIB-SYSTEM: **wr** Library

post

(∃ *cn* ∈ CNtype)

((∀ *b* ∈ COLLECTION(lib-system))

((CALLNUMBER(b) ≠ cn) ∧

(**let** new-lib-book = mk_Lib-books(t,au,cn,"inshelf") **in**

COLLECTION(lib-system)' = COLLECTION(lib-system) ∪ {new-lib-book}

tel)))

3 Add a user to the library

ADD-USER(NEWNAME: String; CODE: String)

ext LIB-SYSTEM: **wr** Library

post

(∃ *id* ∈ IDtype)

((∀ *u* ∈ USERS(lib-system))

((IDNUMBER(u)≠ id) ∧

(code = "faculty") ⊕ (code = "student") ∧

(**let** new-user = mk_Borrowers(id,newname,code) **in**

(USERS(lib-system)'=USERS(lib-system)∪{new-user}) ∧

(REG-USERS(lib-system)'=REG-USERS(lib-system)∪{new-user})

tel)))

4 Borrow a book from the library

BORROW-BOOK(ID: IDtype; CN: CNtype)

ext LIB-SYSTEM: **wr** Library

pre

(∃!u ∈ REG-USERS(lib-system))

((IDNUMBER(u) = id) ∧

(id ∈ dom LOANS (lib-system) ⇒

(USTATUS(u) = "faculty" ⇒

card dom LOANS(lib-system)(id) < 20) ∧

(USTATUS(u) = "student" ⇒

card dom LOANS(lib-system)(id) < 10) ∧

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧ (BSTATUS(b)="inshelf") ∧

(∼ (∃ u_1 ∈ USERS(lib-system))

(cn ∈ dom LOANS(lib-system)(u_1)))) ∧

(cn ∈ dom RESERVED(lib-system)

⇒ hd RESERVED(lib-system)(cn)=id))

post

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧ (BSTATUS(b)="loaned-out") ∧

(∃!u ∈ REG-USERS(lib-system))((IDNUMBER(u)=id)∧

LOANS(lib-system)(id)'=LOANS(lib-system)(id)†[cn→today] ∧

(cn ∈ dom RESERVED(lib-system) ⇒ id ∉ elems RESERVED(lib-system)(cn))

5 Return a borrowed book to the library

RETURN-BOOK(ID: IDtype; CN: CNtype)

ext LIB-SYSTEM: **wr** Library

pre

(∃!u ∈ USERS(lib-system))

((IDNUMBER(u) = id) ∧

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧

(cn ∈ dom LOANS(lib-system)(id)) ∧

(BSTATUS(b) = "loaned-out")))

post

(∃!u ∈ USERS(lib-system))

((IDNUMBER(u) = id) ∧

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧

(LOANS(lib-system)(id)' = {cn} ⩤ LOANS(lib-system)(id) ∧

(BSTATUS(b) = "inshelf")))

6 Reserve a book in the library

RESERVE-BOOK(ID: IDtype; CN: CNtype)

ext LIB-SYSTEM: **wr** Library

pre

(∃!u ∈ USERS(lib-system))

((IDNUMBER(u) = id) ∧

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧

((BSTATUS(b) = "loaned-out") ∧ (BSTATUS(b)="overdue")∨

(BSTATUS(b) = "longdue"))))

post

(∃!u ∈ REG-USERS(lib-system))

((IDNUMBER(u) = id) ∧

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧

(**let** idlist = RESERVED(lib-system)(cn) **in**

RESERVED(lib-system)'=RESERVED(lib-system)†[cn→idlist‖<id>]

tel)))

7 Renew a book borrowed from the library

RENEW-BOOK(ID: IDtype; CN: CNtype)

ext LIB-SYSTEM: **wr** Library

pre

(∃!u ∈ REG-USERS(lib-system))

((IDNUMBER(u) = id) ∧

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧ ((BSTATUS(b) = "loaned-out") ∧

(cn ∈ dom LOANS(lib-system)(id))

∧ (cn ∉ dom RESERVED(lib-system))))

post

(∃!u ∈ REG-USERS(lib-system))

((IDNUMBER(u) = id) ∧

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧

LOANS(lib-system)(id)' = LOANS(lib-system)(id)†[cn→today])))

8 Set the status of a book as "overdue"

SET-OVERDUE(CN: CNtype)

ext

LIB-SYSTEM: **wr** Library

FINES-DUE: **wr** IDtype → CNtype-set

pre

(∃!b ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧ ((BSTATUS(b) = "loaned-out") ∧

(∃!u ∈ USERS(lib-system)

((cn ∈ dom LOANS(lib-system)(IDNUMBER(u))) ∧

(let borrowdate = LOANS(lib-system)(IDNUMBER(u))(cn) **in**

(u ∈ Faculty ⇒ today = borrowdate + 31) ⊕

(u ∈ Student ⇒ today = borrowdate + 16)

tel)))

post

(∃!*b* ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧ ((BSTATUS(b) = "overdue") ∧

(∃!*u* ∈ USERS(lib-system)

((cn ∈ dom LOANS(lib-system)(IDNUMBER(u))) ∧

let cnset = fines-due(IDNUMBER(u)) **in**

fines-due(IDNUMBER(u))'= fines-due(IDNUMBER(u)) ∪ cn

tel)))

9 Set the status of a book as "longdue"

SET-LONGDUE(CN: CNtype)

ext

LIB-SYSTEM: **wr** Library

REG-FEES-DUE: **wr** IDtype-set

pre

(∃!*b* ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧ ((BSTATUS(b) = "overdue") ∧

(∃!*u* ∈ USERS(lib-system)

((cn ∈ dom LOANS(lib-system)(IDNUMBER(u))) ∧

(**let** borrowdate = LOANS(lib-system)(IDNUMBER(u))(cn) **in**

(u ∈ Faculty ⇒ today = borrowdate + 34) ⊕

(u ∈ Student ⇒ today = borrowdate + 19)

tel)))

post

(∃!*b* ∈ COLLECTION(lib-system))

((CALLNUMBER(b)=cn) ∧ ((BSTATUS(b) = "longdue") ∧

(∃!u ∈ USERS(lib-system)

((cn ∈ dom LOANS(lib-system)(IDNUMBER(u))) ∧

(REG-USERS(lib-system)' = REG-USERS(lib-system) - {u}) ∧

(reg-fees-due' = reg-fees-due ∪ {u})))

10 Pay the fine for all the books which are "overdue"

PAY-FINE(ID: IDtype)

ext

LIB-SYSTEM: **wr** Library

FINES-DUE: **wr** IDtype → CNtype-set

pre

(∃ u ∈ REG-USERS(lib-system))

((IDNUMBER(u) = id) ∧ (fines-due(id) ≠ {}))

post

(∃!u ∈ REG-USERS(lib-system))

((IDNUMBER(u) = id) ∧

(**let** cnset = fines-due(id) **in**

(LOANS(lib-system)(id)' = cnset ⩤ LOANS(lib-system)(id) ∧

(∀ cn ∈ cnset)

(∃!b ∈ COLLECTION(lib-system)

((CALLNUMBER(b)=cn) ∧

(BSTATUS(b)="overdue" ⇒ BSTATUS(b)'= "inshelf")

tel) ∧ (fines-due(id)' = {}))

11 Re-register a user after paying the fines and fees for re-registration

RE-REGISTER(ID: IDtype)

ext

LIB-SYSTEM: **wr** Library

REG-FEES-DUE: **wr** IDtype-set

FINES-DUE: **wr** IDtype → CNtype-set

pre

$(\exists! u \in \text{USERS(lib-system)})$

((IDNUMBER(u) = id) $\wedge$ (u $\in$ reg-fees-due))

post

$(\exists! u \in \text{USERS(lib-system)})$

((IDNUMBER(u) = id) $\wedge$

(REG-USERS(lib-system)'=REG-USERS(lib-system) $\cup$ {u}) $\wedge$

(reg-fees-due' = reg-fees-due - {u}) $\wedge$

(fines-due(id)' = {}) $\wedge$

(**let** cnset = dom LOANS(lib-system)(id)) **in**

(LOANS(lib-system)(id)' = cnset $\lhd$ LOANS(lib-system)(id)) $\wedge$

($\forall cn \in$ cnset)

($\exists! b \in$ COLLECTION(lib-system))

((CALLNUMBER(b) = cn) $\wedge$

((BSTATUS(b)="overdue")

$\wedge$ (BSTATUS(b)="longdue") $\Rightarrow$

((BSTATUS(b) = "inshelf")

tel))

B.3 Object-Oriented Design

Library books

class Lib_books

part_of Library

attributes

TITLE : String;

AUTHORS : String;

CALLNUMBER : CNtype;

BSTATUS : Book_status;

operations

GET_STATUS(): Book_status

NEW_BOOK() : CNtype

GET_CN(): CNtype

Users of the library

class borrowers

part_of Library

attributes

IDNUMBER : IDtype;

NAME : String;

USTATUS : String;

operations

NEW_USER(NEWNAME : String) : IDtype

GET_IDNUMBER(): IDtype

CHECK_STATUS(): B

Faculty members

class faculty

inherits borrowers **redefines** CHECK_STATUS

part_of Library

attributes

IDNUMBER : IDtype;

NAME : String;

USTATUS : String;

operations

NEW_USER(NEWNAME: String) : IDtype

GET_IDNUMBER(): IDtype

CHECK_STATUS(): B

Student

class student

inherits borrowers **redefines** CHECK_STATUS

part_of Library

attributes

IDNUMBER : IDtype;

NAME : String;

USTATUS : String;

operations

NEW_USER(NEWNAME: String) : IDtype

GET_IDNUMBER(): IDtype

CHECK_STATUS(): B

Library

class library

part_of Lib_Man_Sys

attributes

COLLECTION : set of Lib_books;

USERS, REG_USERS : set of borrowers;

LOAN : Loanmap;

RESERVED : Queue;

operations

INIT_LIBRARY()

ADD_BOOK (TIT: String; AUT : String)

ADD_USER (NEWNAME : String; CODE : String)

BORROW_BOOK(ID : IDtype; CN : CNtype)

RETURN_BOOK(ID : IDtype; CN : CNtype)

RESERVE_BOOK(ID : IDtype; CN : CNtype)

RENEW_BOOK(ID : IDtype; CN : CNtype)

SET_OVERDUE(CN : CNtype; ID : IDtype)

SET_LONGDUE(CN : CNtype; ID : IDtype)

PAY_FINE(ID : IDtype)

RE_REGISTER(ID : IDtype)

Library management system

class Lib_Man_Sys

attributes

LIB_SYSTEM : library;

FINES_DUE : map from IDtype to set of CNtype;

REG_FEES_DUE : set of IDtype;

operations

INIT_LMS()

SET_LONGDUE(CN : CNtype; ID: IDtype)

SET_OVERDUE(CN : CNtype; ID: IDtype)

PAY_FINE(ID : IDtype)

RE_REGISTER(ID : IDtype)

Books loaned to users

class Loanmap

inherits Map **redefines** Is_Domain_Member, Update, Extract_Range_Element

part_of Library

attributes

operations

INITIALIZE_LOANS()

BOOKS_BORROWED_BY_USER(ID: IDtype)

UPDATE_LOANS(ID: IDtype; CN: CNtype)

IS_BOOK_LOANED_TO_USER(ID: IDtype; CN: CNtype)

Reservation lists

class Queue

inherits Map **redefines** Is_Domain_Member, Update

part_of Library

attributes

operations

INITIALIZE_RESERVE_QUEUE()

IS_RESERVED(CN: CNtype)

FIRST_RESERVED(CN: CNtype): IDtype

ADD_TO_RESERVE(CN: CNtype; ID: IDtype)

Due dates of borrowed books

class Duemap

inherits Map **redefines** Update, Extract_Range_Element

part_of Loanmap

attributes

operations
UPDATE_BORROWED_BOOKS(ID: IDtype; CN: CNtype)
GET_BORROW_DATE(ID: IDtype; CN: CNtype)

B.4 C++ Classes

```
class Lib_books {
private:
String TITLE;
String AUTHORS;
CNtype CALLNUMBER;
Book_status BSTATUS;

public:
Book_status GET_STATUS();
CNtype NEW_BOOK();
CNtype GET_CN();
};

class borrowers {
private:
IDtype IDNUMBER;
String NAME;
String USTATUS;
```

```
public:
IDtype NEW_USER(String NEWNAME);
IDtype GET_IDNUMBER();
boolean CHECK_STATUS();
};

class faculty : public borrowers {
private:
IDtype IDNUMBER;
String NAME;
String USTATUS;

public:
IDtype NEW_USER(String NEWNAME);
IDtype GET_IDNUMBER();
boolean CHECK_STATUS();
};

class student : public borrowers {
private:
IDtype IDNUMBER;
String NAME;
String USTATUS;

public:
IDtype NEW_USER(String NEWNAME);
IDtype GET_IDNUMBER();
boolean CHECK_STATUS();
};
```

```
class library {
private:
set<Lib_books> COLLECTION;
set<borrowers> USERS, REG_USERS;
Loanmap LOAN;
Queue RESERVED;

public:
void INIT_LIBRARY();
void ADD_BOOK(String TIT, String AUT);
void ADD_USER(String NEWNAME, String CODE);
void BORROW_BOOK(IDtype ID, CNtype CN);
void RETURN_BOOK(IDtype ID, CNtype CN);
void RESERVE_BOOK(IDtype ID, CNtype CN);
void RENEW_BOOK(IDtype ID, CNtype CN);
void SET_OVERDUE(CNtype CN, IDtype ID);
void SET_LONGDUE(CNtype CN, IDtype ID);
void PAY_FINE(IDtype ID);
void RE_REGISTER(IDtype ID);
};

class Lib_Man_Sys {
private:
library LIB_SYSTEM;
map<IDtype,set<CNtype>> FINES_DUE;
set<IDtype> REG_FEES_DUE;

public:
void INIT_LMS();
void SET_LONGDUE(CNtype CN, IDtype ID);
```

```
void SET_OVERDUE(CNtype CN, IDtype ID);
void PAY_FINE(IDtype ID);
void RE_REGISTER(IDtype ID);
};
```