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

GAP --- A TOOL FOR TRANSFORMING FROM VDM SPECIFICATION INTO OBJECT-ORIENTED DESIGN

by

GAO Junming

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirement

for the Degree of Master of Computer Science at

CONCORDIA UNIVERSITY

Montréal, Quebéc, Canada

April 1992

ISBN 0-315-80937-X

TO MY WIFE AND DAUGHTERS

GAP — A Tool For Transforming From VDM Specification Into Object-Oriented Design

GAO Junming

ABSTRACT

A tool GAP, which supports direct and automatic transformation from VDM specification of software system into object-oriented design, is presented. The general transformation methodology and the implementation details are discussed. Issues of deriving compilable code from OOD are also discussed. GAP is built on a simplified VDM specification syntax based on LL(1) grammar defined in the thesis. GAP serves two purposes: first as a syntax parser and partial semantics checker to examine the VDM specification for syntactic correctness; second as a translator to transform the specification into classes and their components of an object-oriented design.

The approach endorsed in this study tries to bridge the formal techniques and object-oriented paradigm. That is to apply formal specification technique at the first phase of software development, and then to use object-oriented paradigm to implement the software.

The main outcome of GAP is composed of three parts:

The first part is the class information. It includes designated classes, inheritance relations, client-server relations, member variables and their types, and suggested member functions and parameters for the classes.

The second part explains how to implement VDM operations within the scope of classes.

The last part brings out the possible relationships among the member functions and between member functions and specification functions.

A medium sized sample system is given in Appendices. That shows GAP is both effective and promising for more complex situations.

To further this study, a supporting tool with an interactive interface and a knowledge-based code generator is proposed and briefly described.

Acknowledgements

I am grateful to my supervising professor V.S. Alagar. From him I have learned most of the fundamental concepts and principles of Software Engineering, especially Formal Methods and Specifications.

I am grateful to my wife Zhang Huiling and daughters Fay and Linda. It would have been impossible for me to complete the research without their continuing encouragement and help.

TABLE OF CONTENTS

1

Introduction

This thesis discusses the transformation method proposed in [ALP91] with a view towards adapting it for implementation in C++. The transformation as implemented in the thesis accepts VDM specifications within VDM-SL and generates classes and their structural relationships in C++. Moreover, the relationship between C++ classes and the VDM specifications are established. This is particularly useful for an implementer in ensuring that no junk classes are produced. Thus, the thesis while addressing general issues in the transformation builds an actual model of the translator - showing that the transformation process, the critical link in a formal software development process, is both feasible and practical.

The layout of the thesis is as follows. The introductory section 1 discusses the motivation and the background of the study. Section 2 gives a brief outline of the VDM specification language and introduces a simplified VDM specification language and its extended Backus-Naur form. Section 3 discusses an object oriented paradigm. Only issues relating to the transformation are emphasized, and there is a brief comparison between an object-oriented approach and a traditional function oriented approach. Section 4 is the central piece of the thesis, which provides a detailed discussion of the issues in the transformation process from a VDM specification into object-oriented design. Section 5 illustrates the model translator GAP, which also serves as a VDM specification syntax parser and semantics checker. A

comparison with other systems is presented too. And finally section 6 concludes the study and points out further development directions.

1.1 Formal Methods and VDM

From the early 80's, two trends have been haunting the world of software engineering. One is formal method and another is object-oriented paradigm. Formal method in developing large software systems, pioneered by Vienna Development Method (VDM)[BJo78, BJo82, Jon86, Jon90], is gaining solid ground among more and more people in both the academic and industrial community. Formal method is a term which is used to cover both the use of mathematical notation in the functional specifications of systems and the use of justifications which relate designs to their specifications. Aiming at controlling the ever-growing enormous complexity in the development of large software systems, formal methods provide precise notations for capturing functional specification decisions by their abstract characterizations of the requirements. VDM specification language is used for this purpose. VDM was developed in an industrial environment but has evoked considerable academic research. It offers both specification notations and proof obligations which enable a designer to establish the correctness of design steps. It is a development method in the sense that it provides unambiguous notation and a framework for recording and justifying specifications and design steps.

Since its first appearance, VDM has undergone many innovations. New directions have been pursued. One of the developments is tools for the automation of formal software development process. During its initial stages of development, VDM specification language is mainly a handy tool to avoid "using informal English mixed with technical jargon"[BJo78] to specify the architecture (or models) of software. So, in the formative studies of the high level language development, no interpreter or compiler was intended for the language. However, recently several attempts have been made to build automatic development tools to help write correct specifications or implement software specified by VDM.

To name a few, the MULE system is an example of an environment giving support in the syntactic generation of formal objects such as specifications, and the IPSE 2.5 system is an attempt to produce an industrial scale system to support the use of formal methods over the whole life cycle of a software development[Jon87]. One of its main result is the production of **Mural** tool which is an interactive mathematical reasoning environment designed to assist the kind of theorem proving tasks that arise when following a formal methods approach to software engineering[BRi91, JJL91]. A related project is **SpecBox**[MFr91, FMB89], which is an interactive, window-based support tool for formal specification with four components: a syntax checker, a semantic analyser, a LaTeXdocument generator and a **mural** interface. We have more to say about Mural in Section 5.

The first compiler for VDM specification language was reported in [Haß87]. It served two purposes: first, it may be employed for rapid prototyping; second, it is a step towards a compiler-compiler system that automatically transforms VDM compiler specification into programs.

The RAISE software development method[Pre87, NHW88, Geo91], associated with its specification language and the supporting tools, enables the stepwise development of both sequential and concurrent software from abstract specification through design to implementation. The tool set of RAISE system includes a trial version of the basic syntax directed editor, database facilities, and the proof tools.

It should be pointed out that there is a common feature that prevents the VDM specification language compilers from practical application. These compilers require the specification be written in a constructive manner. Such operation specifications shall indicate ways on how they are computed. To be obligated to do so will greatly weaken the declarative description power of the VDM specification.

A very interesting project, called the VDM domain compiler, started in 1988, is still under development[SHó90, SHó91]. It adopts an approach somewhat similar to that of ours. However it is based on purely functional programming techniques — the state model is absent. More discussions are given in Section 5.

During the later stages of development, it was realized that lack of a standard syntax and semantics for VDM hindered the development of tools and general acceptance of VDM as an industrial software development method. Great effort has been directed to standardize the VDM specification language[Sen87, And88, BSI89, BSI91].

Besides Vienna development method, there are a dozen other specification languages and development methods that emerged following the VDM. We are not going to list them and only mention LARCH which is of interest to this study here. LARCH[GHo80, GHo82, GHo91, GHM90] is a two-tiered specification language, which separates the specification of underlying abstractions from the specification of state transformations. The first tier is the *Larch Shared Language* (LSL) and is used to specify underlying mathematical abstractions while the second tier is a *Larch Interface Language* and is used to specify state transformations. Larch interface languages are programming language specific and LSL is programming language independent. Besides the Larch shared language and interface languages, a software development methodology and several development tools are under development to support the systematic application of Larch[Win83, WZa91]. Some comparison of Larch with our approach will be presented later.

More supporting tools and new methodologies are reported in the fourth international symposium of VDM Europe[PTo91].

1.2 Object-Oriented Paradigm

The emergence of object-oriented paradigm for designing and implementing software systems[Boo86] is a relatively new development in software engineering. Object-oriented design leads to software architectures based on the objects every system or subsystem manipulates. A more precise definition is that object-oriented design is the construction of software systems as structured collections of abstract data type implementations. Its major goals are to improve programmer productivity by increasing software extensibility and reusability and to control the complexity

and cost of software maintenance. When object-oriented paradigm is used, the design phase of software development is linked more closely to the implementation phase. It is conjectured that object-oriented paradigm is the solution to the problems suffered in traditional function-oriented paradigms.

However, the design process for the object-oriented paradigm remains ad hoc. Several efforts have been made to improve the situation. B. Alabiso[Ala88] illustrated the transformation of data flow analysis model into object-oriented design. P.T. Ward[War89] described how to integrate object-orientation with structured analysis and design. K. Beck and H. Cunningham[BCu89] suggested the use of index cards to record initial class designs. R. Wirfs-Brock, B. Wilkerson and L. Wiener[WWW90] presented a detailed example of object-oriented design using a responsibility-driven approach, in which the index card is used. L.L. Constantine[Con90] discussed an approach to mix conventional function-oriented design with object-oriented design concepts in a hybrid design strategy. J. Rumbaugh, M. Blaha and W. Premerlani, in their book[RBP91], covered the entire development life cycle — analysis, design, and implementation, using a graphical notation and methodology developed by the authors.

Another direction of research is towards a formal systematic development of object-oriented design. Unlike function-oriented paradigm, where formal method has been widely accepted and applied, object-oriented approach is still ad-hoc; some formal approaches that are under development are reported in [AGo91, Bre91, Dah90].

1.3 Overview of This Study

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. All the methods and supporting tools for VDM are function-oriented. This study tries to bridge the two approaches to enjoy the advantages of both of them. Our goal is to build a detailed object-oriented design from VDM specifications. Thus the modelling methods built for function-oriented

approach, which are widely accepted and applied, can be used for object-oriented design. The result is a supporting tool that supports automatic transformation. To be specific, our tasks are:

- to identify object-class model from the specification,
- to implement the VDM variable types in the framework of OOD,
- to implement the built-in operators associated with the VDM variable types,
- to find the inheritance relationship between the classes,
- to identify operations acting on the classes,
- to determine client and server relationship among the system,
- to create functions that accomplish the operations specified by VDM specification,
- to build supporting tools to automatically implement the transformation.

We have selected C++ as our target object-oriented programming language. This study is the first comprehensive account of the practicality of the suggested method in [ALP91]. There are also some major improvements over this method. That we have made:

1 Implementation of the VDM built-in variable types and built-in operators in C++,

2 Identifying inheritance relationship among C++ classes,

3 Creating member functions for classes,

4 Determining client-server relationship between classes,

5 Treating the state invariant in a way consistent with reusability, and

6 Presenting the idea that when comparison is made to find the possible relationship between two predicates, the variable types shall be used to replace the variables themselves.

To derive an automatic transformation, we are discussing a simplified version of VDM specification language and its formal syntax.

GAP system is built to achieve the automatic transformation. It serves two purposes:

1 as a VDM specification syntax parser and partial semantics checker; and

2 as a transformer that carries out the actual transformation into object-oriented design.

The main output of GAP has three parts. The first part shows the class information. For each class created by the translator, the following information will be included:

- class name;
- inheritance from base classes;
- member variables and their types; and
- suggested member functions and their parameters;
- server classes and the associated messages.

The second part of the output deals with the operations specified in VDM operation specifications. The result for each operation displays:

- operation name;
- class name to which the operation goes;
- what and where are the functions which compose the operation (what client-server pairs are needed to perform the operation).

The last part reveals the possible relationships between the suggested functions. Each section in this part contains the following items:

- function name;
- client-server pairs; and
- function specification.

The other results of GAP are a listing file, which lists source input and any syntax error and semantics error ever detected, and a symbol table file, which shows the symbol table built for the source specification.

To further our study, a new supporting tool is also discussed. The tool reifies and implements each member function for classes, generates software source code ready for compilation. GAP associated with this tool will then provide an interactive and integrated software development environment from VDM specification into final programming language dependent implementation. The two basic components of the tool are: an interactive interface which enables a developer to grasp all information available to implement the member functions and a knowledge-based code generator which produces compilation ready code and records the classes and functions for later reuse.

2

Vienna Development Method – A Brief Outline

Vienna Development Method (VDM) is a model based formal software development method, which was developed at the IBM Vienna Research Laboratories during the 1970s. VDM was based on the approach to programming language theory known as denotational semantics[BJo78]. It was subsequently developed into a general-purpose software development method and applied to a wide variety of applications.

The objective of VDM is to control the complexity of any complex system by the construction of a formal definition of the required functions. This specification is a reference point for subsequent development processes. The feed back from each previous step provides a justification of the correctness.

The common carrier of these justifications and proofs is the specification language based on **Meta-IV**.

2.1 VDM Specification Language

Internally, the specification language of VDM is called **Meta-IV**. We are not going to distinguish the term Meta-IV and VDM hereafter.

At the beginning, VDM specification language was mainly a handy tool to avoid "using informal English mixed with technical jargon"[BJo78] to specify the architecture (or models) of software. Being a high level specification language, no interpreter or compiler was intended for the language. In fact, as one author mentioned: "We wish, as we have done in the past, and as we intend to continue doing in the future, to further develop the notation and to express notions in ways for which no mechanical interpreter system can ever be provided"[BJo78].

In subsequent years, it was realized that lack of a standard syntax and semantics hindered the development of tools and general acceptance of VDM as an industrial software development method. Great effort was invested to standardize the VDM specification language[Sen87, And88]. Two versions of the VDM specification language have been made public by British Standards Institute[BSI89, BSI91].

To simplify our work in this study, we follow the notation in [CHJ86]. We have found this notation sufficient to meet most specification requirements of the systems in which we are interested. What is absent in this subset from the BSI proto-standard is the notion of **Modularization**, which is a supplement to the VDM specification language by BSI. We intend to make the problem simple enough at this stage of study.

A VDM specification is composed of three parts. The first part is the definition of a number of abstract variable types. The variables of these type are used to represent the internal state of the system modelled. The second part is the invariant definitions. These are predicates which define additional constraints on the values that variables may assume. These additional constrains are usually not specified in the type definition part. The last part is the definitions of operations and functions that act on the variables. Each part of the VDM specification will be discussed in more detail in the following paragraphs.

2.1.1 Variables and Type Definitions

The Basic notation in VDM specification language is the concept of **set**. A set is a collection of similar items, like values and names. Each item is called an **element**.

A **variable** is part of the internal state of the system being modelled. Each variable has a **type**, which denotes a set of possible values that the variable can take. At any time, the variable must assume one of the values in the set.

Comparing with some high level programming languages, VDM provides more primitive and few variable types. It is based on the use of mathematical abstractions such as sets and finite mappings. So it more flexibe and easier for developer to grasp the abstract aspect of a system model.

The basic types in VDM specification are:

Boolean	the set of boolean values
Int	the set of integers
Nat0	the set of non-negative integers
Nat	the set of positive integers
String	the set of any composition of letters and digits.

▷Example 2.1◁

Age : *Nat0* defines a variable *Age* of type Nat0; that is to say *Age* can only assume a non-negative value at any time.

□

One way to create new variable types is to define the set of all possible values the variable can assume. For example, the built-in type **Int** is the infinite set of integers. It is possible for the user to create a new type by listing all possible values, if it is a finite set. One of the best examples of this kind is the built-in *Boolean* type, which is defined as

$$Boolean = \{true, false\}$$

Note the syntax difference of defining a variable and creating a new type in above declarations.

In the case of infinite set, or in the case that all possible values are not known yet, or the set is defined by some predicates, VDM allows the user to leave the definition open to later determination. Informal specification in the form of a comment can be a definition of the new variable type.

From an existing type, user can define a **subtype**, which can take a subset of a set as the possible values of a variable.

It is also allowed to rename a type to have a **renamed type**, whose name may be more meaningful to the user.

In set theory, some **operations** can act on one or more sets to produce new sets. These operations are **union, difference** and **intersection.** They can be used to yield new types in VDM. There are also boolean operations defined on a set. **Membership** of an element in a set or **relationship** between two sets are defined in VDM too, according to set theory. The **cardinality** of a set is defined as the number of elements of the set.

The above types can be considered as primitive types. VDM also provides constructs to create composite types to specify more complex models. In a composite type, a variable can assume a sequence of values of another type(or types) as its value.

It is possible that even a subset of a set can be the element of another set. This is the **powerset** type that is the set of all subset of the base set (type). In VDM, it is denoted by a suffix *'-set'* appended to a type name to create the powerset of that type. A variable of a powerset type will take a subset of the base set (base type) as its value.

Similarly, a **list** type is introduced to specify a variable which takes a sequence of values from the base type as its value. The suffix *'-list'* attached to a type name creates the set of finite lists which can be made from the base set (base type). A list of values is enclosed by the left angle '⟨' and the right angle '⟩'.

The operators provided for operating on lists are: **hd**, **tl**, **len**, and **elems**. They are unary operators which yield respectively the first element of the list, the

rest of list after removing the first element, the number of elements of the list, and the element set of the list respectively. Two lists can also be concatenated by a '⌊' operator.

The **record** type, like in some programming languages, allows a variable to assume a fixed length sequence of values drawn from possible different types as its value. The component parts are called **fields** of the record.

Two operations are defined for a record type. The construction of an instance of a record type is done by a **make** function, which coerces discrete values into a record structure in a special notation. The field **selector** is used to access individual fields of records.

▷Example 2.2◁

A *Pixel* record type is defined for CGA monitor in color graphic mode (320×200 resolution) as:

Pixel :: X : *Width*
Y : *Hight*
COLOR : *Colorset*
Width={0..319}
Hight={0..199}
Colorset={blue, green, red, brown}.

One instance of *Pixel* is created and assigned to the variable *onepixel*:

$$onepixel = \textbf{mk-}Pixel(100, 80, red)$$

The field selector is used as:

X(onepixel)=100
Y(onepixel)=80

COLOR(onepixel)=red

□

The **mapping** type has a finite domain and provides a flexible alternative way to define functions. Mappings are special kind of functions which map elements of the domain set to elements of the range set.

▷Example 2.3◁

One of the explicitly defined mappings is

$m = [red \rightarrow 1, blue \rightarrow 2, green \rightarrow 3, white \rightarrow 4]$

which is a mapping from $Colorset$ to $S = \{1..4\}$.

□

To define a mapping type, of which a variable assumes a value of a map, user shall indicates the domain set and the range set by an arrow between them.

▷Example 2.4◁

$M = Colorset \rightarrow S$

defines a mapping type. A variable m of type M can take the value as previously assigned in [Example 2.3].

□

There are several operators defined to operate on mapping or mappings. The unary operators **dom** and **rng** yield the domain set and the range set of a mapping. The binary operator **overwrite** denoted as '†' combines two mappings and yields a combined mapping, in which the range element in the second operand takes priority. The **restrict by** '\' and the **restrict to** '/' operators take a mapping and a set as

their operants, and create a new mapping as result. A mapping may be applied to an element of its domain to obtain the corresponding element from the range.

▷Example 2.5◁

$m(red) = 1$ (mapping applied to element).

Suppose:

$n = [red \rightarrow 1, blue \rightarrow 2, white \rightarrow 3, yellow \rightarrow 5]$.

Then we have

$m \dagger n = [red \rightarrow 1, blue \rightarrow 2, white \rightarrow 3, green \rightarrow 4, yellow \rightarrow 5]$,

$n \backslash \{red, blue\} = [red \rightarrow 1, blue \rightarrow 2]$, and

$n / \{red, blue\} = [white \rightarrow 3, yellow \rightarrow 5]$.

[ ]

Equality operator in VDM can be applied to any two variables of the same type.

2.1.2 States, Invariants and Operations

As described in the previous sections that a variable type is an abstraction of a physical object, a variable of that type is used to represent the internal state of the object. The first part of the specification is indeed an abstract model of the system to be specified. In fact, each variable type is part of the system model.

The heart of this consideration is the concept of the system state. The objective is to make it easier to specify operations of the system model. The state of the system is a special record which consists of a set of variables that are the key description of the system model. The key description here means the highest level abstraction of the system model. As a matter of fact, state definition is not

mandatory for a specification, it can be defined when a user feels it will help to enhance the system modelling.

Another concept associated with variable types is **invariant condition**. Invariants are a set of predicates which define additional constraints on the values that variables may assume. These constraints are not or can not be fully expressed in the type definition session and are preserved in the life time of a variable of that type. Invariant condition can be defined for the system state, as well as for other variable types.

Operations are those which specifies changes to the values of some global variables (introduced in the state definition). There are two ways to specify an operation. One is an implicit operation specification, and another is an explicit operation specification. To illustrate the explicit and implicit specifications, we take the example of a marriage bureau database[CHJ86].

▷Example 2.6◁

State ::	UNMARRIED : *Person-set*
	MARRIED : *Person-set*
Person =	/* to be defined later */
inv-State:=	$unmarried \cap married = \{\}$
REGISTER	(P:*Person*)
ext	UNMARRIED :**wr** *Person-set*
	MARRIED :**rd** *Person-set*
pre	$p \notin unmarried \wedge p \notin married$
post	$unmarried' = unmarried \cup \{p\}$

□

The implicit specification of an operation consists of four parts:

1 the name part, which contains the name of the operation, the input parameters, and the possible output parameters.

2 the **ext**ernal part, which indicates which part of the state the operation will need access. The key word **ext** leads the external part and the keywords **wr, rd** are used to specify the access status. **wr** means write and read access(write-access) and **rd** means read only access(read-access).

3 the **pre**-condition part, which is a predicate over the values of the input parameters and the initial state. It indicates the conditions for which the operation is defined to have an effect. If the precondition fails, it means error and the operation is not defined.

4 the **post**-condition part, which indicates how the values of the variables are affected by the operation or how the output parameter is generated. In the post condition, it is necessary to refer to both values of a variable before the operation and after the operation. Conventionally, the variables belonging to the post-state are sufficed with a prime.

As we see, an implicit specification specifies the name, the input and out parameters, the external part of the state that the operation will access, and the logical conditions before the operation and after the operation, as well as what and how it is affected by the operation. What an implicit specification does not specify is how to carry out the operation.

An explicit specification does give a method to compute the result. That is, an explicit specification requires that a rule be given to compute the result from the input parameters. An explicit specification contains two parts.

1 the signature, which indicates the name of the operation, the input parameter list, and the possible output parameter, and

2 the body, which gives an operational procedure to carry out the operation.

⊳Example 2.7⊲

The specification of an operation for computing the distance of two points in space would be:

```
distance : Point × Point → Nat0
post-distance (p, q, d)  ≜
```

$$d' = \sqrt{(X(p) - X(q))^2 + (Y(p) - Y(q))^2 + (Z(p) - Z(q))^2}$$

□

2.2 Syntax of VDM Specification Language

The VDM specification language can be strictly defined by a context-free grammar. It is convenient to use extended Backus Naur form to define a grammar. A complete syntax of the VDM specification language in extended Backus Naur form can be found in Appendix A. Compared with the suggested standard VDM specification language[Jon90], the syntax defined in Appendix A is a simplified version.

Since VDM specification language is designed for software designer to specify the system design, it is not a programming language. As mentioned before, it was intentionally designed to prevent creating any interpreter. Therefore the syntax is not very suitable for direct processing by computers.

In order to overcome this difficulty, we have made several modifications to the base language.

1 The language will not distinguish upper cases and lower cases, although it is conventional to differ the usage of upper cases and lower cases for more readable specifications.

2 It is supposed that every type used in the specification must be either a primitive type, or an explicitly defined type. That is to say, an implicit set definition cannot be used to define a variable or other types.

▷Example 2.8◁

$x : \{male, female\}$ is not allowed. However,

$Sex = \{male, female\}$
$x : Sex$

is a legal definition.

□

3 For very practical reasons, only ASCII symbols are allowed. The rich set of VDM operators are re-defined by ASCII symbols. The details are also listed in Appendix D: Keywords and Special Symbols.

4 There is no particular difference between an operation and a function. Conventionally, an operation and a function differ from their parameters and the data they have access to. An operation has access to global variables that are part of the state, while a function only takes local variables as its parameters.

5 The unspecified built-in functions can be used throughout the specification. Function name such as **abs, sqrt, sin, cos, etc,** can be used without specification as far as the syntax of the function call being observed.

6 To make the language unambiguous, an **end** sign is put at the end of the if statement. Otherwise, to understand the language correctly, some kind of convention shall be followed to avoid the ambiguity. We have adapted a simple approach to solve this problem.

7 To follow the convention developed in the research environment, in which this study is carried on, a **tel** sign is put at the end of the let statement.

8 To simplify the handling of mapping type, the domain type is excluded by a power-set type, or a list type, or a mapping type.

9 To reduce the total number of operators, several operators have more than one semantic meanings in case that they will not lead to ambiguity. These operators are: '+' for plus and union, '−' for minus, negative and difference, '∗' for times

and intersection, '/' for divide and restrict to, '|' for or and concatenate. Besides the meaning of the operators it is also important to ensure that the precedence of the operators shall not conflict. It is fortunate that the precedence of the operators do not violate the original precedence.

10 To allow easy specification, variable name overlapping is allowed. That is some variable names can be used as components of different type definitions, or as local variables in operations. However, each variable type shall have a distinct name.

11 To avoid ambiguity in our syntax, operator $\in$ is translated to different symbols in different situations. In quantified expressions, it is translated as *in*; and in membership relation, as *isam*(*is a member*), although they are basically of the same meaning.

The top-down nature of the VDM specification syntax allows a top-down parser to be developed to parse the specification. The well developed and relatively simple recursive descent parsing technique is used to parse the grammar.

The problem with recursive descent parsing is that is it not a deterministic method. When there is more than one production for one nonterminal, especially, when there is a common prefix among these productions, the parser can not decide which production to derive. The fact that backtracking becomes necessary in such a situation will greatly reduce the performance of the technique.

▷Example 2.9◁

Let us look at the productions that generate type definitions in VDM (Number before the rule indicates the index of the rule in Appendix A):

2 <state_definition> ⇒ <type_defn> { <type_definition> }
3 <type_definition> ⇒ <record_type_def> | <set_type_def> |
<map_type_def> | <rename_type_def> |
<union_type_def> | <undefined_type_def>

4 <record_type_def> ⇒ <type_name> ' ::' <rec_field_list>
5 <rec_field_list> ⇒ <field_name> ' : ' <type_name_plus> { <rec_field_list> }
6 <field_name> ⇒ <var_name>
7 <type_name_plus> ⇒ <type_name> [<type_suffix>]
8 <type_name> ⇒ <Basic_type> | <var_name>
9 <type_suffix> ⇒ -set | -list
10 <Basic_type> ⇒ boolean | int | nat0 | nat | string

It is obvious that this grammar is not LL(1). In production set 3, more than one rule begins with a type name. There is no way to determine which one to follow by just one look-ahead token.

Let us focus on one specific rule. Even after <record_type_def> is selected, it will not be possible to judge which rule to follow after one field of the record type is defined. There are several possibilities to pursue.

□

According to the syntax, we can have three cases after any field of a record is defined:

1 There is more field to be defined:

▷Example 2.10◁

State :: *FirstField* : *FieldType*
MoreField : *MoreType*

□

2 There is no more field. The next statement is a new type definition:

▷Example 2.11◁

State :: FirstField : FieldType
NewType = ···

or

State :: FirstField : FieldType
NewType :: ···

□

When the parser comes to the first token of the second line, it is an identifier. The parser can not decide which case it is without one more look-ahead token.

3 There is no more type definition and the next statement is an invariant definition statement. This case does not cause any trouble, since the terminal **inv** serves as the predict symbol.

Fortunately, in special cases, the recursive decent parser is very effective. LL(1) grammar is one of the solutions: the first L in LL(1) means reading the context to be parsed from left to right; and the second L says deriving from the left-most and the number 1 in the parentheses indicates that only one lookahead symbol is needed beyond the current symbol.

An LL(1) class grammar is defined to have disjoint predict sets for productions that share a common left-hand side. It is possible to rewrite the productions using equivalent transformations for some grammars and to make them in LL(1) form. The SVDM syntax can be rewritten into LL(1). We have examined and rewritten each rule in Appendix A to eliminate common prefixes and left recursion and to preserve the uniqueness of the predict sets.

The two most important re-write rules that we used to transform the grammar to LL(1) are the following:

Rule 1. *There are two productions or more with the same left-hand side and a common prefix in the right-hand side. Let*

$$S = \{A \rightarrow \alpha\beta, \cdots, A \rightarrow \alpha\zeta\}$$

Create a new nonterminal N; Replace S with the new production set

$$S' = \{A \to \alpha\ N,\ \ , N \to \beta, \cdots, N \to \zeta\}$$

Rule 2. *Suppose that the following production set occurs in the grammar:*

$$S = \{A \to \cdots B \cdots,\ \ B \to C,\ \ \cdots\}.$$

S can be replaced by the new production set

$$S' = \{A \to \cdots C \cdots,\ \ \cdots\}$$

▷Example 2.12◁

The rewritten production set for the productions in Example 2.9 are given below (only necessary productions are listed):

2 <state_definition> ⇒ <type_defn> { <state_definition> }
3 <state_definition> ⇒ nil
4 <type_definition> ⇒ <type_name> <type_def_body>
5 <type_def_body> ⇒ ' ::' <var_name> ' : ' <type_name_plus> <more_rec_field>
6 <type_def_body> ⇒ ' =' <other_type_body>
16 <more_rec_field> ⇒ <var_name> <rec_field_more>
17 <more_rec_field> ⇒ nil
18 <rec_field_more> ⇒ ' : ' <type_name_plus> <more_rec_field>
19 <rec_field_more> ⇒ <type_def_body>
20 <type_name_plus> ⇒ <type_name> <type_suffix>
23 <type_suffix> ⇒ -set
24 <type_suffix> ⇒ -list
25 <type_suffix> ⇒ nil

Now the predict sets of different production rules for the same non-terminal are dis-joint; that is to say, the intersection of these predict sets is empty.

□

The transformation to LL(1) causes one problem in understanding each production rule. The semantics is slightly different from the original one, since the terminals and non-terminals are reorganized. New non-terminals may be created, and old ones may vanish. Therefore the original boundary of different entities may disappear. This will result in different meanings among new production rules and new production rules.

▷Example 2.13◁

We have <record_type_def> for generating record type and have one production set for generating each variable type in the simplified VDM. In the LL(1) syntax, there is no such a clear distinct production set for each type definition. The production rules to generate these variable types are mixed. It is difficult to match them with the original ones.

□

The gained advantage, however, is that the grammar can be easily understood by a recursive descent parser. The complete rewritten LL(1) grammar is listed in Appendix B.

There is a modification over the grammar in Appendix A besides equivalent production rewriting. All productions that create valid variable names, constant strings, and constant integers are ignored, since it is supposed that the scanner can correctly pick up these tokens. The new tokens *Ident*, *Constring*, and *ConstInt* as well as < *comment* > are treated as terminals.

3

Object-Oriented Design and C++

The term **object-oriented design** (OOD) is widely used, but it seems that experts cannot agree on a common definition. However most experts agree that object-oriented approach involves: 1) defining **abstract data types** representing complex real world or abstract objects and 2) organizing software around the collection of abstract data types with a view toward exploiting their common features. Its underlying concepts are **data abstraction, inheritance**, and **polymorphism**. The term **data abstraction** refers to the process of defining abstract data types; **inheritance** and **polymorphism** refer to the mechanisms that enable us to take advantage of the common characteristics of the abstract data types the **objects** in OOD.

OOD enables us to remain close to the conceptual, high level model of the real world problem we are trying to solve. The modularity of objects and the ability to implement program in relatively independent units that are easy to maintain and extend are advantages of object-oriented design.

3.1 Data Abstraction

Data abstraction is the process of defining a data type (abstract data type). An abstract data type is described by its external view: avaiable services and properties

of these service, without indicating concrete implementation representations.

An object can be created from an abstract data type. We can think of the abstract data type as a template from which specific instances of objects can be created as needed. The variables represent the information contained in the object, whereas the functions define the operations that can be performed on that object.

In C++, class is an implementation of an abstract data type. The interface part of a class describes the services provided and the parameters they need. The private part is a set of variables used to represent the internal state of the object. The functions (services) can be implemented inside or outside the class definition.

The variables are called member variables in C++. The functions are called member functions and are known as methods in some OOD literatures.

In C++, one can perform an operation by invoking one of the member functions (methods). In Smalltalk-based OOD literatures, this mechanism is described as sending a message to an object and causing it to execute the specified method.

A class serves two roles, as a lexical scope and as a type. As a lexical scope, a class defines a set of immutable bindings between names and certain kinds of entities, which include data declarations and various kinds of literals (types, enumerations, and functions). The members of a class have distinct names, except for member functions, which must be distinguishable only by their names and their declared argument types (functions distinguished only by their declared argument types are called overloaded functions). A class provides lexical context for the definitions(bodies) of its member functions and for nested class and function definitions; the enclosed definitions can be of several varieties: ordinary, static, virtual, and pure virtual. Data declarations can be of two varieties: ordinary and static.

A class also defines a type, which is a pattern for instantiating objects, called class instances. A class instance is a compound object: it consists of multiple subobjects, called components. In C++ term, the subobjects are member variables. The member variables of a class instance are determined by the class. For a simple class (not defined by derivation), there is one instance member variable for each declaration, each ordinary member function, and each virtual member function.

Member functions can be viewed as closures unique to the instance: they have direct access to the member variables of that instance, and can refer to the instance itself using the variable *this*.

Class members can be named directly (in their scope) or using the notation *classname* :: *membername*. Members can be accessed using the notation *instance.member*, where *instance* is an expression denoting a class instance and *member* names a class member.

▷Example 3.1◁

A complex example is $a.A :: B :: x$, where x is a member of the class that is the B member of class A, and a is an instance of a class containing a corresponding x member.

□

Within a member function, member variables of *this* are accessed using the name alone.

3.2 Inheritance

Data abstraction does not cover an important characteristic of objects. Real-world objects do not exist in isolation. Each object is related to one or more objects in their world of existence. We can describe a new kind of object by pointing out how the new object's characteristics and behavior differ from that of a class of objects that already exists. This notion of defining a new object(class) in terms of an old one is an integral part of object-oriented design. Inheritance is a mechanism to achieve this goal.

Inheritance imposes a hierarchical relationship among classes in which a child class inherits data and behavior from its parent. In C++ terminology, the parent class is known as the base class, the child is known as the derived class. The

relationship between the parent class and the child class is often called *is-a* relation. An object of the derived class *is-an* object of the base class too.

The effect of derivation on the derived class lexical scope is similar to nested scopes: the lexical scope of the derived class includes not only the member variables it defined explicitly, but also any member of the base class that is neither redefined in the derived class nor ambiguous in multiple base classes. The effect of derivation on instances of a derived class is composition: an instance of a derived class contains not only the members corresponding to the members defined directly in the derived class, but also one unnamed instance member of each base class. These unnamed members are called *base members*.

A member function can access only the member variables of the instance of its defining class. Although redefined base class members are not part of the lexical scope of the derived class, they can be accessed from the derived class scope using explicit qualification.

An implicit type conversion is defined from type 'pointer to derived class' to type 'pointer to base class', for each base class. Its effect is to convert a pointer to the derived class instance to a pointer to the corresponding base member. This conversion achieves the effect of inclusion polymorphism: a pointer to a derived class instance can be passed as an argument to a function expecting a pointer to a base class instance.

3.3 Polymorphism

In a literal sense, *polymorphism* means the feature of having more than one form. In the context of OOD, polymorphism refers to the fact that a single operation can have different behavior in different objects. In other words, different objects react differently to the same message.

Polymorphism helps us to simplify the syntax of performing the same operation on a collection of objects. In C++, polymorphism is supported in two ways. First,

overloading enables polymorphic functions (operators) in the same class working with many different argument types.

Second, *late binding* or *dynamic binding* simplifies the syntax of performing the same operation with a hierarchy of classes. The interface to the classes can be kept clean, because it does not have to be unique names for similar operations on each derived class.

3.4 Responsibility-Driven Design

The discussions so far did not touch the heart of the object-oriented paradigm. The most important aspect of object-oriented paradigm is that it is to be considered a design technique driven by delegation of responsibilities. This technique is called *responsibility separation.*

In the traditional process-state model which describes the behavior of a computer executing a program, the computer is a data manager, following some pattern of instructions, wandering through memory, pulling values out of various slots, transforming them in some manner, and pushing the results back into other slots. By examining the values in the slots, we can determine the state of the machine or the results produced by a computation.

In contrast, in the object-oriented framework, memory addresses or variables or assignments or any of the conventional programming terms are never mentioned. Instead they are objects, messages, and responsibility for some action. In D. Ingalls' memorable phrase, "Instead of a bit-grinding processor...plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires"[Ing81].

The basic philosophy in object-oriented design is the delegation of responsibility for activity. Every object is responsible for its own internal state, and makes changes to that state according to a few fixed rules of behavior. Alternatively, every activity must be the responsibility of some object, or it will not be performed.

In a recent book, Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener [WWW91] present what they call a responsibility-driven design method for object-oriented software. The authors model the software as a collection of collaborating objects, each with specific responsibilities. The objects are modeled by a client-server relationship, in which a client class makes a request to have a specific task performed by the server.

The responsibility-driven design is broken down into two phases:

1 *Exploratory phase.* Here developer discovers the classes required to model the application, divides up the system's total responsibility, and delegates the appropriate responsibilities to the classes. Developer also needs to identify the collaboration among the classes.

2 *Analysis phase.* The purpose of this phase is to refine the design created during the exploratory phase. Developer moves the common responsibilities to the base classes so that code is reused as much as possible. Here developer groups classes that work closely together. In the analysis phase, developer also derives complete specifications of the classes and determines the message protocols to be used by client-server pairs.

3.5 Comparison with Function-Oriented Paradigm

The traditional function-oriented design paradigm is based on the top-down strategy. There are several problems with the top-down approach so prevalent in structured techniques. As pointed out by B. Meyer[Mey88], top-down design

- does not allow for evolutionary changes in software.
- characterizes the system as having a single top-level function, which is not always true (in Meyer's words: "Real systems have no top").
- gives functions more importance than data, thus ignoring important characteristics of the data.

- hampers reusability, since submodules are usually written to satisfy the specific needs of a higher-level module.

Due to these weaknesses, design flaws arise in the functional-oriented models. Below we discuss them in some aspects.

Application Dependency

The functional-oriented method will lead to heavy application domain dependent software. Since it assumes that functions are on the top of concern, the system is characterised by one top function. The data structure aspect is neglected. The data structure is scattered among the functions that use it. Any modification in function specification usually requires changes both in function construction and data structure. Therefore enhancement and maintenance become very expensive.

On the contrary, in object-oriented method, the conceptual entities in the problem domain are potential objects in the software. It is easy and natural to identify physical entities and conceptual entities in the problem domain in analysis. The functions and operations are seen as natural associations to the objects and thus more close to the object than that of functional approach. Software produced in object-oriented method will be less application domain dependent than in functional approach.

Reusable Modules

Another quality that suffers from heavy application domain dependency is reusability. Different point of view will have different understandings over the functions. Therefore, different software systems will have quite different data structure even their domain objects are somewhat similar or the same. Thus reusing any data structure or program module may require major re-writing. Usually the cost is so high that no one tries to reuse programs written by others.

In case of object-oriented design, reusability can be enhanced in many ways. The generalization provided by inheritance allows the capturing of commonalities within a group of implementations. This becomes a higher level abstraction to be reused later. Also the use of inheritance for specialization helps to reuse the

already developed behaviour of a class, making the class derivation an important programming tool. The concept of generics or parameterized types are very useful for reusability in strongly typed systems.

Design Process

The functional design method takes a top-down step wise refinement approach. The initially defined top level function is refined to simple functions. The process is not iterative. This may be quite easy and natural for some systems, but may not be for others.

The object-oriented design process is bottom up to a great extent. The potential objects are initially identified. Then the knowledge of each object and their responsibilities are defined, which takes a major part of the effort. During this process, objects having large amounts of responsibilities are decomposed resulting in new simple objects. Thus the design as such is an iterative process. Finally, the objects are connected by figuring out the relationship between them. This final step gives the solution to the problem, whereas the initial steps are all focused on the modelling of the problem domain. Apart from this, the object-oriented development process is iterative. This has the advantage of avoiding early binding of any critical decision. This also tightens the coupling between the specificati n and the implementation and helps to reduce the inconsistency between them.

Other Issues

One of the assets of object-oriented paradigm is that, in each phase of software development(*i.e.*, analysis, design and implementation) the focus is on a common set of items - objects. This brings out coherence and integrity in the development process and makes object-oriented approach an unified paradigm. The combination of object-oriented analysis, design and programming allows the capture and encapsulating of abstraction directly into code.

A modelling point of view is taken in object-oriented approach. Analysis and design phase work together to depict a model of problem domain. The information developed in analysis phase forms an integral part in the design phase. The main component of the solution to a problem is derived from the message interaction

among the independently implemented classes. This is different from the traditional approach where there is a shift from problem domain in analysis phase olution domain in design phase.

An important requirement in effective software project management is conformance to the **open-closed principle**[Mey88]. That is, a module should be both open and closed. A closed module is one that is ready for use by its clients. An open module is one that is still subject to extension. This is not supported by conventional programming. But the object-oriented methodology solves this problem with an effective tool ie. inheritance. The use of inheritance coupled with polymorphism and dynamic binding in object-oriented approach, facilitates extensibility and evolutionary development.

Also, if we look at the variation of cost with the software development experience, there will be a major gain in object-oriented approach. The gain in conventional approach could only be due to improved design decisions. But in object-oriented approach, apart from the improvement in design decisions, the reusability will play a major role in reducing the cost as development experience increases. The object-oriented paradigm provides good support for design issues such as modularity, information hiding, weak coupling, abstraction, strong cohesion, extensibility and composability.

One of the disadvantages of object-oriented paradigm is that it is difficult to understand the overall goal of the software. This may be due to the extensive decentralized style of object-oriented paradigm. Each class may be understood independently; but collectively, the function provided by a set of classes is difficult to make out. But proper high level documentation (not associated with any class) and features in the compiler to display the client view, heir view *etc.*, will help to solve this problem. The presence of the condensed 'top level function' of a software is necessary, at least in documentation.

Unlike function-oriented paradigm, where formal methods have been widely accepted and applied, formal approach to OOD is only at the beginning of the development[AGo91, Bre91, Dah90]. In fact, some techniques in VDM specification have been adopted by object-oriented languages like Eiffel[Mey88].

Finally, due to the high initial costs, object-oriented paradigm doesn't fare well for systems which are small and those which have fixed solutions. For example, to implement an algorithm or for a computation intensive problem the conventional approach is better than object-oriented paradigm. Also an object-oriented approach relies heavily on supporting tools and development environment.

In favor of functional-oriented approach we remark that there are a number of analysis methods, supporting tools, and implementation experiences which may be used in software development.

4

Transforming VDM Specification into C++

This chapter describes the details of transforming a VDM specification into an object-oriented design. Both methodology and implementation issues are addressed. The discussion is basically programming language independent, although we use C++ as an idea carrier. To begin the chapter, we first state the goals of the transformation; general transformation issues are discussed in the second section, and then the subsequent sections illustrate the handling of each element of VDM specification.

4.1 Introduction

It is our thesis that we can combine the virtues of object oriented approach and formal method (VDM specification). This makes it possible to use the modelling methods built for function-oriented approach, which are widely accepted and applied, for object-oriented design. In fact, with the result of function-based analysis technique VDM, the tasks accomplished in the steps of the responsibility-driven software design can be achieved from VDM specifications with additional analysis and effort.

The basic task of this study is to implement the transformation of a piece of VDM specification into an object-oriented implementation design. The result of this transformation will achieve the following goals:

1 Determine the classes that are necessary to model the physical objects. Provide rules to implement the variable types by means of classes.

2 Provide guidelines to implement the built-in operations for VDM variable types.

3 Identify the member functions needed to perform the operations that VDM specification indicated.

4 Provide guidelines to make use of inheritance, polymorphism, and other concepts.

4.2 Transformation Process in General

There are many ways to transferring a specification into an object-oriented design. The approach presented here is a straightforward method, which is a cookbook to transfer the data types, operations and functions into the classes and member functions according to their relationships.

In most cases only a guideline for creating classes and member functions is provided. More detailed issues are open to the implementer to fill in. The results we obtain may be a mixture of analysis, design, and implementation.

The emerging object-oriented design techniques also lead to similar results. The steps of analysis, design, and implementation are still necessary, but the separation between them is blurred.

The basic feature of object-oriented design goes around the class concept. There are two interpretations of the class concept[Bud91]. Here we use the explanation given by [Str91], that considers a class is 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[Str91]. We will not distinguish the terms of a class and a type.

In VDM specification, variable and its type are abstractions of the physical object. The relation between a variable and its variable type is the same as that of an object and its class in object-oriented programming languages. Variables in VDM specification can be transferred to objects of the classes that are obtained from the variable type transformations.

However, in object-oriented design, a class is more than merely an abstraction of a physical object. The access to data of a class is restricted to a specific set of access functions known as member functions. Member functions shall be contained as part of a class declaration. A class as a basic unit of the system, in an object-oriented point of view is an abstraction of a physical object and its connection to the outside world is only through its member functions.

Designing, in object-oriented view, is a process of establishing classes, data fields and member functions. Our central task in this study is to create classes, to identify their data fields and member functions from an established VDM specification. The whole process can be seen as a transformation from VDM specification into object-oriented design.

Any variable type in VDM specification can be implemented or simulated in programming languages such as C++. It is not necessary to implement a variable type in C++ in the same way as it is defined in VDM. There are ways to implement any variable type of VDM according to developer's choice. However, in this thesis we are going to provide a straightforward way to transform each of VDM variable types into a correspondent class in C++. We try to provide as many details as possible according to the known features of the VDM specification and the language C++ which supports many suitable features to implement the data types and the built-in operations in VDM specification language.

First, each variable type can be transferred into a class in C++.

A class is a representation of a data type. Although some basic variable types in VDM specification can be implemented in C data types, transferring them into C++

classes will ensure the features defined in VDM specification by using initialization checking to avoid accident violation of value. For example, transferring *nat*0 into *unsigned int* will be all right, but transferring *nat* into *unsigned int* will not prevent a variable of type *unsigned int* to be assigned value zero. On the other hand, a class *nat* with initialization checking and member function checking will effectively eliminate this possibility.

The relationship between variable types in VDM specification can provide helpful information about the relationship between the corresponding classes.

The so-called *is-a* and *part-of* relations can be detected and found in the original variable type definitions in VDM specification.

Second, the invariant checking of VDM specification can be implemented or partly implemented by C++ initialization mechanisms, though basically invariant conditions are design issues rather than implementation issues.

Third, even the operation and function specifications can reveal the relationships between classes in an object-oriented design from VDM specification.

Analyzing the operation specification is necessary to identify the member functions of the classes. The input parameters and the external parameters of an operation determine the classes with which it is involved.

In an object-oriented paradigm, the classes are the center of the design. Operations allowed on the data types are implemented by member functions of the class. The correspondence between operation and member function may not be one to one. The member functions may be fundamental operations permitted on a class. Of course, the operations specified in VDM will be among the member functions of classes.

In VDM, an operation specification states the constraints it is subjected to, but it does not restrict how these constraints are illustrated. Two operations may have the same complex expression as part of their pre-conditions or post-conditions. We call this specification redundancy. Specification redundancy is not uncommon among VDM specifications. In function-oriented approach it is expected to divide

complex operations into smaller and more basic operations. We hope that the specification redundancy can be eliminated or at least reduced in the design stage as much as possible.

The transformation is not a static process. Each step can interact with each other. Each step will form a part of the whole object-oriented design. Each step will supply certain information to different aspects of a complete design. Until the whole process finishes, the components will not be clear.

If a variable type A in VDM specification is transformed into a class A, we say that $\bar{A}$ is a transformation of A and A is the *pre image* of A. We can also call A the *image* of A. We may ignore these different faces of an abstract object in case that no confusion could occur. In fact, variable type in VDM specification and class in C++ are both models of abstract objects.

In the following sections, we will discuss the details of transferring of each element in VDM specification into object-oriented features in C++. The important topics of object-oriented design such as inheritance, data abstraction, *etc.*, are discussed. Some implementation design issues are also considered.

4.3 Transferring Basic Types

Let us look at five basic types in VDM specification: *boolean*, *int*, *nat*, *nat*0, and *string*. There are many ways to implement them in C++. We have adapted a simple but natural approach. Of these types, *int* and *nat*0 can be represented by C data types *int* and *unsigned int* in full observation of the original requirements, except conceptual differences(for example, *int* of VDM is an infinite set and *int* of C++ is a finite set).

The type *nat* has one more restriction compared with *nat*0, it can not assume value zero. Other features and operations are identical. So, we can define *nat* as a derived class from base class *nat*0. The non-zero restriction can be ensured by initialization and member function implementations. Or, this invariant condition

can be created as a member function to check if it is obeyed or not. In Section 4.5, we have detailed discussion about invariant conditions.

The most straight forward way to transfer *boolean* is to define an enumeration type with the same name and with the same definition as

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

In practice, *typedef* is used to declare a boolean type as an equivalent type to integer type in C, and define *false* as zero and *true* as one. The reasons are 1) C treats an enumeration type as an integer type; 2) in the above enumeration definition, *false* and *true* also assume value zero and one respectively; 3) the *if* statement will treat all non-zero valued expression as true in C and this fact is handled by the integer definition of *boolean* type very well. 4) the enumeration definition only contains two values and does not match the *if* statement completely.

The *string* type can simply be transferred to a string class. The common operators such as =, [], ==, and != (assignment, element, equality, and inequality respectively) can be defined as member functions. Different assignment situations should be considered to make it both flexible and convenient for usage.

A possible implementation is shown in [Listing 4.1].

▷Listing 4.1◁ Implementation of Class *String* in C++

```
class String {
    char *cptr;   // pointer to string contents
    int length;       // length of string in characters
public:
            // three different constructors
    String(char *text);        // constructor for existing string
    String(int size = 80);     // creates default empty string
    String(String& s);         // for assignment from another
                               // object of this class
    ~String() {delete cptr; };     // inline destructor
    int len (void);
    String operator+ (String& Arg);  // concatenation operator
    char& operator[](int i);         // index operator
```

```
    friend int operator==(const String &x, const char *s)
      { return strcmp(x.cptr, s)==0; }

    friend int operator==(const String &x, const String &y)
      { return strcmp(x.cptr, y.cptr)==0; }

    friend int operator!=(const String &x, const char *s)
      { return strcmp(x.cptr, s)!=0; }

    friend int operator!=(const String &x, const String &y)
      { return strcmp(x.cptr, y.cptr)!=0; }

};

String::String (char *text)
{
   length = strlen(text);  // get length of text
   cptr = new char[length + 1];
   strcpy(cptr, text);
};

String::String (int size)
{
   length = size;
   cptr = new char[length+1];
   *cptr = '\0';
};

String::String (String& s)
{
   length = s.length;                // length of other string
   cptr = new char [length + 1];     // allocate the memory
   strcpy (cptr, s.cptr);            // copy the text
};

String String::operator+ (String& Arg)
{
  String Temp( length + Arg.length );
  strcpy(Temp.cptr, cptr);
  strcat(Temp.cptr, Arg.cptr);
  return Temp;
}
```

```
int String::len(void)
{
   return (length);
};

void error(const char* p)
{
    ceer << p << '\n';
    exit(1);
}
char& String::operator[](int i)
{
    if( i<0 || length<i) error("index out of range");
    return cptr[i];
}
```

□

Please note the concatenation operation "+".

4.4 Transferring User Defined Types

We have eight kinds of user defined variable types in VDM specification: enumeration type, rename type, union type, record type, power-set type, list type, mapping type, as well as undefined type.

4.4.1 Enumeration Types

It is natural to use an enumeration type in C to represent an enumeration type in VDM specification. There is no difference in the definition. The range form of the enumeration type in VDM specification is also transferred into an enumeration type in C.

4.4.2 Rename Types

Rename type usually comes with two possibilities: 1) the new type is exactly the same as the original type, and 2) the new type is slightly different and is usually a subset of the original type. The difference is not big enough to demand the introduction of a different type.

In these two cases, transferring the rename type as a derived class from the base type can meet the requirements.

4.4.3 Union Types

Union type means that a broader set comes from two or more sets. That is to say the component sets are a subset of the union type. This is exactly the *is-a* relationship extensively discussed in object-oriented design[Bud91].

So transferring the union type as a base class and the components type as derived class is a sound choice. The base type only contains the fundamental data and member functions common to all component types. Each derived class can have more data and more member functions, even new definitions for member functions already defined in the base class (function overloading).

▷Example 4.1◁

The type *borrower* in [ALP91] is defined as:

```
Borrower  = Faculty | Student
```

In the design, a *borrower* class can be created and the necessary data fields and member functions established to associate with it as shown below:

```
enum borrower_status {F, S};

class borrower{
    char * name;
    unsigned ID;
```

```
    borrower_status   type;
  ...  ...
};

class faculty :  public borrower{
  ...  ...
public:
    faculty(char* n, unsigned id) :  borrower(n, id, F);
  ...  ...
};

class student :  public borrower{
  ...  ...
public:
    student(char* n, unsigned id) :  borrower(n, id, S);
  ...  ...
};
```

The classes *faculty* and *student* will inherit data fields and member functions of the base class borrower. The initializations will ensure the status being correct. The member functions are to be determined and filled in the later stages of design.

□

4.4.4 Record Types

The definition of a record type is very similar to that of a class definition in C++ except without member functions involved in record type definition. It is easy to transfer a record type in VDM specification into a class in C++.

The components of a record type are transferred to the corresponding components of the class. Between the class and the component classes, the so-called *part-of* relation holds obviously.

It is also possible to make use of the concept of multiple inheritance, a feature provided by C++ to support deriving a class from more than one different base class.

▷Example 4.2◁

Suppose there is a record type *MCIRCLE*.

```
MCIRCLE ::  CIRKLE  : CIRCLE
            MESSAGE : GMESSAGE
...
```

An implementation in C++ will create first two classes: the *Circle* class which defines an abstract circle, and the *GMessage* class which defines an abstract type to display a message on the screen. Both classes have a member function *show*: *Circle* :: *show* displays the circle, and *GMessage* :: *show* displays the message.

Using multiple inheritance, a *MCircle* class is derived from both base classes and inherits all variables and functions from both classes.

Without multiple inheritance, *MCircle* type can only be created by part-of relation. In that case, two component variables shall be defined. One shall be of type *Circle* and another *GMessage*.

One object of *MCircle* can be created and show function called like:

```
MCircle Large(250, 250, 225, GOTHIC_FONT, "Universe");
Large.Show();
```

Note that both base classes have a common member function *show*() with different actions though.

Listing 4.2 shows how this piece of design is implemented and multiple inheritance is used.

▷Listing 4.2◁ Multiple Inheritance in C++

```
...
class Circle :  public Point  // Derived from class Point and
{                             // ultimately from class Location
protected:
```

```
   int Radius;              // other component
public:
   Circle(int InitX, int InitY, int InitRadius);
   void Show(void);
};
...        // implementation of constructor and function show

class GMessage :  public Location
{               // display a message on graphics screen
   char *msg;               // message to be displayed
   int Font;                // font to use
   int Field;               // size of field for text scaling

public:
   // Initialize message
   GMessage(int msgX, int msgY, int MsgFont, int FieldSize,
            char *text);
   void Show(void);         // show message
};
...        // implementation of constructor and function show

class MCircle :  Circle, GMessage  // multiple inheritance
{
public:
   MCircle(int mcircX, int mcircY, int mcircRadius, int Font,
           char *msg);
   void Show(void);               // show circle with message
};

           //Member functions for MCircle class

           //MCircle constructor
MCircle::MCircle(int mcircX, int mcircY, int mcircRadius,
            int Font, char *msg) :  Circle (mcircX, mcircY,
            mcircRadius), GMessage(mcircX,mcircY,Font,
            2*mcircRadius,msg)
{
}

void MCircle::Show(void)  //MCircle Show function
{
   Circle::Show();
```

```
    GMessage::Show();
}
```

□

However, not every record type can be transferred to class with multiple inheritance definition. Only those record types whose component variables are of different types can be implemented in a class of multiple inheritance.

Because, if a record type with two variables have the same type, it is not possible to distinguish which member is for which variable, if the class is to inherit from this class twice.

▷Example 4.3◁

Suppose that a class *DCIRCLE* has two circles instead one: One external circle and one internal circle. If *DCIRCLE* is allowed to inherit from *CIRCLE*, it will not be clear whether *dcircle.Radius* means the radius for the external circle or the internal circle.

□

When multiple inheritance is properly used, a subtle but nevertheless important change in the view of inheritance takes place. The *is-a* interpretation of inheritance, used in single inheritance views a subclass as a more specialized form of another category, represented by the parent class. When multiple inheritance is used, a class is viewed as a *combination* or collection of several different components, each providing a different protocol and some basic behavior, which is then specialized to the case at hand[Bud91].

4.4.5 Power-set Types, List Types and Mapping Types

The variable types in VDM are not supposed to impose any restriction by any means on how they will be actually implemented in a programming language. There are many ways to implement composite type of VDM in C++. Different emphasis,

like efficiency, easy for reuse, flexibility, can lead to quite different implementations. For example, a set can be implemented as an array, or a linked list.

What we have presented here is a straightforward implementation according to the definition. The advantage is that the implementation allows maximum flexibility and requires minimum modification to a particular system. The disadvantage is that efficiency may (not necessarily) be damaged. However, we believe that the flexibility could compensate the overall achievable efficiency. In fact, efficiency can be attempted at later stages of design.

There are two ways to build classes in C++ to implement power-set types, list types, and mapping types. These classes are usually known as container classes, since these data structures are all used to maintain collections of elements.

One way is to make use of subclass coupling to create general-purpose classes even in the presence of strong typing. The static typing of C++ implies that container classes can not hold arbitrary objects, since the type of the values held in the class must be known at compile time. To overcome this difficulty, a basic class shall be defined. It will be a parent class for any value held in a container.

A very important advantage of this approach is that there are several commercial class libraries ready for use. Container classes for different base type can be formed easily although more class names will be created. Built-in operators in VDM specification for power-set type, list type and mapping type all can be implemented by container classes.

Another way to implement is to use the ability to parameterize class descriptions, which is currently supported by C++. Using parameterized classes, class template in C++ term, the user declares a class template in which the types of various fields are left unfilled. A parameterized class can be thought of as a family of classes, where the parameter value is used to generate specific qualified versions of the class. Specific instances can then be created from the qualified classes.

By making use of the template concept, power-set types and list types can easily be transferred into container classes in C++. A template allows container classes to be simply defined and implemented without loss of static type checking

or run-time efficiency. It also allows generic functions, such as *sort*(), to be defined once for a family of types.

Built-in functions in VDM specification such as *dom*, *rng*, *card*, *hd*, *tl*, *len*, *etc*, can be transferred into member operator functions of the template. They can still be used as prefix operators.

Listing 4.3 shows the definition of a template *stack* in C++.

▷Listing 4.3◁ A Template Class *stack* in C++

```
template<class T>
class stack {
      T* v;
      T* p;
      int sz;

public:
      stack(int s) { v = p = new T[sz=s];}
     ~stack() { delete[] v;}

      void push(T a) { *p++ = a; }
      T pop() { return *--p; }

      int size() const { return p-v; }
};
```

▷Example 4.4◁

To declare a variable of type *stack* for integer stack, one writes

```
stack<int> si(100);   // stack of integer of size 100
```

□

The common operators are also handy to be defined as member functions.

Since friend functions allow more flexibilities especially when implicit type convention is involved, for practical considerations, some built-in functions can also be defined as friend functions instead of member functions.

The interfaces for the power-set template, the list template, and the mapping template as well as their associated functions are given in Listing 4.4, 4.5,and 4.6. The interface, in object-oriented terminology, is the view that the user of object sees how the object is interfaced to the world at large. It is precisely the class definition part, which provides the intended user with all the information needed to manipulate an instance of the class correctly, and with nothing more. Complete listing of the source code can be found in many commercial C++ compilers, such as **Borland** C++ 3.0.

First is the interface for the list type. There are many ways to implement a list. Here a single linked list is defined. [Listing 4.4] shows what its interface looks like.

▷Listing 4.4◁ Interface of Class Template *Slist*

```
template<class T>
class Slist :  private slist_base {
public:

    void insert(T& a);       // insert an element at head
    void append(T& a);       // insert an element at tail
    T get();                 // get first and remove from list
    T operator hd();                 // get first element
    Slist operator tl();             // remove first element
    int operator len();              // length of list
    T operator[](const int& index)   // index operator
    Boolean includes(T& a); // check if a is in list
    Boolean ==(Slist<T>& a);// equality operator
    Boolean !=(Slist<T>& a);// inequality operator
    Set<T> operator elems();         // obtain a set of elements
};
```

□

The base class *slist_base* is an auxiliary class to implement *Slist*. *Slist* inherits all the member functions of *slist_base*. The implementation details of *slist_base* are omitted here.

The set class template is similar to the list class template. In fact the set class template can be implemented in the same way as the list class template although they are quite different objects mathematically. The only thing needed to be modified is that each time an object is inserted, it shall be checked if the object is already in the list; If it is found then nothing is done; otherwise it is inserted. Listing 4.5 is the interface for set template.

▷Listing 4.5◁ Interface of Class Template *Set*

```
template<class T>
class Set :  private Slist {
public:
    void insert(T& a);          // insert an element
    void remove(T& a);          // remove an element
    int operator card();                // length of set
    Boolean isam(T& a);         // check membership
    Boolean ==(Set<T>& a);      // equality operator
    Boolean !=(Set<T>& a);      // inequality operator
    Boolean <<=(Set<T>& a);     // subset or equal operator
    Boolean >>=(Set<T>& a);     // supset or equal operator
    Set +(Set<T>& t);           // union with a set of same type
};
```

□

The mapping class template uses an auxiliary template *Link* to implement it. Listing 4.6 shows the interface of the template.

▷Listing 4.6◁ Interface of Class Template *Map*

```
template<class K, class V> class Map {
    Link<K,V>* head;
    Link<K,V>* current;
    V def_val;
    K def_key;
    int sz;
```

```
    void find(const K&);
    void init() { sz = 0; head = 0; current = 0; }

public:

    Map();                          // constructor
    Map(const K& k, const V& d); // constructor
    ~Map();                         // delete all links

    Map(const Map&);
    Map& operator= (const Map&);

    V& operator[] (const K&)        // map applied to element

    int size();                     // size of the pair
    void remove(const K& k);        // remove pair

    Set<K> operator dom();                  // domain set
    Set<V> operator rng();                  // range set
    Map ++(Map<K, V>& m);           // over write a map of same type
    Map operator\(K& k);            // strict by operator
    Map operator/(K& k);            // strict to operator
    BOOLEAN ==(Map<K,V>& m);        // equality operator
    BOOLEAN !=(Map<K,V>& m);        // inequality operator
```

□

4.4.6 Undefined Types

Undefined types are open to the later design considerations as the components to surface. A class without data can be created from an undefined type.

4.5 Mapping Invariants

Invariants stated in the specifications should be carried over to the design in order to ensure consistency and correctness of the design[ALP91]. Invariants assert the static relationships among certain global variables. Although many software design methods do not include a separate section for invariants checking, we insist that the invariants be distributed among the classes in order to assert the validity of every operation in the design. Fortunately, C++ provides facilities that easily lead us to implement the invariants.

Invariants are expressed in VDM specification by conjunctive clauses. An invariant may also involve many components of a class. However, an invariant is an organic whole. And the predicate condition is meaningful only in the level of that class.

4.5.1 Invariant Implementation

Since each class will take responsibility for its own data consistency and validity in C++, we can have the spirit of invariants carried over to the initializations and member functions of the class. Or a separate member function can be defined to check the invariant conditions.

In fact, "the purpose of initialization is to establish the invariance for an object" [Str91]. Each operation on a class can assume that it will find the invariant true on entry and must leave the invariant true on exit.

Let us see an example of the *nat* class, which corresponds to VDM built-in type `nat`. The invariant is that the value of an object shall be greater than 0. The operators that are possible to break the invariant are '--'(difference) and '/'(divide). The code in Listing 4.7 checks the invariant condition where these operations are carried out implicitly.

▷Listing 4.7◁ Implementation of Class *nat*

```
#define inv_err     printf("invariant violation")

class nat
```

```
{
private:
   int val;

public:                                              // interface
   nat();
   nat(int a);
   nat(const nat0& b);
...
   boolean inv();
   int value();
   friend nat operator+(nat&, nat&);
   friend nat operator+(nat0&, nat&);
   friend nat operator+(int&, nat&);
   friend nat operator-(nat&, nat&);
   friend nat operator-(nat0&, nat&);
   friend nat operator-(int&, nat&);
...
   friend nat operator/(nat&, nat&);
   friend nat operator/(nat0&, nat&);
   friend nat operator/(int&, nat&);
...
}
                              // implementation
nat::nat() { val = 1; }       // default initial value 1
nat::nat(int a) { if(a>0) then val=a; else inv_err; }
                              // initialize an integer into a nat
nat::nat(const nat0& b) { if(b.value()>0) then val=b.value();
      else inv_err; }         // initialize a nat0 object into a nat
...
boolean nat::inv(){if(val>0) then return true; else return false;}
                              // invariant checking
int value() return val;       // return value, user can't change it
nat operator+(nat&, nat&)
...
nat operator-(nat& a, nat& b) return a.val-b.val;
nat operator-(nat0& a, nat& b) return a.value()-b.val;
nat operator-(int a, nat& b) return a-b.val;
...
nat operator/(nat& a, nat& b) return a.val/b.val;
nat operator/(nat0& a, nat& b) return a.value()/b.val;
nat operator/(int a, nat& b) return a/b.val;
```

...

□

Class *nat* has three constructors: the first initializes an object without parameter and the default value is 1; the second initializes an object with an integer and the invariant is checked; and the third initializes an object from an object of type *nat*0 and the invariant is also checked.

For invariant checking, an *inv* function is created. It shall be called by any function (other than '−' and '/') whose actions may lead to the invariant breaking.

Note that the operator '−' does not escape from invariant checking. The return value is of the type *int*. In fact it will be converted to the type *nat* from the type *int* by the second constructor, which has a checking mechanism.

4.5.2 Handling State Invariant

As to the state invariant in VDM specification, there are two ways to deal with it. The first approach is to treat the state as a special object in VDM and distribute the invariant among the post images of the global variables[ALP91]. The second way is to handle the state as a record type.

The way to distribute the state invariant is to divide the invariant according to the conjunctive assumptions.

The state invariant is composed of several predicate clauses linked by "and" symbol ∧ (& in our machine symbol). Each clause states a constraint over the values that the global variables can assume. Each clause involves some variables of some variable types.

The best case is when a clause involves only one variable type. We can consider the clause as an additional invariant condition over the variable type besides its own invariant condition imposed by its structure. In this case the final invariant condition for the class (the post image of the type) will be composed of two parts: one comes from the state invariant, and the other comes from its original invariant.

In the case when more than one variable type is involved in a clause, the least super class of the classes involved shall be found, under whom the clause is placed as an additional invariant condition.

The advantage of this approach is that consistency of the system is preserved by each class member function. No other explicit invariant checking is needed for the state and the checking operation is optimized.

The disadvantage of distributing state invariant to classes in the design is that the additional invariant condition for a class which comes from the state invariant prevents further reuse of the class. Since that part of the invariant is not common among different systems while the original invariant condition which comes from the modelling of the class itself, is a characteristic of the class and is reusable to other systems.

▷Example 4.5◁

A *school* type is composed of *faculty* type, *student* type, and *staff* type. A *student* type is composed of *name* type, *age* type, *sex* type, and *ID* type. Suppose our system is about a school for continuing education. One of the constraints in the state invariant is that student should be at least 18 years old. If we distribute this condition and create an invariant in class *student*, then class *student* can not be reused by a high school, where the constraint is different.

□

The other approach, which treats the state as a usual record type, improves reusability. From the syntax point of view, the state is just a record type with a name which appears in all the VDM specifications. A corresponding class can be obtained from the procedure introduced in the previous section. Not only the member variables, but also the invariant function and other member functions can be created for the class. (Member function creation will be discussed in the next section). An explicit invariant checking function can be built in the same way as

discussed in the previous section. To investigate the system consistency, a message is sent to the object of class state, *i.e.*, a function call like *Stat.inv*() in C++.

The disadvantage of this approach is that an explicit call to the invariant function of the state class is required to keep the system consistent, and the state invariant checking function takes more calculations than its counter part in the first approach.

4.6 Member Function Creation

A class without member function has no value in practice. The classes created from variable types in VDM specification are not complete unless their member functions are identified and created. This is by far the most difficult task in this study. The member functions are used to implement the operations specified in VDM specification.

Remember that the member functions define possible operations affecting the object. We believe that the operation specification in VDM contains sufficient information of the kind of operations that are required to act on an abstract data type, namely a class (in design).

Analyzing the predicate clauses in pre-conditions and post-conditions of an operation specification will reveal the member functions required to implement the operations.

The concrete measures are:

1 dividing the pre- and the post-conditions into separate clauses,

2 categorizing each clause into different classes according to the variables they affect, and using these clauses as specifications to define member functions in the classes,

3 reorganizing member functions generated so far to eliminate any possible duplication, and

4 implementing operations by member functions created during the previous steps.

It is important to note that we are dealing with operation specification instead of an operation procedure itself. What we can obtain is member function specification from an operation specification.

In the following, we will detail each step and discuss problems that may occur.

4.6.1 Decomposing Pre- and Post-Conditions

The task is simple here. We assume that the pre- and post-conditions of the operation specification are in conjunctive normal form. This assumption is vital. If the specification is written in a form that sub-operations can not be distinguished in the syntax analysis phase, *i.e.*, if the predicate is not in the form of $A \wedge B \wedge \cdots W$, the member function creation process described in this study will generate condensed functions which contain too many fundamental operations and can not reveal sufficient clues to implement them.

The clauses appearing in the pre-conditions are assumptions about the status of arguments before an operation can be validly taken. When the pre-condition of a specification fail to hold, the operations will be meaningless.

The clauses in post-condition describe the status of arguments after the operation and define how the output is related to other parameters. Sometimes values of arguments both before the operation and after operation may occur in the clause.

Each clause appearing in a pre-condition or in a post-condition states the relation between its variables.

An operation is usually a composition of a series of sub-operations. We can assume each sub-operation only accomplishes a fundamental task. The specifications of these sub-operation will be much simpler than that of the composite. Conversely, if we have the specification of sub-operations, we can implement them just like that in a composite operation.

Each clause can be considered as a specification of a basic operation or function, no matter where the clause comes from.

The difference of an operation and a function in VDM lies in the access to the external variables in a VDM specification. We do not see any difference in a C++ implementation. So we will ignore this difference as we did before.

4.6.2 Creating Member Functions

The basic operations are taken as the basis to define member functions of the classes. The reason we say as the basis of defining member functions instead of as member function directly is that as to the member functions, these basic operations may still be a composite operation themselves. Even more fundamental operations may have to be performed by member functions. However from the VDM operation specifications, we can not determine how to divide them. The clauses we obtained from dismantling pre-conditions and post-conditions will be our smallest elements. So at this moment, we will create member functions according to the clauses.

The member functions created this way are still far from perfect. In fact, one of the biggest problems is their implementation.

▷Example 4.6◁

Clause A is defined as $a' = a + 5$, where a is complex. Now a function A is created as $A(a) : a' = a+5$ and one can interpret it as a function which increases a by 5. This may not be complete and adequate. More specific functions are needed. At least three operations are required. First, there is a need for **plus** operation for complex numbers. Since complex is not a built-in type in C++, it shall be implemented by developer. Second, a type conversion function is needed to convert integer 5 to complex. Otherwise, it is still not able to compute $a + 5$ using **plus** operator. Or an additional function for **plus** operation that can take a complex and an integer as operants. And third, since the clause may be understood as a predicate

stating that the value of a' shall be equal to that of $a + 5$, an equality operator is needed for two complex numbers.

□

Mechanical transformation can not provide these kinds of information. Member functions shall be implemented by developer after transformation. More functions may be created in later stages of design. However, transformation result shall give specifications for these member functions, which serve as basis of implementation.

Another problem is the naming of member functions. Since each clause has its own semantic meaning, a mechanical convention is difficult to establish to name these member functions according to their meanings. Fortunately, at the very beginning of creating member functions, the names are not of great importance. Well sounding names can be obtained after human interventions.

A clause may access one or more parameters and external variables of the operation. According to the variable or variables accessed by a clause, member function is created for the class (post-image of the variable type). Hereafter, when we talk about the classes, we are talking about the classes whose pre-image variables appeared in the clause.

▷Example 4.7◁

A clause $C(a, b)$ may appear somewhere in a predicate, where variable a is of type A, b is of B. The post-images of A and B are $\bar{A}$ and $\bar{B}$ respectively. In discussion, we refer a and b as variables of $C(a, b)$, A and B as variable types, and $\bar{A}$ and $\bar{B}$ as classes.

□

For each variable that is affected in the clause, one member function is created in the post-image class of the variable. The components of a variable are also accounted as a variable (of the class the components are) if the components are involved in the clause too. If the component is a power-set type, or list type, or

mapping type, since the functions of these classes are defined already in their class templates, they can be ignored. Assuming that operations acting on a built-in class (or type) have already been defined, we omit the process to create member functions for them.

▷Example 4.8◁

Consider *BORROW_BOOK* operation of the library system[ALP91]. The pre- and post-conditions as stated in [ALP91] are not in conjunctive normal form, and we rewrite them as follows:

```
BORROW_BOOK(ID : IDtype; CN : CNtype)
/* User with ID number ID borrows book with call number CN */

ext lib_system : wr Library
pre
                    /* borrower must be eligible to borrow */
(∃!u∈REG_USERS(lib_system).(IDNUMBER(u)=id) ∧
    (id∈ dom LOANS(lib_system)⇒
        (USTATUS(u)="F"⇒card dom LOANS(lib_system)(id)<20) ∧
         USTATUS(u)="S"⇒card dom LOANS(lib_system)(id)<10)))) ∧
                    /* book shall be eligible to be borrowed */
(∃!b∈ COLLECTION(lib_system).(CALLNUMBER(b)=cn)∧
    (BSTATUS(b)="inshelf")∧
    (∼(∃u1∈USERS(lib_system).cn ∈ dom LOANS(lib_system)(u1)))) ∧
                    /* borrower must be 1st on reserve queue */
(cn∈ dom RESERVED(lib_system)⇒ hd RESERVED(lib_system)(cn)=id)


post

                    /* mark the book loaned out */
(∃!b∈ COLLECTION(lib_system).
    (CALLNUMBER(b)=cn)∧(BSTATUS(b)="loanout")) ∧
                    /* update borrower's record */
(∃!u∈REG_USERS(lib_system).(IDNUMBER(u)=id) ∧
    (LOANS(lib_system)(id)'=LOANS(lib_system)(id)++[cn→today])) ∧
                    /* borrower moved from reserve queue */
```

(cn∈ dom RESERVED(lib_system)⇒id∉ elems RESERVED(lib_system)(cn))

There are three clauses in each of the pre-condition and the post-condition.

The parameters are *ID:IDtype*, *CN:CNtype*, and *Lib_System:Library*. The post-image classes are *idtype*, *cntype*, and *library*. Some components of *library* and even their components are also involved in the pre- and post-conditions. They shall be considered as variables too. Their corresponding classes are *lib_books*, *borrower*, *borrower_status*, *book_status*, *loanmap*, and *queue*. The powerset and list classes as well as mapping classes are considered as container class, and the corresponding operators are defined already. They will not be taken into account here. *borrower_status* and *book_status* are of enumeration type, which is also considered as built-in type of C++. Therefore, we do not include them in our discussion.

The classes occuring in each clause are listed in Listing 4.8.

▷Listing 4.8◁ Class and Clause Relationship

clause	classes
pre 1	library, idtype, borrower
pre 2	library, cntype, lib_book, borrower
pre 3	library, idtype, cntype
post 1	library, cntype, lib_book
post 2	library, cntype, lib_book, borrower, idtype
post 1	library, idtype, cntype

A member function shall be created for each class listed above.

□

4.6.3 Reorganizing and Implementing Member Functions

The member functions created so far are far from perfect. A member function of a class is supposed to define an operation acting on the object of the class, *i.e.*, on some member variables. The clean up phase consists of the following steps:

First step eliminates duplicate functions caused by repeated clauses. Same clauses may appear in different operation specifications and certainly will cause duplicate member functions.

Second step redefines the member function according to the role that the variable plays in the clause. Different member functions created in different classes due to the same clause (from pre- or post-conditions) shall be redefined to accomplish quite different actions.

One principle of object-oriented design is that each class shall take care of the data consistence of its own and ensure only its member functions and friend functions have access to its data. If a class component is an object of another class, the access to the components of the object shall be carried out though the member functions of the component class.

The principle here in redefining a function in a class is that the function shall only act on the class itself.

▷Example 4.9◁

In the clause *pre_1* of *BORROW_BOOK*, the objects involved are *lib_system*, *id*, *u*, *ustatus*, and *loans*. The post-image classes are *Library*, *IDtype*, *borrower*, enumeration type *borrower_status*, and container class *loanmap*.

- For class IDtype, a function which returns the idnumber of its object shall be created by this clause.
- For class *borrower*, two functions may be created:

1. Idnumber checking functions,

2. the status of user (whether $USTATUS$ is "F" or "S").

- For class *Library*, a function to implement the specification of the clause is needed.

- Since *borrower_status* is an enumeration type and *loanmap* is a mapping class, the built-in functions are supposed to exist already, we need not consider them again.

□

It is possible that even for a single variable in the clause, different operations may be included in the clause. In Example 4.9, there are two member functions created for class *borrower*.

In fact, to split the clause into more basic operations is the main target here. Of course there may be further duplications. This further duplication elimination can only be performed by human interaction and is left to the developer in later implementation.

As stated above, one clause will lead to different member functions in different classes. The actual operation performed by these functions will be quite different from the original actions specified by the clause. However there will be one function which will perform the operation specified by the clause. We call the function as primary function for the clause and the class in which the function is defined as the primary class of the clause.

▷Example 4.10◁

Class *library* is the primary class of all the clauses in Example 4.8.

□

Here we give the rule to determine the primary class for a clause.

Rule 1. *The primary class is determined from the role of the corresponding variable in the clause:*

1 *If there is a write-access external variable, the corresponding class will be primary class.*

2 *If there is a read-access external variable and no write-access variable, the corresponding class will be primary class.*

3 *If there is no external variable, this is a function specification. We have two different situations.*

3.1 *If there is only one variable, the class will be primary class.*

3.2 *There exists a containing relation (to be defined in the next paragraph) among one parameter type and the rest of parameter types. The class which contains others is the primary class.*

3.3 *No such relation exists. One global function shall be created from the clause as the primary function.*

Definition 1. *Type A is said to contain type B, if*

1 *A is B;*

2 *A is a power-set type or a list type. The element type of A is type C. And C contains B;*

3 *A is a rename type, the original type C of A contains B;*

4 *A is a mapping type, the domain type C of A contains B; and*

5 *A is a record type, one of the component types C contains B.*

In a pre-condition clause, there will be no write-access variable, so the first Rule will not be applied. However, a write-access external variable for the operation is of priority to be a primary class if it occurs in a pre-condition clause.

⌐Example 4.11⌐

In MAIL system[CHJ86], function *is_in_list* is specified as:

$is\_in\_list : Uid \times Name\_list \times Nmap \rightarrow Boolean$

$is\_in\_list(user, names, directory) \widehat{=}$

$$user \in \{directory(n) | \ n \in \ elemsnames\}$$

Among three variable types: *Uid*, *Name_list* and *Nmap*, there is no type which contains the other two types. So function *is_in_list* shall be defined as global function.

□

The role that played by a primary class is like that of a coordinator. The classes involved in a clause are like team members. However, the team is so loosely organized that a hierarchy may not exist. In case there is a hierarchy, the primary class will be at top of the hierarchy, which is ensured by the selecting rule. The primary class will take the responsibility to coordinate the involved classes and achieve the operation specified by the clause.

The consideration to implement the clause specified in an operation by primary function in primary class is that although the other classes are involved in the operation which is not expected according to the principle of limiting the operation within the scope of a class itself, the degree of violation is to be reduced to the minimum by implementing it in the primary class. Note that this violation is necessary and permitted. It is done by sending a message to the server class. Since, in case of 1) in Rule 1 it will not change objects of other classes; in case of 2) only parameter classes are requested to report some thing. Note parameter variables are less important than external variable or they are in lower level of classes hierarchy; in case of 3.1). non-primary class is element class, or base class, or component class of primary class, or the same class as primary class. So non-primary class is considered reasonably effected by primary class.

This approach is different from a similar handling in [ALP91], where a least super class is used to replace primary class. The key reason for our approach is to provide reusability.

▷Example 4.12◁

There is no primary class in Example 4.11, so the function will be created as a global function and can be reused by other systems. If the function is implemented as a member function of the least super class *state*, its usage will be limited inside the MAIL system. The meaning of the function is obviously not limited inside the mail system. However, in order to reuse *is_in_list*, class *state* shall be reused and this may not be reasonable in some situations, because *state* is strictly for the MAIL system. However, a global function can be reused without concerning the MAIL system.

□

According to the responsibility separation principle, if a member function requests action on an object of another class, it can not have direct access to that object. Instead it sends a message to the object. In other words, it performs that part of action by calling one of the member functions of that class. At least the primary function is required to call functions of non-primary classes to perform operations involving other objects. By calling a function of other classes to perform certain operations, a function can be simplified indeed. If all the possible member functions were simplified this way, we have advanced one step more toward responsibility separation.

That is to say, between the primary class and non-primary classes, client-server relation exists. The primary class will be *client* and non-primary class be *server*.

▷Example 4.13◁

The member function *library* :: *borrow_book_pre_1* created from operation *BORROW_BOOK*, can be implemented by calling the member functions in class *IDtype* and *borrower* created from the same clause.

□

If one part of a clause is the same as an existing clause, or the entire clause is the same, no new functions shall be created and the existing functions shall be used through client-server relations (calling the primary function created by that clause).

There is a very important issue in judging if two clauses are the same or one is part of another. That is, the clauses as they are shall not be compared without semantics consideration. The variable names shall be replaced by their type names and the resulting clauses would be compared, since for any variable appearing in a clause, only the object which represents is significant in comparison and its name is not.

▷Example 4.14◁

In the mail system[CHJ86], operations *ADDUSER* and *DELUSER* both check whether the object user is in the list or not. However, the variable name for the user in *ADDUSER* is *newuser*, and in *DELUSER* it is *olduser*. It would be very difficult to tell the two clauses have a relation only by symbolic comparison of them.

□

In summary, there are two rules at this stage:

1 Implementing non-primary functions at the beginning. The first functions to implement shall be those that perform very fundamental operations. The relative complex ones are implemented later. The primary function is implemented at the end.

2 In implementing any functions, always trying to make use of existing functions by message sending and avoiding any function duplications.

4.6.4 Implementing Operations by Member Functions

After obtaining member functions for classes from each clause, we can implement the operations specified in VDM operation specifications by making use of primary functions.

As to the other operations in object-oriented design, VDM specified operations are also defined as member functions of some classes according to the parameters and the external variables accessed by these operations. In some cases they are defined as global functions because of similar reasons discussed above for clause created functions.

We have created many member functions for each class, which perform certain fundamental operations specified in the clauses of the pre-conditions and the post-conditions. Each primary function will perform the operations specified by that particular clause. By calling these primary functions (or in object-oriented design's term, sending messages to objects of these primary classes) in a sequence, the actions specified in the operations shall be performed. We call the process of determining the sequence and the messages as implementing the operations.

We summarize the steps to implement VDM operations as:

1 Creating member functions from clauses. This is described in the previous section. The detailed implementation of these member functions are achieved by human interaction. In the transforming stage, only specifications of these member functions are provided.

2 Implementing operations by primary functions. This can be done by collecting the primary functions of each clause appearing in one operation's pre- and post-condition.

The implemented operations are also subject to be categorized into classes according to the parameters to which they request the access. The Rule is similar to that of the primary functions with an extension to deal with the case that more than one external variable is in consideration. The following Rules shall be applied in the order of *write-access external variable, read-access external variable, and non-external variables.* That is: apply them for write-access external variables first; if no one Rule is applicable, apply for read-access external variable next, and so on.

Rule 2.

1 *If there is one and only one variable in write-access, or read-access, or parameter, the operation will be assigned to the corresponding class;*

2 *There exists a containing relation among one variable class and the rest of variable classes. The class which contains others is the primary class;*

3 *No such relation exists among the variables. One global function shall be created for the operation and it shall be declared as friend in each class with which it involved.*

▷Example 4.15◁

Operation *BORROW_BOOK* will be created in class *library* from its six primary functions. Since *library* is a write-access external variable type.

□

▷Example 4.16◁

The following operations are specified in MAIL system (see Appendices E and F).

Operation *PPOST* is created in class *UMAP*. Since it is the only external access class.

Operation *is_in_list* is created as global function, since there is no external access variable and no class contains other classes. This is the same result as discussed in member function implementation for clauses, because there is only one clause in this operation.

Operation *ADDUSER* is created as global function, since there are two write-access classes and none of them contains another class.

□

5

GAP — The Transformer

GAP system is built to implement the transformation method discussed in the previous sections. This section first introduces the system architecture and some system techniques unique to the transformation process. Then we discuss how the specification may be written for better transformation. The system output files are described in detail. The last part is a brief comparison with other existing systems.

5.1 GAP System Architecture

GAP serves two purposes as stated before: first as a syntax parser and partial semantics checker for the underlying VDM specifications, second as an automatic transformer to generate C++ classes. Although the syntactic checking process and the transforming process are both syntax-driven and there is no visible boundary between the two processes, we still can describe GAP architecture as in Figure 5.1.

The input to the system is a VDM specification file. One sample file is shown in Appendix E.

The system output is composed of three files. The most important one is the object-oriented design. These files are described later. One sample object-oriented design output file is given in Appendix F.

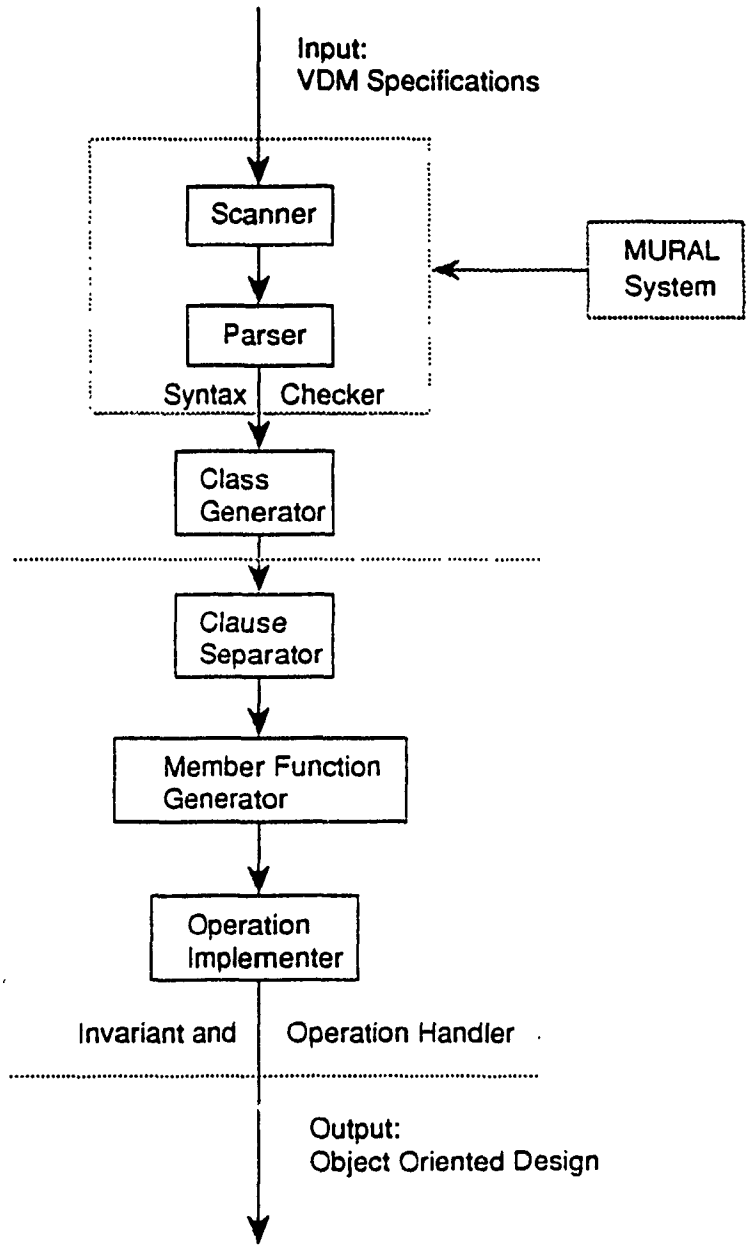

Figure 5.1 System Architecture of GAP

The syntax checker is composed of the scanner and the parser. The scanner reads the input of source specifications 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 The symbol table generating is the most important task of the parser.

The transformer consists of the class generator and the invariant and operation handler.

The class generator takes the information from the symbol table and generates classes for each VDM variable types. Any relationship between classes is detected by the class generator.

The invariant and operation handler can also be divided into three functional parts: 1) the clause separator; 2) the member function generator; and 3) the operation implementer. The clause separator divides predicate into clauses according to the syntactic analysis. The *if-then-else* structure is also divided to *if-clause*, *then-clause*, and *else-clause*. The member function generator determines the primary classes and generates member functions for each clause. It also checks the possible relationship between different clauses. The operation implementer determines the primary class for the VDM specified operations and invariants, and implements the operation or invariant by calling the primary function of each clause.

5.2 Some Special Handling Techniques

In many ways, GAP resembles a compiler without code generation for the target language. Based on the LL(1) grammar for a simplified VDM specification language, an one-pass approach for analysis and synthesis is adopted in the implementation. Due to the nature of the transformation, the information needed in the transformation is not totally the same as a conventional compiler.

Here we discuss and introduce some special techniques used in GAP. These techniques may not be seen in conventional compilers and may only arise in this transformation.

5.2.1 Clause Separation

It is hoped that the predicates be divided into fundamental clauses. Each clause only specifies a basic operation. The member functions of the classes are created from these clauses. It is also expected that each clause form a basic unit and preserve some semantic meanings.

The basic strategy in dividing a predicate is according to the conjunctive assumption. That is to find the highest level **ands**(&s) in a predicate and divide the predicate.

However, in the *if*-expression and the *let*-expression special measures are taken to further divide a predicate, although an *if*-expression or a *let*-expression can be seen as a complete clause.

For the *if*-expression, we divide it to *if*-clause, *then*-clause and *else*-clause. These clauses are also subject to further separation as an usual predicate.

For the *let*-expression, the *let*-part is first analyzed and put in a special template, and then the *in*-part is treated like an usual predicate and divided according to its syntax. Information about the temporary variables defined in the *let*-part are kept for each clause.

5.2.2 Clause Comparison

It is important to find whether the underlying clause is identical to or is part of an existing clause. While a complete analysis according to the semantics may be impossible to carry out, GAP performs the task by strict symbolic comparison.

The clauses to be compared are first cleaned by internal representation of tokens. Then the variable names in the clauses are replaced by their class names(type names). This unifies the clause to a standard expression.

5.2.3 Temporary Variable Handling

There are temporary variables in VDM specification in addition to the external variables and the local variables (the parameters of operation). They appear in the

quantified expression and the let expression.

The temporary variables themselves are not important in the transformation. However, the temporary variables in the let expression can be seen as a condensed variable which contains information about the presence of the external variables and the local variables. This is important in determining the associated classes and in determining the primary classes.

So, the temporary variables and the variables that compute the temporary variables are kept for each clauses of the let expression.

5.2.4 Member Function Naming

For each clause obtained from predicate, a sounding name is important to help user understand the function meaning. However, in this stage of transformation, it is impossible to determine the semantic meaning of a member function without human interaction. So, a mechanical procedure is developed to name the member functions (clauses). Since the member function is created from a clause. GAP gives the clause and the member functions it created the same name, even more than one function may be created for different classes.

The clause is named by combining the operation name, string "pre" or "post" according to the predicate and the clause index in that predicate.

▷Example 5.1◁

The clauses in Listing 4.8 are named as

□

BORROW_BOOK_pre_1,	BORROW_BOOK_post_1
BORROW_BOOK_pre_2,	BORROW_BOOK_post_2
BORROW_BOOK_pre_3,	BORROW_BOOK_post_3.

5.3 Desired Style of VDM Specification

Due to the loose structure and the abstract nature of a VDM specification, some information about the variable types or functions used in an operation specification may be very difficult to obtain on the fly with an one pass compiler. For example, it is a common practice to let the implementation dependent element to be decided later at implementation stages, and only use a commentary in the place during specification. The undefined type is an example in state definition, and the comment predicate is an example in operation specification. This feature is considered an advantage for abstract specification.

Another example is that any function can be used in an operation specification before its definition. It is absolutely normal that developer is thinking about application domain functions at time of specification. The idea that some functions are needed to specify an operation comes first and thus they are used in specification. Then after the operation specification, the detailed consideration and specification about the functions are taken up at the end. If these features are restricted, the abstraction and description power of VDM specification will be jeopardized. However, these features prevent an one pass compiler from obtaining complete information about the system specified or request complex structure of the compiler.

There are two solutions to the problem. One is a multiple pass compiler. At least one pass for analysis and one pass for synthesis. Another way is to write VDM specification in a favorite style for better understanding.

The second approach is suggested for GAP. This comes with the desired specification style. The desired style is the style that enables GAP to obtain more information, thus leading to better result than other specification styles. The condition is that the style will not weaken the specification capability and will not greatly effect the way that developer writes the specification.

The following are the desired features for GAP:

1 Functions are defined (specified) before they are used in operation specification. This will help GAP to build compact symbol tables and reduce the time spent on symbol table searches.

2 Predicates are written in a conjunctive manner as much as possible. This may be too abstract. The spirit is that independent clauses shall be separated by **and** connective, and different clauses shall be syntactically distinct. Here are the details:

2.1 Quantified expressions are used as locally as possible.

2.2 Temporary variables used in **let, quantified, and set** expressions are used as close to its first occurrence as possible.

▷Example 5.2◁

The predicates in *BORROW_BOOK* operation[ALP91] are rewritten to be recognized that there are three clauses for each predicate according to syntax analysis. Original pre-condition and post-condition are both embedded, thus to separate them needs more analysis in deeper layers.

□

3 Negation operator **not** is used in a way to allow greater similarity between the positive clause and the negative clause. This will allow similar functions to be detected and avoid creating duplicated member functions.

▷Example 5.3◁

Instead of writing

$$user! \in UserSet,$$

a better expression for the translator will be

$$!(user \in UserSet).$$

□

It should be pointed that these desired features are not mandatory. Not doing so will not lead to any error and not necessarily lead to worse design.

5.4 System Output Files

Though we carry our discussion with programming language C++, the result obtained by GAP transformation remains programming language independent. The output is purely generic for object-oriented design.

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

- Listing file (with extension *.lst*): The original source of input is listed with line numbers. All syntax errors and semantics errors are indicated by pointing to where they occur. At the end of file, there will be a short summary of errors.
- Symbol table file (with extension *.tab*): The symbol table built by GAP for the underlying source input is displayed in this file.
- Object-orient design report file (with extension *.odr*): The ODR file is composed of three parts of result.

To explain each of the three parts of an ODR file, Example 5.3 lists some pieces of output taken from Appendix F.

▷Example 5.3◁

```
                Part I

  Class MDIR                of  MAPPING  type
 ================================================
    Domain set ::  Class  UID
    Range  set ::  Class  NMAP
                           (Inheritance information if any     comment)
```

```
The suggested member functions are
----------------------------------
   1          inv-STATE_1                      MDIR
   2          inv-STATE_2                      MDIR
   3  Prim    ADDNAME_pre_1                    MDIR
                Server Class    and            Parameter Name
              ------------------------------------------------
                UID                            INVOKER
   4  Prim    ADDNAME_pre_2                    MDIR
                Server Class    and            Parameter Name
              ------------------------------------------------
                NAME                           N
                UID                            INVOKER
   5  Prim    ADDNAME_post_1                   MDIR
                Server Class    and            Parameter Name
              ------------------------------------------------
                UID                            INVOKER
   6  Prim    ADDNAME                          MDIR
                Server Class    and            Parameter Name
              ------------------------------------------------
                UID                            INVOKER
                NAME                           N
                UID                            U
   7  Prim    DELNAME_pre_2                    MDIR
                Server Class    and            Parameter Name
              ------------------------------------------------
                NAME                           N
                UID                            INVOKER
  :
  13          INIT_post_2                      MDIR
  :
               Part II
  :
identifiers                obj
----------------------------------
102 ADDNAME                operation

   1          ADDNAME_pre_1                    MDIR
   2          ADDNAME_pre_2                    MDIR
   3          ADDNAME_post_1                   MDIR
```

```
This Op shall be implemented in Class MDIR
  :
            Part III
  :
 18 ADDNAME_pre_1                          MDIR
-------------------------------------------------
  Specification:
      UID  ISAM  DOM  MDIR
  :
 20 ADDNAME_post_1                         MDIR
-------------------------------------------------
  Specification:
      LET  OLDDIR = MDIR ( UID ) IN  MDIR '= MDIR ++[ UID ->( OLD
     DIR ++[ N -> U ])]

 21 DELNAME_pre_1                          MDIR
    Contains function (Server and Client relation)
              Server                        Message
-------------------------------------------------
Identical to   MDIR                        ::ADDNAME_pre_1
-------------------------------------------------
  Specification:
      UID  ISAM  DOM  MDIR
```

[ ]

As shown in Example 5.3, the first part shows the class information. It includes the following information for each class GAP created:

- class name;
- inheritance from base classes;
- member variables and their types; and
- suggested member functions and their parameters;
- client classes and the associated messages.

In fact there is no listing for the last information listed above, it is obtained from the suggested member functions and their parameters. The client classes are the classes of the parameters and the messages are with the same names as the underlying function.

The second part is about the operations specified in VDM operation specifications. The result for each operation displays:

- operation name;
- name of class to which the operation is transformed;
- what and where are the functions which compose the operation (what client-server pairs are needed to perform the operation), that is the primary classes and the primary functions for each clause.

The last part reveals the possible relationship between the suggested functions. The function names are generated from where they appear. In fact, the clause names are used as function names. Each section in this part contains the following items:

- clause name;
- client-server pairs; and
- clause specification.

5.5 Comparison with Other Projects

The approach discussed in this thesis is unique in itself. To our knowledge, no other system exists with which this work can be compared.

However, some systems (projects) are similar to GAP in one aspect or the other. We are going to discuss and partially compare them with GAP in this section.

5.5.1 The VDM Domain Compiler

The VDM domain compiler[SH690, SH691] is a tool for supporting the development of programs written in traditional programming languages from VDM specifications. It works with a source code database. For every VDM domain constructor the database contains a collection of different implementations with different runtime and space characteristics.

The user is forced to select a suitable implementation for every domain equation. The functional specification must still be translated by hand into the programming language in an easy line by line transformation. All data types and associated operations from the specification are directly available in the programming language via the generated library. Now the libraries are available for PASCAL and C. An interface for C++ is planned to be ready in spring 1992 without mentioning whether object-oriented paradigm is involved or not.

The approach is somewhat similar as what we discussed in the third and fourth subsections of section 4: transferring basic types and transferring user defined types, where we discussed how to find possible C++ correspondence to VDM variable types. In their approach, these correspondences are found through a library. Along with these data types, the associated operators also shall be found in the library if possible. If there is not, a hand translation is done line by line.

However, quite different from ours, domain compiler's approach is in traditional programming paradigm in Pascal and C. Both of them are strongly typed language and function-oriented. That make it impossible to provide a unique transformation for VDM type like set type or mapping type. If array is selected to implement the set type, at least an array of integer, an array of floating points, and an array of characters are needed to cope with different situations in VDM, not mentioning the differences between the possible operators upon them. So, as reported in [SH691], there are 40 different implementations for the domain constructors for sets, maps, and *etc.*in Pascal and C.

In our approach, the parameterized class (class template) makes it possible to have a unique implementation for each VDM type like sets, mapping types, and *etc.*. Their operators are unique no matter how the actual type would be. This

greatly reduces the total number of types and operators (operations) in a system. Therefore the corresponding task to identify the classes from VDM specification becomes relatively easier. That is why our focus is not concentrated on finding data types.

As to implementing functions, since domain compiler does not provide a mechanism to decompose the specification, the underlying function remains difficult to implement by hand if the specification is big. In GAP, the basic implementation unit is the clause of the pre- and post-conditions, which is apparently easier to implement.

5.5.2 The Larch Project

The Larch[GHM90, GHo91] family of languages is used to specify program interfaces in a two-tiered definitional style. Each Larch specification has components written in two languages: one that is designed for a specific programming language and another that is independent of any programming language. The former are the Larch interface languages, and the latter is the Larch Shared Language(LSL).

Modularity is the key to controlling overall system organization and components in building systems. Specifications are essential for achieving program modularity. A Larch interface specification describes the interface that a program component provides to *clients*(programs that use it). Each interface specification is written in Larch interface language. It relies on definitions from an *auxiliary specification*, written in LSL.

Each Larch interface language deals with what can be observed about the behavior of system components. It incorporates programming-specific notations for features such as side affect, exception handling, iterators and concurrency. To understand a specification written in such as Larch/C interface language(LCL), it is necessary to know both the meanings of the interface language constructors and the meanings of operators appearing in expressions. LSL specifications are used to provide a semantics for the primitive terms. Specifier are not limited to a fixed set of primitive terms, but can use LSL to define specialized vocabularies suitable for particular interface specifications.

In short, Larch's two-tiered approach attempts to minimize the gap between specification and implementation by a purely mathematical specification (carried by LSL) with an implementation via a programming language dependent Larch interface language.

Larch also provides tools supporting syntax checking, head file generating, *etc.*.

In the sense of filling the gap between specification and implementation, GAP shares some common features with Larch while they are quite different approaches.

GAP's main aim is also to fill the gap between the specification and the implementation. However, GAP attempts to make a direct link between VDM specification and object-oriented implementation. We try to provide as many implementation details as possible from specification. Information provided for classes and functions are based on implementation-minded languages and terms. As indicated in [PHG91], to implement formal specification, a programmer must not only switch notations, but also "shift gears" mentally from the abstract mathematical world of sets, sequences, and predicates, to the concrete operational world of arrays, pointers, and statements. In the words of the authors, "The absence of intermediate stages to smooth this transition entails a major semantic gap".

GAP's strategy to overcome this difficulty is to transform the entities of VDM specification into entities of implementation. Although GAP can not completely fill the gap and developer still need to understand concrete predicate clause, however, this requirement is minimized. Unlike Larch, GAP does not have an intermediate language (like LCL). Larch's approach requires developer to master three level languages — LSL used in abstraction; LCL used in interface specification and C (or other programming languages) in implementation. This requirement burdens the task of developer. The fact that to understand an interface one need to know three level of definitions certainly does not benefit the maintenance and reusability of a software module.

One of Larch tools provides the ability to form header files for developer. It helps to avoid repeating the same job. It is achieved by delete unrelated lines in the interface specification file. It shall be pointed out that It does not generate new information.

Larch is also based on function-oriented approach. LSL is a algebraic specification language. Although it has a similar structure as object in object-oriented paradigm, it does not support inheritance.

5.5.3 The Mural System

The Mural system is a totally different system compared to GAP. To compare them is perhaps not appropriate at first glance, since they aim and accomplish different things. However, in many ways they are complementary to each other in their missions towards formal software development.

The Mural system[BRi91, JJL91, MFr91] is primarily concerned with providing generic support for construction of fully formal mathematical proofs to VDM specifications. It can be used to verify the internal consistency of a specification by discharging the appropriate proof obligations. It can also be used to validate a formal specification against an informal description of the system being specified by stating and proving properties of the system which its designer believes it should exhibit.

The main component of Mural is a proof assistant. It consists of a hierarchy of *theories*, each theory containing a set of declarations of the *symbols* which can be used to build valid mathematical expressions in that theory (*e.g.* : for type assignment, + for arithmetic addition, ∀ for universal quantification, *etc.*), a set of *axioms* stating those properties of these symbols which are assumed to hold without proof(*e.g.* substitutivity of equals, that the union of two sets is a set), and a set of *rules* stating other properties of the symbols(*e.g.* associativity of set union, distribution of multiplication over addition). Properties expressed by rules may be proved using axioms and other rules. The reasoning power of Mural can be extended simply, either by adding new theories or by adding new rules to existing theories. A *tactic language* is also provided which allows the user to code and parameterize commonly used proof strategies.

In addition to the proof assistant, Mural contains a VDM support tool – VST. VST allows the user to construct specification in a subset of VDM, either by using

the built-in structure editor or by directly reading a specification file. VST also allows a specification to be designated as a refinement of another specification.

Besides, Mural provides a window-based environment with excellent user interface.

In summary, Mural is a system that supports writing correct VDM specification.

What Mural does not provide for formal software system development process, GAP tries to accomplish. First GAP is a syntactic checker (in Mural's approach, it is supported by an interface to **SpecBox**[MFr91], which is a syntactic checker). GAP will check the syntactic consistency of the input specification, point out all errors and their possible reasons. GAP's semantic checking ability is partial. At this stage Mural can interface with our tool to ensure correction of specifications. Second GAP provide an automatic transformation from VDM specification into an object-oriented design. And third, the discussions in this thesis provide an implementation strategy based on the design obtained from GAP. However GAP does not guarantee the correctness of the results, since it replies heavily on the correctness of the VDM specification. This is achieved by Mural system interface.

What we are hoping is that GAP system can be enhanced by making use of Mural system, if possible, in an integrated environment.

This will provide a systematic software development environment from formal specification to object-oriented design. Associating with other suggested enhancement to GAP (will be discussed in the next subsection), it will extend the environment service all the way to implementation.

6

Conclusion and Further Work

This section concludes the thesis and points to further development directions.

6.1 Conclusion

The automatic transformation from VDM to C++ design, which is supported by GAP system, is both feasible and practical. VDM specification provides not only formal language to support data abstraction and system modelling, but also clues to implementations. In this thesis, a systematic view on how to extract this information is presented. It also discusses an implementation strategy on how to implement software with the obtained information. GAP is implemented to realize the automatic transformation.

GAP supports object-oriented design paradigm and provides design details classes and their member functions as well as relations among classes, directly from VDM specification. The sample results shows our approach is both effective and promising for more complex situations.

GAP contributes to both formal development methods and object-oriented paradigm.

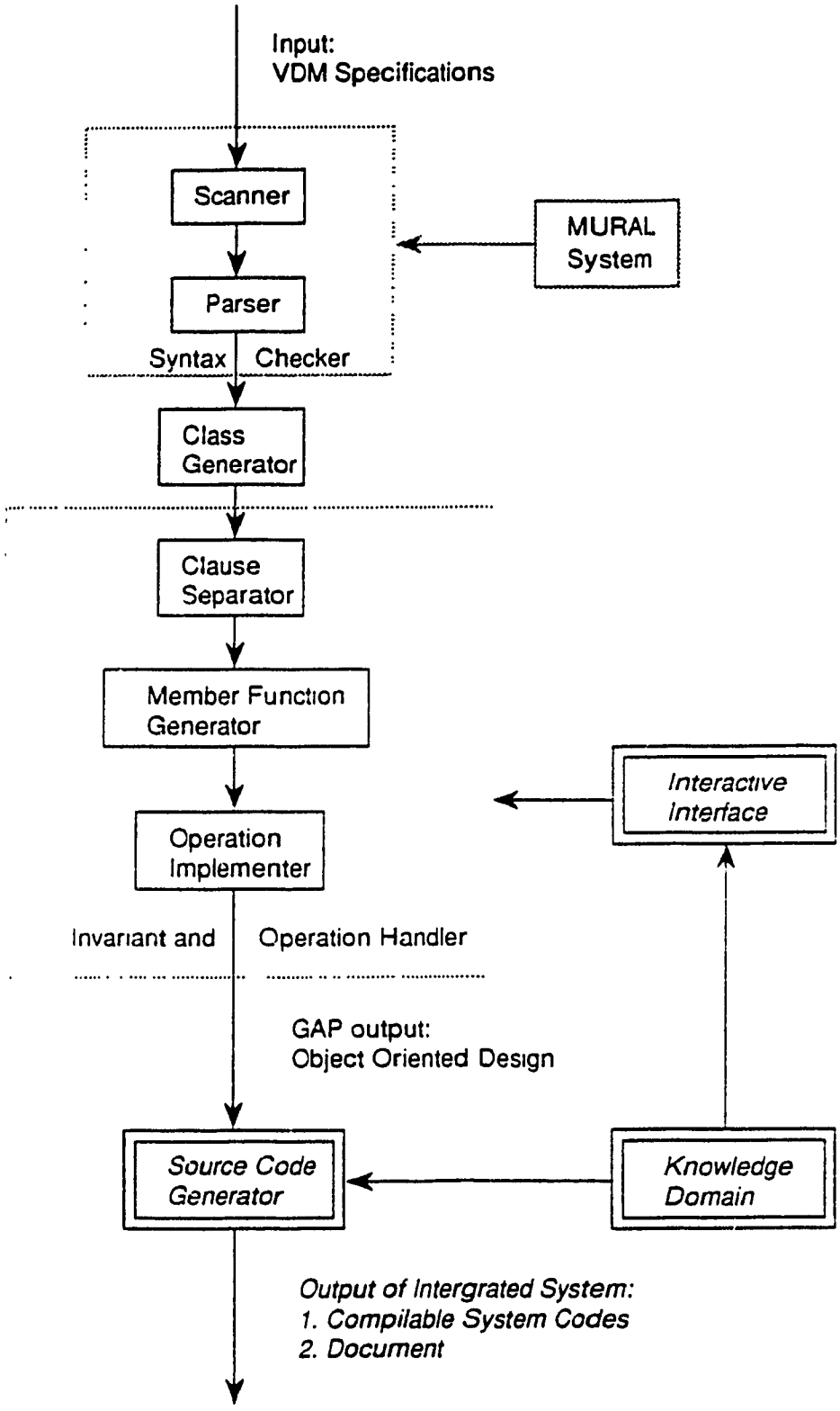

Figure 6.1 System Architecture of the Integrated Development System

GAP provides a direct development method from formal specification to implementation. It is always an annoying problem for researchers as to how links between formal specification and programming language implementation be made. GAP provides a solution to this problem. Besides, GAP's solution also supports many features that traditional technique does not provide, such as object-orientness, and reusability.

GAP provides a bridge between formal methods and object-oriented paradigm. This means that in order to develop an object-oriented software system one can make use of formal methods to ensure the correctness of the software and then the tool GAP implemented here.

6.2 Further Work

There are two major directions in which further work can be done to improve the scope of the transformer, besides interface to Mural.

First, GAP's ability can be enhanced by a multiple-pass compiler as mentioned previously.

Second, further tools can be built interactively to support human interaction in the implementation of member functions suggested and specified by GAP. The tools can be programming language specific, and generate code which is ready for compilation.

The structure of this development is described in Figure 6.1.

The core of this kind of tool will be an interactive interface and a knowledge-based code generator.

Interactive interface will enable developers to grasp the class information easily, and generate member functions as well as their specification. One of the important tasks is to rename the member functions, and all related parties shall be informed about the new names.

Knowledge-based code generator will synthesize the results obtained from GAP and the interactive interface about classes and its components (member variables and member functions), access the library of generic classes and generic functions, and finally generate the code of software system. The library is a collection of generic classes and built-in member functions as well as generic functions, which is supported by OOD technique.

Both interface and code generator will be programming language dependent. However many parts will remain the same under different object-oriented programming languages.

Another possible non-technical development is a document generator. Since the whole derivation process from VDM specifications to classes and their components are well recorded, it had already been organized to documentation. It will not be a difficult job, but an important one indeed.

Here we have more suggestions on building an interactive interface. Window-based interactive interface shall be adapted. Multiple windows at the same screen enables developer to see different aspects of the needed information to implement member functions for classes. Similar Window-based systems, such as Mural, Annalyzer[LST91](used for Anna specification and development environment for Ada), are very successful sample systems for us to model the further development of GAP.

For example, **Class Window** may show a list of classes. One of them is used as **current class** and is highlighted. Perhaps a mechanism remembers the last un-finished class as **default**.

Member Window displays the member variables and functions of current class. The types of member variables and functions, the parameters of member functions are all displayed. A **current member** is also highlighted and remembered.

Client Window lists all other classes which have at least one member function sending message to current class and requests service from current member. If current member is a variable member, this window will be empty.

A **Fellow Window** prints the possible relations among current member (function) and other fellow functions in the same class. If current member is a member variable, this window is empty.

Then a **Implementation Window** gives developer freedom to implement the function. It will display the full specification of this function and whether it is primary or not. Developer has the choice to choose from two options. First developer can actually input a string of characters that forms the function in input area, which may make use of other functions listed in Fellow Window (a function call instead of repeating the same code). Or developer use a **mouse** to select a function in Fellow Window that shall be indicated as identical in Fellow Window to replace this (current) member function. In both case, a re-check shall allow developer to re-consider the decision.

A template can be used to record all the changes for each class, which enables developer to re-install the original information before saving the results of this execution.

In case of implementing current member function, developer also has the option to rename current function — giving current member function a new name to replace the old one, not changing current member. As soon as the implementation is accepted, all the related windows shall be renewed. If the name of current member is changed, Member Window shall be renewed. In the second case of implementation, the highlight bar shall go to the selected function, it becomes current member, and old current member shall disappear from the window. More importantly, all other classes in Client Window shall be informed about the modification.

All the windows mentioned above are not necessarily opened at the same time. It is up to developer to open them with a mouse.

It should be pointed out that all the items to be displayed in these windows are already presented in the output files of GAP. The implementation of this window-based interface is not particularly difficult within the current architecture of GAP.

References

[ALP91]

Alagar, V.S. and Periyasamy, K., *A Methodology for Deriving an Object-Oriented Design from Functional Specifications*, 1991, to appear in *Software Engineering Journal.*

[AGo91]

Alencar, J.A. and Goguen, A.J., *OOZE: An Object Oriented Z Environment*, In *ECOOP'91 European Conference on Object-Oriented Programming*, edited by Pierre America, *Springer-Verlag*, pp. 180-199, 1991.

[Ala88]

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

[And88]

Andrews, D., *Report from the BSI Panel for the Standardisation of VDM (IST/5/50)*, In *VDM'88 VDM – The Way Ahead*, edited by Bloomfield, R., Marshall, L. and Jones, R., *Springer-Verlag*, pp.74-78, 1988.

[Bai89]

Bailin, S.C., *An Object-Oriented Requirements Specification Method*, *Communications of the ACM*, **32**, 5, pp.608-623, 1989.

[Bar91]

Barkakati, N., *Object-Oriented Programming in C++*, *SAMS*, 1991.

[BCu89]

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

[BJo78]

Bjørner, D. and Jones, C.B., *The Vienna Development Method: The Meta Language*, *Lecture Notes in Computer Science 61*, *Springer-Verlag*, 1978.

[BJo82]

Bjørner, D. and Jones, C.B., *Formal Specification and Software Development*, *Prentice-Hall Intl. Inc.*, 1982

[Boo86]

Booch, G., *Object-Oriented Development*, *IEEE Transactions on Software Engineering*, **SE-12**, 2, pp.211-221, 1986.

[Boo90]

Booch, G., *Software Engineering in Ada*, 3nd Edt., *Benjamin/Cummings Publishing Company*, 1990.

[BRi91]

Bicarregui, J.C. and Ritchie, B., *Reasoning about VDM Developments using The VDM Support Tool in Mural*, In *VDM'91 Formal Software Development Methods, volume 1*, *Lecture Notes in Computer Science 551*, *Springer-Verlag*, pp. 371-388, 1991.

[Bre91]

Breu, R., *Algebraic Specification Techniques in Object Oriented Programming Environments*, *Lecture Notes in Computer Science 562*, *Springer-Verlag*, 1991.

[BSI89]

BSI, *VDM Specification Language: Proto-Standard*, *IST/5/50*, 1989.

[BSI91]

BSI, *VDM Specification Language: Proto-Standard, IST/5/50*, 1991.

[Bud91]

Budd, T., *An Introduction to Object-Oriented Programming. Addison-Wesley Publishing Company*, 1991.

[Cha91]

Champeaux, D., *Object-Oriented Analysis and Top-Town Software Development*, In *ECOOP'91 European Conference on Object-Oriented Programming*, edited by Pierre America, *Springer-Verlag*, pp. 360-376, 1991.

[CHJ86]

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

[Con90]

Constantine, L.L., *Objects, Functions, and Program Extensiblity*, *Computer Language*, 7, 1, pp.34-54, 1990.

[Cox90]

Cox, B., *Object-Oriented Programming - An Evolutionary Approach*, *Readings: Addison-Wesley Publishing Company*, 1986.

[Dah90]

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

Ref[DDR89] Duke, D., Duke, R., Rose, G., and Smith, G., *Object-Z: An object-oriented extension to Z*, In *Proceedings of Formal Description Techniques (FORTE'89), North Holland*, 1989.

[Geo91]

George, C., *The RAISE Specification Language: A Tutorial*, In *VDM'91 Formal Software Development Methods, volume 2, Lecture Notes in Computer Science 551, Springer-Verlag*, pp. 238-319, 1991.

[GHM90]

Guttag, J.V., Horning, J.J. and Modet, A., *Report on the Larch Shared Language: Version 2.3, Technical Report 58, System Research Center, Digital Equipment Corporation*, 1990.

[GHo80]

Guttag, J.V. and Horning, J.J., Formal Specifications as a Design Tool, In *proceedings of the 7th Annual Symposium on Principles of Programming Languages, ACM*, pp. 251-261, 1980.

[GHo82]

Guttag, J.V. and Horning, J.J., *Some Remarks on Putting Formal Specifications to Productive Use*, In *Science of Computer programming, North-Holland*, Vol. 2, No. 1, pp.53-68, 1982.

[GHo91]

Guttag, J.V. and Horning, J.J., *Introduction to LCL, A Larch/C Interface Language, Technical Report 74, System Research Center, Digital Equipment Corporation*, 1991.

[Haß87]

Haß, M., *Development and Application of a META IV Compiler*, In *VDM-Europe Symposium 1987, VDM — A Formal Method at Work*, edited by D. Bjørner, C.B. Jones, M. Mac an Airchinnigh and E.J. Neuhold, *Springer-Verlag*, pp. 118-140, 1987.

[HEd90]

Henderson, B. and Edwards, J.M., *The Object-Oriented Systems Life-Cycle*, *Communications of the ACM*, **33**, 9, 1990.

[Ing81]

Ingalls, D., *Design Principles Behind Smalltalk*, *Byte*, August, 1981.

[JJL91]

Jones, C.B., Jones, K.D., Lindsay, P.A. and Moore, R., *mural : A Formal Development Support System*, *Springer-Verlag*, 1991.

[Jon90]

Jones, C.B., *Systematic Software Development using VDM*, 2nd version, *Prentice-Hall Intl. Inc.*, 1990.

[Jon87]

Jones, K.D., *Supporting Environments for VDM*, In *VDM-Europe Symposium 1987, VDM A Formal Method at Work*, edited by D. Bjørner, C.B. Jones, M. Mac an Airchinnigh and E.J. Neuhold, *Springer-Verlag*, pp. 110-117, 1987.

[LST91]

Luckham, D., Sankar, S. and Takahashi, S., *Two-Dimensional Pinpointing: Debugging with Formal Specifications*, *IEEE SOFTWARE*, pp.74-84, January, 1991

[Mey88]

Meyer, B., *Object-Oriented Software Construction*, *Prentice-Hall Intl. Inc.*, 1988.

[MFr91]

Moore, R. and Froome, P., *mural and SpecBox*, In *VDM'91 Formal Software Development Methods, vol 1, Lecture Notes in Computer Science 551, Springer-Verlag*, pp.672-674, 1991.

[NHW88]

Nielsen, M., Havelund, K., Wagner, K.R., and George, C., *The RAISE language, Method and Tools*, In *VDM'88 VDM - The Way Ahead*, edited by Bloomfield, R., Marshall, L. and Jones, R., *Springer-Verlag*, pp.376-405, 1988.

[PHG91]

Penny, D.A., Holt, R.C. and Godfrey, M.W., *Formal Specification in Metamorphic Programming*, In *VDM'91 Formal Software Development Methods, vol 1, Lecture Notes in Computer Science 551, Springer-Verlag*, pp. 11-30, 1991.

[PreS7]

Prehn, S., *From VDM to RAISE*, In *VDM-Europe Symposium 1987. VDM - A Formal Method at Work*, edited by D. Bjorner, C.B. Jones, M. Mac an Airchinnigh and E.J. Neuhold, *Springer-Verlag*, pp. 141-150, 1987.

[PTo91]

Prehn, S. and Toetenel, W.J., *VDM'91 Formal Software Development Methods, vols. 1 and 2, Lecture Notes in Computer Science 551, Springer-Verlag*, 1991.

[RBP91]

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

[Sen87]

Sen, D., *Objectives of the British Standardisation of a language to support the Vienna Development Method*, In *VDM-Europe Symposium 1987. VDM A Formal Method at Work*, edited by D. Bjorner, C.B. Jones, M. Mac an Airchinnigh and E.J. Neuhold, *Springer-Verlag*, pp. 321 323, 1987.

[SHó90]

Schmidt, U. and Hócher, H.M., *Programming with VDM Domains*, In *VDM' 90 VDM and Z -- Formal Methods in Software Development*, Bjøner, D., Hoare, C.A.R. and Langmaack, H. (Eds.), *Springer-Verlag*, pp. 122-134, 1990.

[SHó91]

Schmidt, U. and Hócher, H.M., *The VDM Domain Compiler A VDM Class Library Generator*, In *Proceedings of the VDM'91 Conference, Lecture Notes in Computer Science, Springer-Verlag*, pp. 675-676, 1991.

[Sny91]

Snyder, A., *Modelling the C++ Object Model*, In *ECOOP'91 European Conference on Object-Oriented Programming*, edited by Pierre America, *Springer-Verlag*, pp. 1-19, 1991.

[Str91]

Stroustrup, B., *The C++ Programming Language*, 2nd Edition, *Addison-Wesley Publishing Company*, 1991.

[War89]

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

[Win83]

Wing, J.M., *A Two Tiered Approach to Specifying Programs*, *Technical Report MIT-LCS-TR-299, MIT Laboratory for Computer Science*, 1983.

[WWW90]

Wirfs-Brock, R., Wilkerson, B. and Wiener, L., *Designing Object Oriented Software*, *Prentice-Hall*, 1990.

[WZa91]

Wing, J.M. and Zaremski, A.M., *Unintrusive Ways to Integrate Formal Specifications in Practice*, In *Proceedings of the VDM'91 Conference, Lecture Notes in Computer Science, Springer-Verlag*, pp. 545-570, 1991.

Appendix A

EBNF of A Simplified VDM-SL Syntax

1	<vdm_specification>	⇒	<state_definition> <invariant_definition> <operation_specifs>
2	<state_definition>	⇒	<type_definition> { <type_definition> }
3	<type_definition>	⇒	<record_type_def> | <set_type_def> | <map_type_def> | <rename_type_def> | <union_type_def> | <undefined_type_def>
4	<record_type_def>	⇒	<type_name> ' ::' <rec_field_list>
5	<rec_field_list>	⇒	<field_name> ' : ' <type_name_plus> { <rec_field_list> }
6	<field_name>	⇒	<var_name>
7	<type_name_plus>	⇒	<type_name> [<type_suffix>]
8	<type_name>	⇒	<basic_type> | <var_name>
9	<type_suffix>	⇒	-set | -list
10	<basic_type>	⇒	boolean | int | nat0 | nat | string
11	<var_name>	⇒	<letter> { <letter> | <single_digit> | '_' }
12	<constant>	⇒	true | false | 'nil' | <digit> | <string>
13	<letter>	⇒	a | b | ... | z
14	<digit>	⇒	<single_digit> { <single_digit> }
15	<single_digit>	⇒	1 | 2 | ... | 9 | 0
16	<set_type_def>	⇒	<type_name> ' =' <set_definition>
17	<set_definition>	⇒	'{' [<item_list>] '}' | ' <' <digit> ..<digit> ' >'
18	<item_list>	⇒	<item> { ',' <item_list> }
19	<item>	⇒	<var_name> | <constant>

20 <string> ⇒ '"' Any letter and digit combination '"'
21 <map_type_def> ⇒ <type_name> ' =' <type_name> '–>' <type_name>
22 <renamed_type_def> ⇒ <type_name> ' =' <type_name>
23 <union_type_def> ⇒ <type_name> ' =' <type_name> '|' <type_name>
{ '|' <type_name> }
24 <undefined_type_def>⇒ <type_name> ' =' <comment>
25 <comment> ⇒ /* Any string and digit combinations */

26 <invariant_definition>⇒ <inv_def_statement> { <inv_def_statement> }
27 <inv_def_statement> ⇒ inv-<type_name> ' :=' <expression>
28 <expression> ⇒ <comment> | <arithmetic_expression>
29 <arith_expression> ⇒ <simple_expression> [<relation> <simple_expression>]
30 <simple_expression> ⇒ [<sign>] <term> { <add_op> <term> }
31 <sign> ⇒ '+' | '–'
32 <relation> ⇒ ' =' | '! =' | ' <' | ' >' | ' <=' | ' >=' | ' >>' | ' <<' |
' <<=' | ' >>=' | ' →' | in
33 <term> ⇒ <factor> { <mul_op> <factor> }
34 <add_op> ⇒ '+' | '–' | '|'
35 <mul_op> ⇒ '*' | '/' | '\' | mod | '&'
36 <factor> ⇒ <constant> | <var_name> [<func_call_exp>]
<unary_op> <factor> | '(' <expression> ')' |
<let_expression> | <if_expression> |
<set_expression> | <list_definition>
<map_expression> | <quantified_exp> |
<record_expression>
37 <unary_op> ⇒ '!' | card | dom | rng | elems | len
38 <func_call_exp> ⇒ '(' <item_list> ')' | ""
39 <let_expression> ⇒ let <assign_expressions> in <expression> tel
40 <assign_expression> ⇒ <var_name> ' =' <expression>
{ ',' <assign_expression> }
41 <if_expression> ⇒ if <expression> then <expression>
+ '[' else <expression> ']' end
42 <set_expression> ⇒ <set_definition> | <set_comprehension >
43 <set_comprehension >⇒ '}' <expression> '|' <bind_var> isam <expression>

44 <bind_var> ⇒ <var_name> { ',' <bind_var> }
45 <list_definition> ⇒ '[' [<item_list>] ']'
46 <map_expression> ⇒ <map_definition> | <apply_map_to_ele>
47 <map_definition> ⇒ '[' [<map_item_list>] ']'
48 <map_item_list> ⇒ <item> '->' <item> { ',' <map_item_list> }
49 <apply_map_to_ele> ⇒ <var_name> '(' <item> ')'
50 <quantified_exp> ⇒ '(' <quantifier> <bind_var> in <expression>
'.' <expression> ')'
51 <quantifier> ⇒ forall | exists | existsone
52 <record_expression> ⇒ <mk_expression> | <select_expression>
53 <mk_expression> ⇒ mk-<type_name> '(' <item_list> ')'
54 <select_expression> ⇒ <var_name> '(' <var_name> ')'
55 <func_call_exp> ⇒ <var_name> '(' <item_list> ')'
56 <operation_specifs> ⇒ <operation_specif> { <operation_specif> }
57 <operation_specif> ⇒ <implicit_specif> | <explicit_specif>
58 <implicit_specif> ⇒ <name_part> [<ext_part>] [<pre_condition>]
<post_condition>
59 <name_part> ⇒ <func_name> '(' [<para_list>] ')' [<out_para>]
60 <func_name> ⇒ <var_name>
61 <para_list> ⇒ <var_name> ':' <type_name_plus> { ',' <para_list> }
62 <out_para> ⇒ <var_name> ':' <type_name_plus>
63 <ext_part> ⇒ ext <ext_var_list>
64 <ext_var_list> ⇒ <var_name> ':' <access> <type_name_plus>
{ <ext_var_list> }
65 <access> ⇒ wr | rd
66 <pre_condition> ⇒ pre <expression>
67 <post_condition> ⇒ post <expression>
68 <implicit_specif> ⇒ <signature> <function_body>
69 <signature> ⇒ <var_name> ':' [<func_para_list>] '->' <var_name>
70 <func_para_list> ⇒ <var_name> { '×' <var_name> }
71 <function_body> ⇒ ':=' <expression>

Appendix B

LL(1) Grammar of SVDM-SL

1	<vdm_specification>	⇒	<state_definition> <invariant_definition> <operation_specifs>
2	<state_definition>	⇒	<type_defnn> <state_definition>
3	<state_definition>	⇒	nil
4	<type_defnn>	⇒	<type_name> <type_def_body>
5	<type_def_body>	⇒	' ::' <var_name> ' : ' <type_name_plus> <more_rec_field>
6	<type_def_body>	⇒	' =' <other_type_body>
7	<other_type_body>	⇒	<set_definition>
8	<other_type_body>	⇒	<comment>
9	<other_type_body>	⇒	<map_union_rename>
10	<comment>	⇒	'/*' Any letter and digital combination '/*'
11	<map_union_rename>	⇒	<type_name> <map_to>
12	<map_to>	⇒	'->' <type_name>
13	<map_to>	⇒	<union_nil>
14	<union_nil>	⇒	'|' <type_name> <union_nil>
15	<union_nil>	⇒	nil
16	<more_rec_field>	⇒	<var_name> <rec_field_more>
17	<more_rec_field>	⇒	nil
18	<rec_field_more>	⇒	' : ' <type_name_plus> <more_rec_field>
19	<rec_field_more>	⇒	<type_def_body>
20	<type_name_plus>	⇒	<type_name> <type_suffix>

21 <type_name> ⇒ <basic_type>
22 <type_name> ⇒ <var_name>
23 <type_suffix> ⇒ -set
24 <type_suffix> ⇒ -list
25 <type_suffix> ⇒ nil
26 <basic_type> ⇒ boolean
27 <basic_type> ⇒ int
28 <basic_type> ⇒ nat0
29 <basic_type> ⇒ nat
30 <basic_type> ⇒ string
31 <var_name> ⇒ Ident
32 <constant> ⇒ true
33 <constant> ⇒ false
34 <constant> ⇒ 'nil'
35 <constant> ⇒ <Constring>
36 <constant> ⇒ <ConstInt>
37 <set_definition> ⇒ '{' <item_list> '}'
38 <set_definition> ⇒ ' <' <digit> '..' <digit> ' >'
39 <item_list> ⇒ <item_list_nonempty>
40 <item_list> ⇒ nil
41 <item_list_nonempty> ⇒ <item> <item_list_follow>
42 <item_list_follow> ⇒ ',' <item_list_nonempty>
43 <item_list_follow> ⇒ nil
44 <item> ⇒ <var_name> | <constant>
45 <invariant_definition> ⇒ <inv_def_statement> <invariant_definition>
46 <invariant_definition> ⇒ nil
47 <inv_def_statement> ⇒ inv-<type_name> ' :=' <expression>
48 <expression> ⇒ <comment>
49 <expression> ⇒ <arith_expression>
50 <arith_expression> ⇒ <simple_expression> <rel_expression>
51 <simple_expression> ⇒ ' + ' <term_or_add_term>
52 <simple_expression> ⇒ ' − ' <term_or_add_term>
53 <simple_expression> ⇒ <term_or_add_term>
54 <rel_expression> ⇒ <relation> <simple_expression>

55 <rel_expression> ⇒ nil
56 <term_or_add_term> ⇒ <term> <add_term>
57 <add_term> ⇒ <add_op> <term>
58 <add_term> ⇒ nil
59 <relation> ⇒ ' ='
60 <relation> ⇒ '! ='
61 <relation> ⇒ ' <'
62 <relation> ⇒ ' >'
63 <relation> ⇒ ' <='
64 <relation> ⇒ ' >='
65 <relation> ⇒ ' >>'
66 <relation> ⇒ ' <<'
67 <relation> ⇒ ' <<='
68 <relation> ⇒ ' >>='
69 <relation> ⇒ '->'
70 <relation> ⇒ in
71 <unary_op> ⇒ '!'
72 <unary_op> ⇒ card
73 <unary_op> ⇒ dom
74 <unary_op> ⇒ rng
75 <unary_op> ⇒ elems
76 <unary_op> ⇒ len
77 <add_op> ⇒ '+'
78 <add_op> ⇒ '-'
79 <add_op> ⇒ '|'
80 <term> ⇒ <factor> <mul_factor>
81 <mul_factor> ⇒ <mul_op> <factor> <mul_factor>
82 <mul_factor> ⇒ nil
83 <mul_op> ⇒ '*'
84 <mul_op> ⇒ '/'
85 <mul_op> ⇒ '\'
86 <mul_op> ⇒ mod
87 <mul_op> ⇒ '&'
88 <mul_op> ⇒ ' + +'

89 <factor> ⇒ <constant>
90 <factor> ⇒ <var_name> <func_call_exp>
91 <factor> ⇒ <unary_op> <factor>
92 <factor> ⇒ <let_expression>
93 <factor> ⇒ <if_expression>
94 <factor> ⇒ <set_expression>
95 <factor> ⇒ <list-definition>
96 <factor> ⇒ <map_definition>
97 <factor> ⇒ <mk_expression>
98 <factor> ⇒ <paren_expression>
99 <let_expression> ⇒ let <assign_expressions> in <expression> tel
100 <assign_expression> ⇒ <var_name> ' =' <expression> <more_assignments>
101 <more_assignments> ⇒ ',' <assignment_expression> <more_assignments>
102 <more_assignments> ⇒ nil
103 <if_expression> ⇒ if <expression> then <expression> <else_part>
104 <else_part> ⇒ else <expression> end
105 <else_part> ⇒ end
106 <set_expression> ⇒ ' <' <digit> '..' <digit> ' >'
107 <set_expression> ⇒ '{' <expression> <set_rest>
108 <set_rest> ⇒ '}'
109 <set_rest> ⇒ ',' <item_list> '}'
110 <set_rest> ⇒ '|' <bind_var> isam <expression> '}'
111 <list_definition> ⇒ ' <' <item_list> ' >'
112 <map_definition> ⇒ '[' <map_item_list> ']'
113 <map_item_list> ⇒ <map_item_list_nonempty>
114 <map_item_list> ⇒ nil
115 <map_item_nonempty> ⇒ <item> '->' <item> <map_item_follow>
116 <map_item_follow> ⇒ ',' <map_item_nonempty>
117 <map_item_follow> ⇒ nil
118 <mk_expression> ⇒ mk-<type_name> '(' <item_list> ')'
119 <paren_expression> ⇒ '(' <paren_follow> ')'
120 <paren_follow> ⇒ <quantifier> <bind_var> in <expression>
'.' <expression>
121 <paren_follow> ⇒ <expression>

122 <bind_var> ⇒ <var_name> <more_bind>
123 <more_bind> ⇒ ',' <bind_var>
124 <more_bind> ⇒ nil
125 <quantifier> ⇒ forall
126 <quantifier> ⇒ exists
127 <quantifier> ⇒ existsone
128 <func_call_exp> ⇒ '(' <item_list> ')'
129 <func_call_exp> ⇒ "'"
130 <func_call_exp> ⇒ nil
131 <operation_specifs> ⇒ <operation_specif> <operation_specifs>
132 <operation_specifs> ⇒ nil
133 <operation_specif> ⇒ <func_name> <func_follow>
134 <func_follow> ⇒ '(' <implicit_part>
135 <func_follow> ⇒ ':' <explicit_part>
136 <implicit_part> ⇒ <para_list> ')' <out_para> <ext_part>
<pre_condition> <post_condition>
137 <implicit_part> ⇒ ')' <out_para> <ext_part>
<pre_condition> <post_condition>
138 <para_list> ⇒ <para_list_nonempty>
139 <para_list> ⇒ nil
140 <para_list_nonempty> ⇒ <var_name> ':' <type_name> <para_list_follow>
141 <para_list_follow> ⇒ ',' <para_list_nonempty>
142 <para_list_follow> ⇒ nil
143 <out_para> ⇒ <var_name> ':' <type_name_plus>
144 <out_para> ⇒ nil
145 <ext_part> ⇒ ext <ext_var_list>
146 <ext_part> ⇒ nil
147 <ext_var_list> ⇒ <var_name> ':' <access> <type_name>
148 <access> ⇒ wr
149 <access> ⇒ rd
150 <pre_condition> ⇒ pre <expression>
151 <pre_condition> ⇒ nil
152 <post_condition> ⇒ post <expression>
153 <explicit_part> ⇒ <exp_func_para_list> '->' <var-name>

<function_body>

154 <exp_func_para_list> ⇒ <var_name> <para_listx>

155 <exp_func_para_list> ⇒ nil

156 <para_listx> ⇒ 'x' <var_name> <para_listx>

157 <para_listx> ⇒ nil

158 <function_body> ⇒ ':=' <expression>

Appendix C

Predict Set of the SVDM-SL

Only the predict sets of those productions that have the same prefix are listed. To make the listing more readable, some non-terminals appear in the predict sets where no confusion can be created.

2	<state_definition>	{ Ident }
3	<state_definition>	{ inv- }
5	<type_def_body>	{ ' ::' }
6	<type_def_body>	{ ' =' }
7	<other_type_body>	{ '}' }
8	<other_type_body>	{''}
9	<other_type_body>	{ Ident }
12	<map_to>	{ '->' }
13	<map_to>	{ '|', Ident, inv- }
14	<union_nil>	{ '|' }
15	<union_nil>	{ Ident, inv- }
16	<more_rec_field>	{ Ident }
17	<more_rec_field>	{ inv- }
18	<rec_field_more>	{ ' : ' }
19	<rec_field_more>	{ ' =', ' ::' }
21	<type_name>	{ boolean, int, nat, nat0, string }
22	<type_name>	{ Ident }

23	<type_suffix>	{ -set }
24	<type_suffix>	{ -list }
25	<type_suffix>	{ Ident, inv- }
26	<basic_type>	{ boolean }
27	<basic_type>	{ int }
28	<basic_type>	{ nat0 }
29	<basic_type>	{ nat }
30	<basic_type>	{ string }
32	<constant>	{ true }
33	<constant>	{ false }
34	<constant>	{ 'nil' }
35	<constant>	{ <Constring> }
36	<constant>	{ <ConstInt> }
37	<set_definition>	{ '{' }
38	<set_definition>	{ ' <' }
39	<item_list>	{ '/*', let, if, '{', ' <', '[', mk, '(', Ident, <constant> , '+', '−'}
40	<item_list>	{ ']', ' >', '}' }
42	<item_list_follow>	{ ',' }
43	<item_list_follow>	{ ']', ' >', '}' }
45	<invariant_definition>	{ inv- }
46	<invariant_definition>	{ Ident }
48	<expression>	{'/*'}
49	<expression>	{ <constant> , Ident, '!', card, dom, rng, elems, len, let, if, '{', ' <', '[', mk, '('}
51	<simple_expression>	{ ' + ' }
52	<simple_expression>	{ ' − ' }
53	<simple_expression>	{ <constant > , Ident, '!', card, dom, rng, elems, len, '(' }
54	<rel_expression>	{ <relation> }
55	<rel_expression>	{*tel*, *then*, *else*, *end*, ',', *Ident*, ']', '}', >, '.', ')'}
57	<add_term>	{'+', '−', '|'}
58	<add_term>	{<relation> , tel, then, else, end, ',', Ident, ']', '}', ' >', '.', ')'}
59	<relation>	{ ' =' }

60	<relation>	{ '! =' }
61	<relation>	{ ' <' }
62	<relation>	{ ' >' }
63	<relation>	{ ' <=' }
64	<relation>	{ ' >=' }
65	<relation>	{ ' >>' }
66	<relation>	{ ' <<' }
67	<relation>	{ ' <<=' }
68	<relation>	{ ' >>=' }
69	<relation>	{ '->' }
70	<relation>	{ in }
71	<unary_op>	{ '!' }
72	<unary_op>	{ card }
73	<unary_op>	{ dom }
74	<unary_op>	{ rng }
75	<unary_op>	{ elems }
76	<unary_op>	{ len }
77	<add_op>	{ '+' }
78	<add_op>	{ '−' }
79	<add_op>	{ '|' }
81	<mul_factor>	{'*', '/', '\', *mod*, '&'}
82	<mul_factor>	{ <add_op> , <relation> , tel, then, else, end, ',', Ident, ']', '}', ' >', '.', ')'}
83	<mul_op>	{ '*' }
84	<mul_op>	{ '/' }
85	<mul_op>	{ '\' }
86	<mul_op>	{ mod }
87	<mul_op>	{ '&' }
88	<mul_op>	{ ' + +' }
89	<factor>	{ <constant> }
90	<factor>	{ Ident }
91	<factor>	{ '!', card, dom, rng, elems, len }
92	<factor>	{ let }
93	<factor>	{ if }

94	<factor>	{'{'}
95	<factor>	{'<'}
96	<factor>	{'['}
97	<factor>	{ mk }
98	<factor>	{'('}
101	<more_assignments>	{ ',' }
102	<more_assignments>	{ in }
104	<else_part>	{ else }
105	<else_part>	{ end }
106	<set_expression>	{ ' <' }
107	<set_expression>	{ '{' }
108	<set_rest>	{ ',' }
109	<set_rest>	{ '}' }
110	<set_rest>	{ '|' }
113	<map_item_list>	{ '/*', let, if, '{', ' <', '[', mk, '(', Ident, <constant> , '+', '−'}
114	<map_item_list>	{ ']' }
116	<map_item_follow>	{ ',' }
117	<map_item_follow>	{ ']' }
120	<paren_follow>	{ forall, exists, existsone }
121	<paren_follow>	{ '/*', let, if, '{', ' <', '[', mk, '(', Ident, <constant> , '+', '−'}
123	<more_bind>	{ ',' }
124	<more_bind>	{ in }
125	<quantifier>	{ forall }
126	<quantifier>	{ exists }
127	<quantifier>	{ existsone }
128	<func_call_exp>	{ '(' }
129	<func_call_exp>	{ "" }
130	<func_call_exp>	{<mul_op> , <add_op> , <relation> , tel, else, end, ',', Ident,], }, >, '.', ')'}
131	<operation_specifs>	{ Ident }
132	<operation_specifs>	{ EOF }
134	<func_follow>	{ '(' }

135	<func_follow>	{ ':' }
136	<implicit_part>	{ Ident }
137	<implicit_part>	{ ')' }
138	<para_list>	{ Ident }
139	<para_list>	{ ')' }
141	<para_list_follow>	{ ',' }
142	<para_list_follow>	{ Ident, pre, post }
143	<out_para>	{ Ident }
144	<out_para>	{ pre, post }
145	<ext_part>	{ ext }
146	<ext_part>	{ pre, post }
148	<access>	{ wr }
149	<access>	{ rd }
150	<pre_condition>	{ pre }
151	<pre_condition>	{ post }
154	<exp_func_para_list>	{ Ident }
155	<exp_func_para_list>	{ '->' }
156	<para_listx>	{ '×' }
157	<para_listx>	{ nil }

Appendix D

Reserved Words and Special Symbols

Reserved Words in Specification	Internal Names used in GAP
card	cardsy
dom	domsy
elems	elemsy
else	elsesy
end	endsy
exists	existsy
exists1	exist1sy
ext	extsy
forall	forallsy
hd	hdsy
if	ifsy
in	insy
inv	invsy
isam	isamsy
let	letsy
post	postsy
pre	presy
rd	rdsy
rng	rngsy
tel	telsy
then	thensy
tl	tlsy
wr	wrsy

Special Symbols in Specification	Internal Names used in GAP
'+' (plus, union)	plusy
'−' (minus, negative, difference)	minusy
'*' (times, intersection)	timesy
'**' (power)	powersy
'/' (divide, strict to)	divsy
'\' (strict by)	strictbysy
'!' (not)	notsy
',' (comma)	commasy
' :' (of type)	oftypesy
' ::' (variable definitionsymbol)	vardefsy
'..' (range)	rangesy
'&' (and)	andsy
'++' (overwrite)	ovwrsy
'|' (or, concatenate)	orsy
'− >' (map to)	mapsy
' :=' (define as)	definesy
' =>' (implies)	implysy
' =' (equal to)	eqsy
'! =' (not equal to)	neqsy
' <' (less)	ltsy
' >' (greater)	gtsy
' <=' (less or equal)	leqsy
' >=' (greater or equal)	geqsy
' <<=' (subset)	subsetsy
' >>=' (supset)	supsetsy
'{' (lcurly)	lcurly
'}' (rcurly)	rcurly
'(' (lparent)	lparent
')' (rparent)	rparent
'[' (lbrack)	lbrack
']' (rbrack)	rbrack
eop (end of file)	eof
'.' (used in quantified expression)	dot
'/*' (begin symbol for comment)	bcom
'*/' (end symbol for comment)	ecom
'(*' (begin symbol for comment)	bcom
'*)' (end symbol for comment)	ecom

Appendix E

Sample Input of Source Specification

The example MAIL system takes from [CHJS6]. Several missing brackets were found by GAP and are corrected here. They are:

- In the specification of function *is_well_formed*, the last component select operator missing the parameter and the brackets;
- In the specification of the *COLLECT* operation, there is a missing bracket in the end of the specification;
- In the specification of the *DELNAME* operation, *udir* seems a function without definition anywhere through it is allowed in GAP; and
- In the specification of the *CLEAR* operation, there are two missing brackets, one after *REPLY_REQ*(m), and one after *REFNO*(i).

There are several minor modifications in the specification due to key word collisions. We renamed the variable names in collision by doubling their first letter.

```
State ::  deskof   :  Umap    (* for access to users desks *)
          direct   :  Mdir    (* for access to users directories *)
```

```
Mdir    = Uid -> Nmap
Nmap    = Name-> Uid
Umap    = Uid -> Trays

Trays ::  tin     :  Mailitem-set
          tout    :  Mailitem-set
          pend    :  Mailitem-set

Mailitem ::  to          :  Name-list
             cc          :  Name-list
             from        :  Name
             subject     :  string
             reply_req   :  boolean
             refs        :  Ref-set
             whensent    :  Datetime
             refno       :  Ref
             body        :  string

Sumrec ::  subject     :  string
           from        :  Name
           whensent    :  Datetime  (*  used for outputing *)
           refno       :  Ref        (*  mailbox summaries *)

Name     =   (*  a set of distinct names *)
Ref      =   (*  a set of distinct reference numbers *)
Uid      =   (*  a set of distinct user identification codes *)
Datetime =   (*  a suitable representation of time and date *)


inv-State  := (dom deskof & dom direct) &
(*  every user  has a desk and a directory *)

      (exists u in dom deskof.rng direct (u) <<= dom deskof) &
(* all user ids in directories must be of registered users *)

   let allmailitems = union({tin(d)+pend(d)|  d in rng deskof})
   in (forall m1,m2 in allmailitems.
         (refno(m1)=refno(m2)) ==> (m1 = m2))
(* reference numbers are unique throughout the system *)


ppost (invoker :Uid, M :mailitem)
```

```
(* takes a mail-item from invoker and puts it in out-tray *)

ext deskof:wr Umap

pre   is_well_formed(m) & invoker isam dom deskof

post  let olddesk = deskof(invoker)
      in  deskof'= deskof + [invoker ->
                    mk-Trays(tin(olddesk),
                             tout(olddesk)+ {m},
                             pend(olddesk))]


is_well_formed:Mailitem -> boolean

is_well_formed(m) :=
      to(m)!=<> &
      subject(m)!="" &  (* ""represents the null string *)
      body(m) !="" &
      whensent(m)=nil &
      refno(m) = nil


collect(invoker:uid, r:ref) m:mailitem

(* returns a mail item identified by its refernece number and
             deletes it from the in-tray *)

ext deskof:wr umap

pre   if  invoker isam dom deskof
      then (existsone m in tin(deskof(invoker)).refno(m)=r)
      else false
      endif

post  let  olddesk = deskof (invoker)
      in (existsone item in tin(olddesk).
           (refno(item)=r) &
           (m'=item) &
           (deskof'=deskof++
              [invoker->mk-Tray(tin(olddesk)-item,
                                tout(olddesk),
```

```
                                        pend(olddesk))]))


read (invoker:uid, r:ref) m:mailitem
(* returns a selected mail item from the pending tray, but does
    not       delete it *)

ext   deskof:rd umap

pre   if    invoker isam dom deskof
      then !(existsone m in PEND(deskof(invoker)).REFNO(m)=r)
      else false
      endif

post  let olddesk = deskof (invoker)
      in  !(existsone item in PEND(olddesk).
              (REFNO(item)=r) & (m'=item))


addname(invoker:Uid, n:name, u:uid)
(*    adds a new name for a given user identity code to the
   invoker' directory *)

ext   direct:wr mdir

pre   if   invoker isam dom direct
      then !(n isam dom direct(invoker)) & u isam dom direct
      else false
      endif

post  let  olddir = direct(invoker)
      in   direct' = direct ++ [invoker ->(olddir++[n->u])]


delname(invoker:Uid, n:name, u:uid)
(*    deletes a name for a particular user identity code from the
   invokers directory *)

ext   direct:wr mdir

pre   if   invoker isam dom direct
      then if n isam dom direct(invoker)
```

```
            then udir (n)=u
            else false
            endif
      endif

post  let   olddir = direct(invoker)
      in    direct'= direct++[invoker->(olddir\{n})]


llist(invoker:uid) s:sumrec-set
(* returns a non-selective summary of the contents of the
       invokers in-tray as a set of summary records *)

ext   deskof:rd umap

pre   invoker isam dom deskof

post  let  intray = tin(deskof(invoker))
      in (card s=card intray) &
         (forall j in intray.(exists k in s.k=summaryof(j)))


summaryof :  mailitem -> sumrec

summaryof(m) := mk-sumrec(subject(m),from(m),whensent(m),refno(m))


clear(invoker:uid)

ext   deskof :wr umap
      direct :rd mdir

pre   if invoker isam dom deskof
      then tout (deskof(invoker))/=
      else false
      endif

post  let directory = direct(invoker)
      in (forall u in dom deskof-{invoker}.
          (tin(deskof'(u))=tin(deskof(u))+
           {(franked(m)|  m isam tout(deskof(invoker)) &
              is_in_list(u,to(m)+cc(m),directory))} &
```

```
            pend(deskof'(u))=pend(deskof(u))+
               {(franked(m)|  m isam tout(deskof(invoker)) &
                  is_in_list(u,to(m),directory) &
                  reply_req(m))} &
         tout(deskof'(u))=tout(deskof(u)))) &
(* finish the 1st let expression *)

      deskof'(invoker) =
         let olddesk = deskof(invoker)
         in  mk-trays(tin(olddesk), ,
               pend(olddesk)-{(m|  m isam pend(olddesk) &
                  (exists i in tout(olddesk).refno(m)=refno(i)))})


franked:  mailitem->mailitem

franked(m) :=
       mk-mailitem(to(m),
                   cc(m),
                   from(m),
                   subject(m),
                   reply_req(m),
                   refs(m),
                   now(),
                   nextref(),
                   body(m))


now:->datetime

now:=
(* a system provided function which returns the date and time *)


nextref:->ref

nextref :=
(* a system wide function to supply a unique reference unmber *)


is_in_list:uid x name-list x nmap -> boolean
```

```
is_in_list(user,names,directory) :=
      user isam {directory(n) |  n isam elems names}


has_manage_priv:uid->boolean

has_manage_priv(u):=
(* a system function not defined here *)


adduser(invoker:uid, newuser:uid)

ext   deskof :  wr umap
      direct :  wr mdir

pre   has_manage_priv(invoker) &
      !(newuser isam dom deskof)

post  deskof'=deskof ++ [newuser->mk-trays(,,)] &
      direct'=direct ++ [newuser->[]]


deluser(invoker:uid, exuser:uid)

ext   deskof :  wr umap
      direct :  wr mdir

pre   has_manage_priv(invoker) &
      exuser isam dom deskof

post  deskof'=deskof\{exuser} &
      direct'=direct\{exuser}


is_logged_in:  uid->boolean

is_logged_in(u) :=
(* a system wide function, not defined here *)


init(invoker:uid) suc:boolean
```

```
ext   deskof :  wr umap
      direct :  wr mdir

pre   has_manage_priv(invoker)

post  deskof'=[] & direct'=[]
```

Appendix F

Sample Output — ODR File

PART I

```
Global Functions
----------------------------------

   1       NOW_1
   2       NOW
   3       NEXTREF_1
   4       NEXTREF
   5       IS_IN_LIST_1
   6       IS_IN_LIST
   7       ADDUSER
   8       DELUSER
   9       INIT


 Class STATE                          of  RECORD  type
============================================================
  Member Variable  DESKOF              : Class  UMAP
  Member Variable  DIRECT              : Class  MDIR
```

```
 Class MDIR                              of  MAPPING  type
=============================================================
   Domain set ::  Class  UID
   Range  set ::  Class  NMAP

The suggested member functions are
----------------------------------

   1          inv-STATE_1                          MDIR
   2          inv-STATE_2                          MDIR
   3  Prim    ADDNAME_pre_1                        MDIR
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     UID                        INVOKER
   4  Prim    ADDNAME_pre_2                        MDIR
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     NAME                       N
     UID                        INVOKER
   5  Prim    ADDNAME_post_1                       MDIR
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     UID                        INVOKER
   6  Prim    ADDNAME                              MDIR
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     UID                        INVOKER
     NAME                       N
     UID                        U
   7  Prim    DELNAME_pre_2                        MDIR
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     NAME                       N
     UID                        INVOKER
   8  Prim    DELNAME_post_1                       MDIR
=================================================
     Server Class   and          Parameter Name
```

```
-----------------------------------------------
     UID                          INVOKER
     NAME                         N
   9  Prim   DELNAME                        MDIR
===============================================
     Server Class   and           Parameter Name
-----------------------------------------------
     UID                          INVOKER
     NAME                         N
     UID                          U
  10         CLEAR_post_1                  MDIR
  11         ADDUSER_post_2                MDIR
  12  Prim   DELUSER_post_2                MDIR
===============================================
     Server Class   and           Parameter Name
-----------------------------------------------
     UID                          EXUSER
  13         INIT_post_2                   MDIR


 Class NMAP                        of  MAPPING  type
=============================================================
   Domain set ::  Class  NAME
   Range  set ::  Class  UID

The suggested member functions are
----------------------------------

   1         IS_IN_LIST_1                  NMAP


 Class UMAP                        of  MAPPING  type
=============================================================
   Domain set ::  Class  UID
   Range  set ::  Class  TRAYS

The suggested member functions are
----------------------------------

   1         inv-STATE_1                   UMAP
```

```
   2          inv-STATE_2                        UMAP
   3          inv-STATE_3                        UMAP
   4  Prim    PPOST_pre_2                        UMAP
================================================
     Server Class    and           Parameter Name
------------------------------------------------
     UID                           INVOKER
   5  Prim    PPOST_post_1                       UMAP
================================================
     Server Class    and           Parameter Name
------------------------------------------------
     UID                           INVOKER
     MAILITEM                      TIN
   6  Prim    PPOST                              UMAP
================================================
     Server Class    and           Parameter Name
------------------------------------------------
     UID                           INVOKER
     MAILITEM                      M
   7  Prim    COLLECT_pre_2                      UMAP
================================================
     Server Class    and           Parameter Name
------------------------------------------------
     MAILITEM                      TIN
     UID                           INVOKER
     REF                           REFNO
   8  Prim    COLLECT_post_1                     UMAP
================================================
     Server Class    and           Parameter Name
------------------------------------------------
     UID                           INVOKER
     MAILITEM                      TIN
     REF                           REFNO
   9  Prim    COLLECT                            UMAP
================================================
     Server Class    and           Parameter Name
------------------------------------------------
     UID                           INVOKER
     REF                           R
  10  Prim    READ_pre_2                         UMAP
================================================
     Server Class    and           Parameter Name
```

```
-----------------------------------------------
    MAILITEM                    PEND
    UID                         INVOKER
    REF                         REFNO
 11  Prim   READ_post_1                  UMAP
===============================================
    Server Class   and          Parameter Name
-----------------------------------------------
    UID                         INVOKER
    MAILITEM                    PEND
    REF                         REFNO
 12  Prim   READ                          UMAP
===============================================
    Server Class   and          Parameter Name
-----------------------------------------------
    UID                         INVOKER
    REF                         R
 13  Prim   LLIST_post_1                 UMAP
===============================================
    Server Class   and          Parameter Name
-----------------------------------------------
    MAILITEM                    TIN
    UID                         INVOKER
    SUMREC                      S
 14  Prim   LLIST_post_2                 UMAP
===============================================
    Server Class   and          Parameter Name
-----------------------------------------------
    MAILITEM                    TIN
    UID                         INVOKER
    SUMREC                      S
 15  Prim   LLIST                        UMAP
===============================================
    Server Class   and          Parameter Name
-----------------------------------------------
    UID                         INVOKER
 16  Prim   CLEAR_pre_2                  UMAP
===============================================
    Server Class   and          Parameter Name
-----------------------------------------------
    MAILITEM                    TOUT
    UID                         INVOKER
```

```
  17  Prim   CLEAR_post_1                    UMAP
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     MDIR                        DIRECT
     UID                         INVOKER
     MAILITEM                    TIN
     NAME                        TO
     BOOLEAN                     REPLY_REQ
  18  Prim   CLEAR_post_2                    UMAP
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     UID                         INVOKER
     MAILITEM                    TIN
     REF                         REFNO
  19  Prim   CLEAR                           UMAP
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     UID                         INVOKER
     MDIR                        DIRECT
  20  Prim   ADDUSER_pre_2                   UMAP
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     UID                         NEWUSER
  21         ADDUSER_post_1                  UMAP
  22  Prim   DELUSER_post_1                  UMAP
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     UID                         EXUSER
  23         INIT_post_1                     UMAP


 Class TRAYS                      of  RECORD  type
=============================================================
  Member Variable  TIN              : POWER SET Of Class  MAILITEM
  Member Variable  TOUT             : POWER SET Of Class  MAILITEM
  Member Variable  PEND             : POWER SET Of Class  MAILITEM
```

```
 Class MAILITEM                          of  RECORD  type
=================================================================
  Member Variable  TO                    : LIST Of Class  NAME
  Member Variable  CC                    : LIST Of Class  NAME
  Member Variable  FROM                  : Class  NAME
  Member Variable  SUBJECT               : Class  STRING
  Member Variable  REPLY_REQ             : Class  BOOLEAN
  Member Variable  REFS                  : POWER SET Of Class   REF
  Member Variable  WHENSENT              : Class  DATETIME
  Member Variable  REFNO                 : Class  REF
  Member Variable  BODY                  : Class  STRING

The suggested member functions are
----------------------------------

   1          inv-STATE_3                    MAILITEM
   2          PPOST_pre_1                    MAILITEM
   3          PPOST_post_1                   MAILITEM
   4  Prim    IS_WELL_FORMED_1               MAILITEM
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     NAME                        TO
   5  Prim    IS_WELL_FORMED_2               MAILITEM
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     STRING                      SUBJECT
   6  Prim    IS_WELL_FORMED_3               MAILITEM
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     STRING                      BODY
   7  Prim    IS_WELL_FORMED_4               MAILITEM
=================================================
     Server Class   and          Parameter Name
-------------------------------------------------
     DATETIME                    WHENSENT
   8  Prim    IS_WELL_FORMED_5               MAILITEM
```

```
================================================
     Server Class    and          Parameter Name
------------------------------------------------
     REF                          REFNO
   9  Prim    IS_WELL_FORMED                MAILITEM
  10         COLLECT_pre_2                  MAILITEM
  11         COLLECT_post_1                 MAILITEM
  12         READ_pre_2                     MAILITEM
  13         READ_post_1                    MAILITEM
  14         LLIST_post_1                   MAILITEM
  15         LLIST_post_2                   MAILITEM
  16  Prim    SUMMARYOF_1                   MAILITEM
================================================
     Server Class    and          Parameter Name
------------------------------------------------
     STRING                       SUBJECT
     NAME                         FROM
     DATETIME                     WHENSENT
     REF                          REFNO
  17  Prim    SUMMARYOF                     MAILITEM
  18         CLEAR_pre_2                    MAILITEM
  19         CLEAR_post_1                   MAILITEM
  20         CLEAR_post_2                   MAILITEM
  21  Prim    FRANKED_1                     MAILITEM
================================================
     Server Class    and          Parameter Name
------------------------------------------------
     NAME                         TO
     STRING                       SUBJECT
     BOOLEAN                      REPLY_REQ
     REF                          REFS
  22  Prim    FRANKED                       MAILITEM


 Class SUMREC                      of  RECORD  type
============================================================
  Member Variable  SUBJECT          : Class  STRING
  Member Variable  FROM             : Class  NAME
  Member Variable  WHENSENT         : Class  DATETIME
  Member Variable  REFNO            : Class  REF
```

```
The suggested member functions are
----------------------------------

   1          LLIST_post_1                    SUMREC
   2          LLIST_post_2                    SUMREC


 Class NAME                            of  UNDEFINED  type
============================================================

The suggested member functions are
----------------------------------

   1          IS_WELL_FORMED_1               NAME
   2          ADDNAME_pre_2                  NAME
   3          DELNAME_pre_2                  NAME
   4          DELNAME_post_1                 NAME
   5          SUMMARYOF_1                    NAME
   6          CLEAR_post_1                   NAME
   7          FRANKED_1                      NAME
   8          IS_IN_LIST_1                   NAME


 Class REF                             of  UNDEFINED  type
============================================================

The suggested member functions are
----------------------------------

   1          inv-STATE_3                    REF
   2          IS_WELL_FORMED_5               REF
   3          COLLECT_pre_2                  REF
   4          COLLECT_post_1                 REF
   5          READ_pre_2                     REF
   6          READ_post_1                    REF
   7          SUMMARYOF_1                    REF
   8          CLEAR_post_2                   REF
   9          FRANKED_1                      REF
```

```
 Class UID                                of  UNDEFINED  type
=============================================================

The suggested member functions are
----------------------------------

   1          PPOST_pre_2                        UID
   2          PPOST_post_1                       UID
   3          COLLECT_pre_2                      UID
   4          COLLECT_post_1                     UID
   5          READ_pre_2                         UID
   6          READ_post_1                        UID
   7          ADDNAME_pre_1                      UID
   8          ADDNAME_pre_2                      UID
   9          ADDNAME_post_1                     UID
  10          DELNAME_pre_2                      UID
  11          DELNAME_post_1                     UID
  12          LLIST_post_1                       UID
  13          LLIST_post_2                       UID
  14          CLEAR_pre_2                        UID
  15          CLEAR_post_1                       UID
  16          CLEAR_post_2                       UID
  17          IS_IN_LIST_1                      UID
  18          HAS_MANAGE_PRIV_1                 UID
  19  Prim    HAS_MANAGE_PRIV                    UID
  20          ADDUSER_pre_2                      UID
  21          DELUSER_post_1                     UID
  22          DELUSER_post_2                     UID
  23          IS_LOGGED_IN_1                    UID
  24  Prim    IS_LOGGED_IN                       UID


 Class DATETIME                           of  UNDEFINED  type
=============================================================

The suggested member functions are
----------------------------------

   1          IS_WELL_FORMED_4                 DATETIME
   2          SUMMARYOF_1                       DATETIME
```

PART II

```
identifiers                obj
-----------------------------------
 53 STATE                  invariant

   1          inv-STATE_1                      STATE
   2          inv-STATE_2                      STATE
   3          inv-STATE_3                      STATE
This Op shall be implemented in Class


identifiers                obj
-----------------------------------
 63 PPOST                  operation

   1          PPOST_pre_1                      MAILITEM
   2          PPOST_pre_2                      UMAP
   3          PPOST_post_1                     UMAP
This Op shall be implemented in Class UMAP


identifiers                obj
-----------------------------------
 72 IS_WELL_FORMED         operation

   1          IS_WELL_FORMED_1                 MAILITEM
   2          IS_WELL_FORMED_2                 MAILITEM
   3          IS_WELL_FORMED_3                 MAILITEM
   4          IS_WELL_FORMED_4                 MAILITEM
   5          IS_WELL_FORMED_5                 MAILITEM
This Op shall be implemented in Class MAILITEM


identifiers                obj
-----------------------------------
 80 COLLECT                operation
```

```
   1          COLLECT_pre_1                    UMAP
   2          COLLECT_pre_2                    UMAP
   3          COLLECT_post_1                   UMAP
This Op shall be implemented in Class UMAP


identifiers                obj
-----------------------------------
 92 READ                   operation

   1          READ_pre_1                       UMAP
   2          READ_pre_2                       UMAP
   3          READ_post_1                      UMAP
This Op shall be implemented in Class UMAP


identifiers                obj
-----------------------------------
102 ADDNAME                operation

   1          ADDNAME_pre_1                    MDIR
   2          ADDNAME_pre_2                    MDIR
   3          ADDNAME_post_1                   MDIR
This Op shall be implemented in Class MDIR


identifiers                obj
-----------------------------------
108 DELNAME                operation

   1          DELNAME_pre_1                    MDIR
   2          DELNAME_pre_2                    MDIR
   3          DELNAME_post_1                   MDIR
This Op shall be implemented in Class MDIR


identifiers                obj
-----------------------------------
115 LLIST                  operation

   1          LLIST_pre_1                      UMAP
   2          LLIST_post_1                     UMAP
```

```
   3         LLIST_post_2                    UMAP
This Op shall be implemented in Class UMAP


identifiers              obj
-----------------------------------
124 SUMMARYOF            operation

   1         SUMMARYOF_1                     MAILITEM
This Op shall be implemented in Class MAILITEM


identifiers              obj
-----------------------------------
131 CLEAR                operation

   1         CLEAR_pre_1                     UMAP
   2         CLEAR_pre_2                     UMAP
   3         CLEAR_post_1                    UMAP
   4         CLEAR_post_2                    UMAP
This Op shall be implemented in Class UMAP


identifiers              obj
-----------------------------------
149 FRANKED              operation

   1         FRANKED_1                       MAILITEM
This Op shall be implemented in Class MAILITEM


identifiers              obj
-----------------------------------
161 NOW                  operation

   1         NOW_1
This Op shall be implemented as Global Function


identifiers              obj
-----------------------------------
163 NEXTREF              operation
```

```
   1          NEXTREF_1
This Op shall be implemented as Global Function


identifiers                obj
-----------------------------------
165 IS_IN_LIST             operation

   1          IS_IN_LIST_1
This Op shall be implemented as Global Function


identifiers                obj
-----------------------------------
171 HAS_MANAGE_PRIV        operation

   1          HAS_MANAGE_PRIV_1                UID
This Op shall be implemented in Class UID


identifiers                obj
-----------------------------------
174 ADDUSER                operation

   1          ADDUSER_pre_1                    UID
   2          ADDUSER_pre_2                    UMAP
   3          ADDUSER_post_1                   UMAP
   4          ADDUSER_post_2                   MDIR
This Op shall be implemented as Global Function


identifiers                obj
-----------------------------------
180 DELUSER                operation

   1          DELUSER_pre_1                    UID
   2          DELUSER_pre_2                    UMAP
   3          DELUSER_post_1                   UMAP
   4          DELUSER_post_2                   MDIR
This Op shall be implemented as Global Function
```

```
identifiers                obj
-----------------------------------
186 IS_LOGGED_IN           operation

   1          IS_LOGGED_IN_1                  UID
This Op shall be implemented in Class UID


identifiers                obj
-----------------------------------
189 INIT                   operation

   1          INIT_pre_1                      UID
   2          INIT_post_1                     UMAP
   3          INIT_post_2                     MDIR
This Op shall be implemented as Global Function
```

PART III

Total Clause Printout

```
    Clause Name                    Be Creating for Class
-------------------------------------------------------
  1 inv-STATE_1
-------------------------------------------------
  Specification:

     ( DOM  UMAP & DOM  MDIR )


  2 inv-STATE_2
-------------------------------------------------
  Specification:

     ( EXISTS  U  IN  DOM  DESKOF . RNG  DIRECT ( U )<<= DOM  DES
     KOF )
```

```
  3 inv-STATE_3
-------------------------------------------------
  Specification:

      LET  ALLMAILITEMS = TIN ( RNG  DESKOF )+ PENG ( RNG  DESKOF
      ) IN ( FORALL  M1 , M2  IN  ALLMAILITEMS .( REFNO ( M1 )= R
     EFNO ( M2 ))==>( M1 = M2 ))


  4 PPOST_pre_1                                  MAILITEM
-------------------------------------------------
  Specification:

      IS_WELL_FORMED ( MAILITEM )


  5 PPOST_pre_2                                  UMAP
-------------------------------------------------
  Specification:

      UID  ISAM  DOM  UMAP


  6 PPOST_post_1                                 UMAP
---------  ---------------------------------------
  Specification:

      LET  OLDDESK = UMAP ( UID ) IN  UMAP '= UMAP +[ UID -> MK -
      TRAYS ( TIN ( OLDDESK ), TOUT ( OLDDESK )+{ M }, PEND ( OLD
     DESK ))]


  7 IS_WELL_FORMED_1                             MAILITEM
-------------------------------------------------
  Specification:

      TO ( MAILITEM )!=<>


  8 IS_WELL_FORMED_2                             MAILITEM
-------------------------------------------------
  Specification:
```

```
      SUBJECT ( MAILITEM )!= ""


  9 IS_WELL_FORMED_3                          MAILITEM
-------------------------------------------------------
  Specification:

      BODY ( MAILITEM )!= ""


 10 IS_WELL_FORMED_4                          MAILITEM
-------------------------------------------------------
  Specification:

      WHENSENT ( MAILITEM )= NIL


 11 IS_WELL_FORMED_5                          MAILITEM
-------------------------------------------------------
  Specification:

      REFNO ( MAILITEM )= NIL


 12 COLLECT_pre_1                             UMAP
    Contains function (Server and Client relation)
              Server                           Message
-------------------------------------------------------
Identical to   UMAP                          ::PPOST_pre_2
-------------------------------------------------------
  Specification:

      UID  ISAM  DOM  UMAP


 13 COLLECT_pre_2                             UMAP
-------------------------------------------------------
  Specification:

     ( EXISTSONE  M  IN  TIN ( UMAP ( UID )). REFNO ( M )= R )
```

```
 14 COLLECT_post_1                          UMAP
-------------------------------------------------
  Specification:

      LET  OLDDESK = UMAP ( UID ) IN ( EXISTSONE  ITEM  IN  TIN (
      OLDDESK ).( REFNO ( ITEM )= R )&( M '= ITEM )&( UMAP '= UMA
     P ++[ UID -> MK - TRAY ( TIN ( OLDDESK )- ITEM , TOUT ( OLDD
     ESK ), PEND ( OLDDESK ))]))


 15 READ_pre_1                              UMAP
    Contains function (Server and Client relation)
              Server                         Message
-------------------------------------------------
Identical to   UMAP                        ::PPOST_pre_2
-------------------------------------------------
  Specification:

      UID  ISAM  DOM  UMAP


 16 READ_pre_2                              UMAP
-------------------------------------------------
  Specification:

     !( EXISTSONE  M  IN  PEND ( UMAP ( UID )). REFNO ( M )= R )


 17 READ_post_1                             UMAP
-------------------------------------------------
  Specification:

      LET  OLDDESK = UMAP ( UID ) IN !( EXISTSONE  ITEM  IN  PEND
      ( OLDDESK ).( REFNO ( ITEM )= R )&( M '= ITEM ))


 18 ADDNAME_pre_1                           MDIR
-------------------------------------------------
  Specification:

      UID  ISAM  DOM  MDIR
```

```
 19 ADDNAME_pre_2                          MDIR
-------------------------------------------------
  Specification:

     !( NAME  ISAM  DOM  MDIR ( UID ))& U  ISAM  DOM  MDIR


 20 ADDNAME_post_1                         MDIR
-------------------------------------------------
  Specification:

      LET  OLDDIR = MDIR ( UID ) IN  MDIR '= MDIR ++[ UID ->( OLD
     DIR ++[ N -> U ])]


 21 DELNAME_pre_1                          MDIR
    Contains function (Server and Client relation)
              Server                        Message
-------------------------------------------------
Identical to   MDIR                       ::ADDNAME_pre_1
-------------------------------------------------
  Specification:

      UID  ISAM  DOM  MDIR


 22 DELNAME_pre_2                          MDIR
-------------------------------------------------
  Specification:

      IF  NAME  ISAM  DOM  MDIR ( UID ) THEN  UDIR ( NAME )= U  E
     LSE  FALSE  ENDIF


 23 DELNAME_post_1                         MDIR
-------------------------------------------------
  Specification:

      LET  OLDDIR = MDIR ( UID ) IN  MDIR '= MDIR ++[ UID ->( OLD
     DIR \{ NAME })]
```

```
 24 LLIST_pre_1                              UMAP
    Contains function (Server and Client relation)
              Server                          Message
-------------------------------------------------
Identical to   UMAP                         ::PPOST_pre_2
-------------------------------------------------
  Specification:

      UID  ISAM  DOM  UMAP


 25 LLIST_post_1                             UMAP
-------------------------------------------------
  Specification:

      LET  INTRAY = TIN ( UMAP ( UID )) IN ( CARD  S = CARD  INTR
     AY )


 26 LLIST_post_2                             UMAP
-------------------------------------------------
  Specification:

     ( FORALL  J  IN  INTRAY .( EXISTS  K  IN  S  . K = SUMMARYOF
     ( J )))


 27 SUMMARYOF_1                              MAILITEM
-------------------------------------------------
  Specification:

      MK - SUMREC ( SUBJECT ( MAILITEM ), FROM ( MAILITEM ), WHEN
     SENT ( MAILITEM ), REFNO ( M ))


 28 CLEAR_pre_1                              UMAP
    Contains function (Server and Client relation)
              Server                          Message
-------------------------------------------------
Identical to   UMAP                         ::PPOST_pre_2
```

```
-------------------------------------------------
  Specification:

      UID  ISAM  DOM  UMAP


 29 CLEAR_pre_2                          UMAP
-------------------------------------------------
  Specification:

      TOUT ( UMAP ( UID ))/={}


 30 CLEAR_post_1                         UMAP
-------------------------------------------------
  Specification:

      LET  DIRECTORY = MDIR ( UID ) IN ( FORALL  U  IN  DOM  UMAP
      -{ UID }.( TIN ( UMAP '( U ))= TIN ( UMAP ( U ))+{( FRANKED
      ( M )| M  ISAM  TOUT ( UMAP ( UID ))& IS_IN_LIST ( U , TO (
      M )+ CC ( M ), DIRECTORY ))}& PEND ( UMAP '( U ))= PEND ( U
     MAP ( U ))+{( FRANKED ( M )| M  ISAM  TOUT ( UMAP ( UID ))&
     IS_IN_LIST ( U , TO ( M ), DIRECTORY )& REPLY_REQ ( M ))}& T
     OUT ( UMAP '( U ))= TOUT ( UMAP ( U ))))


 31 CLEAR_pos _2                         UMAP
-------------------------------------------------
  Specification:

      UMAP '( UID )= LET  OLDDESK = UMAP ( UID ) IN  MK - TRAYS (
      TIN ( OLDDESK ),{}, PEND ( OLDDESK )-{( M | M  ISAM  PEND (
      OLDDESK )&( EXISTS  I  IN  TOUT ( OLDDESK ). REFNO ( M )= R
     EFNO ( I )))})


 32 FRANKED_1                            MAILITEM
    Contains function (Server and Client relation)
              Server                       Message
-------------------------------------------------
                                          ::NOW_1
                                          ::NEXTREF_1
```

```
-------------------------------------------------
  Specification:

      MK - MAILITEM ( TO ( MAILITEM ), CC ( MAILITEM ), FROM ( MA
     ILITEM ), SUBJECT ( MAILITEM ), REPLY_REQ ( MAILITEM ), REFS
      ( M ), NOW (), NEXTREF (), BODY ( M ))


 33 NOW_1
-------------------------------------------------
  Specification:

      NOW ()


 34 NEXTREF_1
-------------------------------------------------
  Specification:

      NEXTREF ()


 35 IS_IN_LIST_1
-------------------------------------------------
  Specification:

      UID  ISAM { NMAP ( N )| N  ISAM  ELEMS  NAME }


 36 HAS_MANAGE_PRIV_1                    UID
-------------------------------------------------
  Specification:

      HAS_MANAGE_PRIV ( UID )


 37 ADDUSER_pre_1                        UID
    Contains function (Server and Client relation)
              Server                        Message
-------------------------------------------------
Identical to   UID                      ::HAS_MANAGE_PRIV_1
-------------------------------------------------
```

```
 Specification:

     HAS_MANAGE_PRIV ( UID )


 38 ADDUSER_pre_2                          UMAP
    Contains function (Server and Client relation)
               Server                        Message
-------------------------------------------------
               UMAP                        ::PPOST_pre_2
               UMAP                        ::COLLECT_pre_1
               UMAP                        ::READ_pre_1
               UMAP                        ::LLIST_pre_1
               UMAP                        ::CLEAR_pre_1
-------------------------------------------------
 Specification:

     !( UID  ISAM  DOM  UMAP )


 39 ADDUSER_post_1                         UMAP
-------------------------------------------------
 Specification:

     UMAP '= UMAP ++[ NEWUSER -> MK - TRAYS ({},{},{})]


 40 ADDUSER_post_2                         MDIR
-------------------------------------------------
 Specification:

     MDIR '= MDIR ++[ NEWUSER ->[]]


 41 DELUSER_pre_1                          UID
    Contains function (Server and Client relation)
               Server                        Message
-------------------------------------------------
Identical to   UID                        ::HAS_MANAGE_PRIV_1
-------------------------------------------------
 Specification:
```

```
      HAS_MANAGE_PRIV ( UID )


 42 DELUSER_pre_2                          UMAP
    Contains function (Server and Client relation)
              Server                        Message
-------------------------------------------------
Identical to   UMAP                      ::PPOST_pre_2
-------------------------------------------------
  Specification:

      UID  ISAM  DOM  UMAP


 43 DELUSER_post_1                         UMAP
-------------------------------------------------
  Specification:

      UMAP '= UMAP \{ UID }


 44 DELUSER_post_2                         MDIR
-------------------------------------------------
  Specification:

      MDIR '= MDIR \{ UID }


 45 IS_LOGGED_IN_1                         UID
-------------------------------------------------
  Specification:

      IS_LOGGED_IN ( UID )


 46 INIT_pre_1                             UID
    Contains function (Server and Client relation)
              Server                        Message
-------------------------------------------------
Identical to   UID                       ::HAS_MANAGE_PRIV_1
-------------------------------------------------
  Specification:
```

```
     HAS_MANAGE_PRIV ( UID )

 47 INIT_post_1                                UMAP
-------------------------------------------------------
  Specification:

     UMAP '=[]

 48 INIT_post_2                                MDIR
-------------------------------------------------------
  Specification:

     MDIR '=[]
```