On the Transformation of a Semi-formal Software Description to a VDM Specification

Juliette D'Almeida

A Thesis

in

The Department

of

Computer Science

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

November 1992

ISBN 0-315-84648-8

ABSTRACT

On the Transformation of a Semi-formal Software
Description to a VDM Specification

Juliette D'Almeida

Requirements Analysis is one of the most important but least supported phases of the software development process. Descriptions of the requirements of a software system written in an unconstrained natural language are considered to be informal. Informal descriptions are known to have the potential to contain ambiguities, partial descriptions, inconsistencies, incompleteness and poor ordering of requirements. For that reason, formal methods are introduced in the early stages of the software development process to force a thorough analysis of the system requirements. A formal specification is concise, precise and has a mathematical basis for proofs of consistency and correctness. However, the strict semantics of formal specifications are not well suited for communication with most users. At one end, informal languages are required for naive users; at the other end, designers need the rigour of formal languages. This thesis presents an approach to bridge the gap between these two ends.

The chosen approach is to have a semi-formal description mechanism which contains characteristics of both informal and formal methods, and automatically derive a formal specification from it. The target formal language chosen is VDM. In this thesis, we present a **M**odified **E**ntity-**R**elationship model and the **K**eyword-based **F**ormatted **D**escription techniques as a means for semi-formal specifications. The MER model is a simple modification of the well-known Entity-Relationship approach to data modeling. The MER model describes the different system entities and the relationships among them at various levels of abstraction. The KFDs describe functional requirements of the intended software system as textual descriptions using keywords. *Predefined entity types* can be used in building the MER model. They

represent known facts about the problem domain. The notion of *High-order keywords* presented in this thesis can be viewed as macros manipulating predefined entities. They facilitate the development of KFDs. Both predefined entities and high-order keywords are stored information about the problem domain. A *description preprocessor*, which embodies knowledge about the MER and KFD, ensures that the input description is consistent and syntactically correct. It uses common knowledge to fill incompleteness in the input, whenever possible, and produces an intermediary form which will be the input to the transformation step. The *transformation system* is composed of a set of rules which transform the encoded MER and KFDs into a formal VDM specification. Then, existing automatic proof checking systems can be used to verify the generated VDM specifications.

Dédicace

Je dédie cette thèse

à mes parents

José

et

Eduarda

Remerciements

Je voudrais remercier chaleureusement mon superviseur de thèse, Dr. Radhakrishnan qui a collaboré avec moi tout au long de mes études. Il a été d'un soutien incomparable autant du point de vue professionnel que moral. Je lui suis reconnaissante pour m'avoir fait confiance et m'avoir donné la chance de choisir un sujet de thèse selon mes préférences. Je le remercie pour sa patience infinie durant ces longues heures qui ont permis de produire cette thèse. Sa générosité et son dévouement font de lui une personne exceptionnelle et ce fût un plaisir que de travailler avec lui. Plus particulièrement, je désirerais lui exprimer ma profonde gratitude pour son réconfort et son amitié toujours présente dans les moments difficiles. C'est un homme de grande sagesse de qui on apprend beaucoup plus que simplement de la science.

Egalement, je remercie profondément Ramesh Achuthan pour son aide très appréciée tout au long du développement de cette thèse ainsi que le Dr. Alagar. Leurs idées et leurs connaissances ont contribué à rendre cette thèse ce qu'elle est maintenant. Je leur suis reconnaissante pour le précieux temps qu'ils m'ont dédié.

J'aimerais exprimer un remerciement tout spécial à Christy Yep pour sa présence irremplaçable durant ces quatres dernières années. Sa patience, sa bonté et sa joie de vivre ont fait de cette expérience un moment inoubliable. Egalement, j'adresse un merci particulier à Alain Jules Sarraf pour son amitié spéciale et son soutien moral. Il a fait de notre environnement de travail, un endroit agréable à vivre.

En dernier lieu mais non le moindre, je remercie sincèrement ma famille qui m'a soutenue et encouragée durant toutes mes études et, sans qui, il m'aurait été impossible de compléter cette thèse. Plus particulièrement, je dis un gros merci à mon père, ma mère, Philippe et Grace, Claudette et Jean-Luc. A Catherine, je lui dois ma profonde estime pour sa chaleur et son attention. Encore une fois, je les remercie pour leur patience, leur dévouement et leur amour.

Contents

List of Figures

Chapter 1

Introduction

The evolution of computing machines haven't ceased to increase since 1940's. While the focus was on hardware in the past, there has been a gradual trend in concentrating on problem solving. The price of hardware started to decrease, machines got more powerful, higher level languages were available, so larger software systems were developed. Around the 1960s, the need to focus on software development became critical. Basically, software systems were getting more and more important due to their increasing size, the greater amount of effort required from people and, consequently the higher cost. This is called the "software crisis". Hence, there was a need for new techniques and methodologies to develop software which resulted in the so called "Software Engineering". The goal of Software Engineering is to produce high quality software at low cost. The basic phases in the software development process are: Requirements Analysis, Design, Coding and Testing. In the requirements analysis phase, we identify *what* the system should do. The design phase emphasizes on *how* to satisfy the system requirements. The coding phase translates the design into code in a chosen programming language. Finally, testing detects errors in the software introduced in the coding or the previous stages. This thesis concentrates only on the Requirements Analysis phase.

1.1 Requirements Analysis

Requirements Analysis is one of the most important and least supported phases of the software development process. It has been reported that over 50% of software malfunctions have their origin in the requirements determination. although the errors are detected only at later stages [Loucopoulos'89. Sommerville'89]. This late detection results in a considerable effort from the part of the debugging. testing and maintenance teams and dramatically increases the cost of the system being developed. For that reason, researchers in the field of software engineering are paying more attention to this phenomena.

There are three majors activities in the Requirements Analysis phase: eli citation. formalization and validation. In the first step, the analyst must understand the problem and its context and obtain an informal description of the system requirements. The goals of the systems are clarified. i.e. one clearly identifies the important data entities. the purpose of the different actions to be performed and the interactions of the system with the environment. This identification requires interaction with clients and end users. as well as studying the existing procedures. In the next stage. the requirements are structured and formalized to obtain a formal requirements specification. Hence. some specification language has to be selected to write the requirements document. The formal document should be free from ambiguities. contradictions. redundancies and incorrectness. Lastly. the validation stage ensures that the requirements specified in the document actually correspond to the actual needs and that all requirements have been specified.

The end product of the eli citation stage is an informal requirements document. The next stage formalizes this document into a formal requirements specification. The main difficulty in the Requirements Analysis phase lies in describing the needs of the end users in a structured and formalized way. The basic reason for this difficulty is that the different people involved in the process of Requirements Analysis have different needs on specification languages. There is a communication gap between the users and the designers/constructors of the system. The users are familiar with a natural language and probably visual descriptions such as graphs. On the other hand, the designers need a language which is precise, unambiguous and that supports completeness, correctness and consistency in specifications. Hence, the real problem is in bridging the gap between informal specification (obtained from the users) and the formal specification (for the system designers). This thesis concentrates on the improvement in bridging the gap between the users and the designers in the Requirements Analysis phase of software engineering.

1.2 Informality and Formality in Specification Languages

Specification languages can be classified into two basic classes: formal specification languages and informal specification languages. It is a well known fact that the strengths and weaknesses of the informal and the formal specification of software systems are largely complementary in nature and not competing. [Fraser'91]. Methods to integrate the two of them are required. The critical step in Requirements Analysis is when the goal oriented requirements specification is transformed into a process oriented form that specifies how the requirements are to be achieved. The process oriented specification should express the functionality of the system and only then can the feasibility be analyzed and consistency verified. Only a well formed specification provides conciseness, precision and a mathematical basis for proofs of consistency and syntactic correctness of the specifications. Formal specifications must remove areas of doubt in a specification. Their principal value in the software process is that it forces an analysis of the system requirements at an early stage. Furthermore, the transformation from formal specification to formal implementation is much easier and the risk of misconceptions and ambiguities is reduced. These are the rationales in support of formal specifications.

Considerable research is being done to introduce formal methods in all the stages of the software development process. Even though the advantages accruing from the use of formal methods are quite valuable and convincing, the process of writing formal specifications is a knowledge intensive and probably error prone activity. if attempted by one not well versed in the formalism. The strict semantic interpretations of formal specifications and their terse textual nature do not provide a good medium of communication among the users and the analysts.

It has been agreed that an "English-like" specification language provides an ideal vehicle for eliciting user requirements. Informal specifications have advantages for requirements elicitation because of their ease of learning and they are more suited for communication among naive users. Informal languages use a combination of graphics and semi-formal textual grammars. Because they are not rigid. informal languages often leads to imprecise and ambiguous statements. Thus. there is always a possibility that the user and the analyst or designer have their own interpretations of the requirements which might not be the same. For that reason. several attempts have been made to "formalize" the design of informal languages. We believe that the introduction of formal methods at the early stages of the software engineering development process should not be targeted to totally replace the prevailing informal methods. Both informal and formal languages possess their own strengths and weaknesses in the Requirements Analysis phase and they should be used to properly complement each other. At one end. informal languages are required to communicate with naive users: at the other end. designers need the rigidity of formal languages to specify the requirements. This thesis presents an approach to bridge these two ends.

1.3 The Transformation Approach

Recently. people are realizing the need for a combination of both informal and formal methods to bridge the gap between informal elicitation and formal specification of requirements. Researchers have recognized the need for computer-based tools which aid human designers formulate formal specifications. One approach taken by [Balzer'78] is to develop partial formal descriptions and use computer-based tools that will supplement context to complete these descriptions for constructing a formal specification. Another approach. presented in this thesis. is to have a semi-formal language which is easy to learn but which possesses a structure that will assist in the generation of a formal specification. As discussed in [Balzer'78], the side effects of being informal are: ambiguity, contradiction, poor ordering of requirements, incompleteness or partial specification and inaccuracy. On the other hand, an informal specification comes naturally to the user and the methods used are relatively easy to learn. In the approach taken in this thesis, we view informality and formality as two opposite ends of a spectrum, between which there are several "semi-formal" methods or languages for specifying requirements. A semi-formal description will possess the characteristics of both informal and formal languages. Then, the transitions from informal to semi-formal and from semi-formal to formal will be less rigorous than jumping from informal straight to a formal specification.

A semi-formal medium to express requirements is presented in this thesis. From which a knowledge-based transformation system will generate specifications in a formal language. In this approach, a user develops the specification interactively using a computer-based tool. We distinguish two stages in the formalization process ([Miriyala'91a]): data modeling and specification of operations. For each stage, a distinct software tool is provided. It consists of a graphical component called MER diagram and a textual component called KFD. We believe that they are easy to learn and are closely related to the user's natural way of thinking. The **M**odified **E**ntity **R**elationship model and **K**eyword-based **F**ormatted **D**escriptions together form the semi-formal specification provided by the user. The transformation system translates the MER and KFDs into a formal specification. The target formal language chosen is VDM. The **V**ienna **D**evelopment **M**ethod has been chosen because of its wide acceptance, by both academics and industrial software engineers, as a formal language in the software development. VDM makes use of formal notations for the specification of functional properties of systems. It is not a programming language since it contains non-executable constructs. It is intended for the definition of the functional requirements in a very abstract but formal manner. It offers both specification notation and proof obligations which enable a designer to establish the correctness of design steps. Chapter 2 provides a brief introduction to VDM.

Generating the semi-formal description from the informal requirements is simple because of the "English-like" aspects and the diagrams of the semi-formal description. Then, moving from the semi-formal description to a formal specification (in our case, VDM) can be done by the transformation system. That is how the work presented in this thesis bridges the gap between informal requirements elicitation and formal specifications.

1.3.1 Global View of the Picture

The ultimate goal that we are trying to achieve here is to provide tools to help the system developers in all the phases of the software development life-cycle. By integrating computer-based techniques in the Analysis, Design or Implementation phases of the development process, we attempt to diminish the burden of such tasks. Thus, the work presented here is only one step towards the achievement of this long term goal. Figure 1.1 gives a summarized picture of how we view the development process and where the work presented in this thesis fits in the complete process.

In the analysis phase, the MER and KFDs are description mediums for writing the user requirements. Those requirements are provided by the analyst and/or user. Two appropriate editors are provided to support the writing of MER and KFDs. Then, a computer program with possibly some help from the analyst will "automatically" generate a VDM specification. The next stage is the design phase which follows an Object-Oriented approach. Classes of objects will be automatically derived from the VDM specification. This is the work presented in [Gao'92]. In fact, it would not be right to use the word "automatically" in this context since the analyst may be involved (through questions/answers for clarification) in the generation of object classes. By "automatically" we mean that the work is done (in total or partially) or supported by a computer program. The next step in the design phase is to produce a detailed design description of the classes. From the VDM specification, it is possible to create a high level description of the operations required to handle each class of object. However, to produce a detailed description of the operations (by "detailed" we mean pseudo-code), the required information must be provided by the domain expert and designer. Finally, in the implementation phase, program synthesis techniques will be used to generate the code from the OO design specification. At this stage also, the program synthesis generation is not totally automatic. The programmer might also be involved in the process. Computer-based tools greatly simplify the development process and, above all, the maintainability of the system. They also reduce special training requirements (e.g. expertise in formal notations).

1.3.2 The System Components

Figure 1.2 gives a complete picture of the transformation approach presented in this thesis for requirements analysis. The user inputs the MER diagram and the KFDs through two specialized editors. The MER diagram is a simple modification of the well known Entity-Relationship approach to data modeling, [Desai'90]. The modifications are introduced essentially for the purposes of abstraction. The MER model captures the different system entities and the relationships among themselves at various levels of abstraction. The KFDs form a set of textual descriptions given according to a fixed format using keywords. The KFDs describe the functional requirements of the software system i.e. the set of activities the system must perform. In building the MER model, the user can use *predefined entity* types which are known facts about the current problem domain. Similarly, *higher-order keywords* are available to the user in writing a KFD in addition to the fixed set of general keywords. The higher order keywords can be seen as macros which manipulate predefined entities and they are analogous to domain specific jargons. Predefined entity types and the associated higher-order keywords constitute the stored information about the problem domain. We assume this knowledge is provided by a specialist in that particular problem domain. Both the MER and KFDs can make use of this stored knowledge.

Before transforming the semi-formal input into VDM, the *description preprocessor* verifies the syntax and makes use of common knowledge to assure the consistency between the MER and KFDs. It creates a semantically complete description and produces an "intermediary form" of the description. This preprocessing step may interact with the user to clarify certain ambiguous situations. Common knowledge

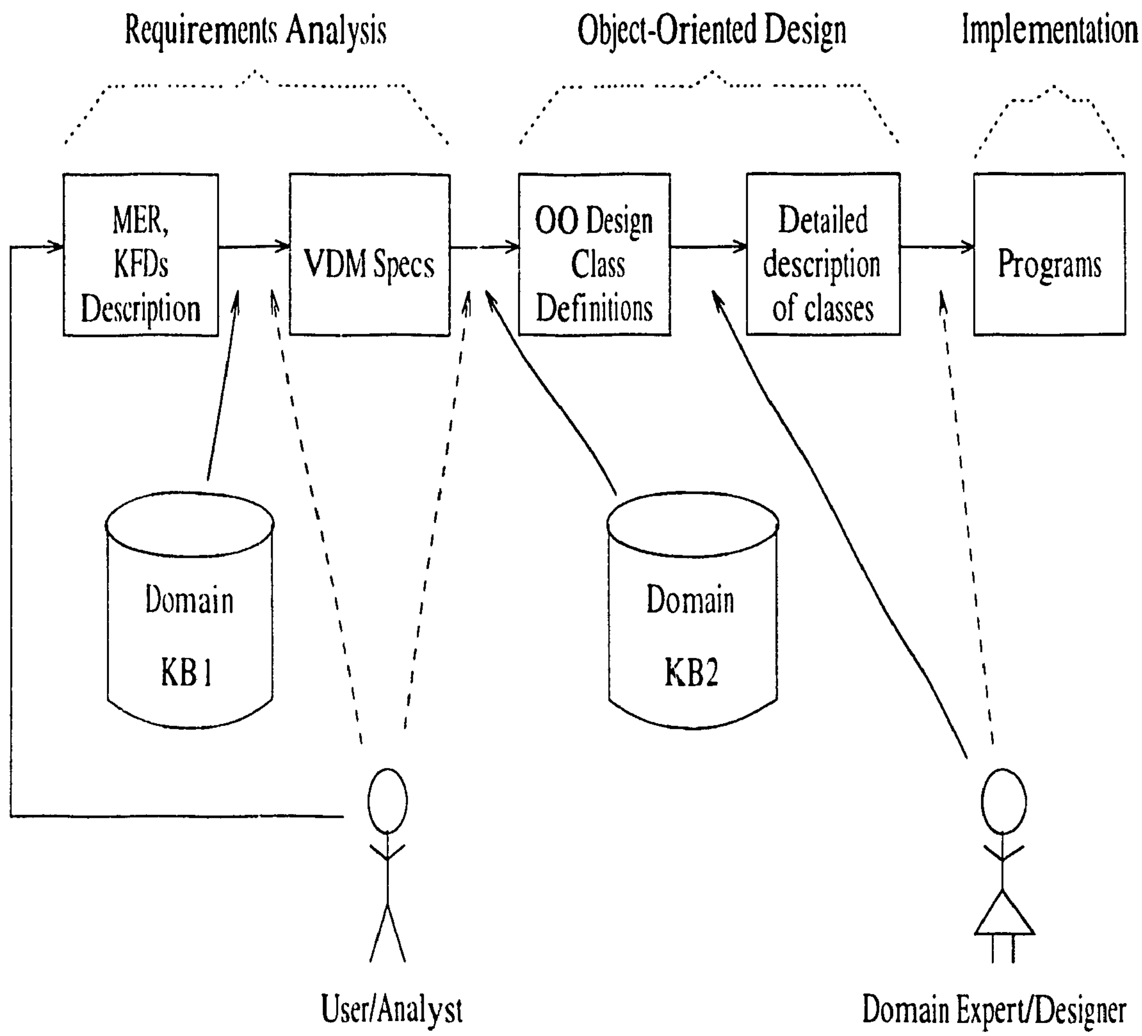


Figure 1.1: Complete picture of the development process

involves knowledge of set theory, relationships and functions. The transformation step takes this intermediate form as input and uses knowledge about the VDM syntax and translates the description into a VDM specification. The transformation process is based on a set of transformation procedures. These procedures and the VDM knowledge are independent of the problem. Finally, as a post-processing step, the generated VDM specification could be optimized to provide an improved specification. Although this part of the system is not covered in this thesis, it has been included for the sake of completeness. All the components of figure 1.2 will be described in more detail in chapter 5.

1.4 Overview of the Thesis

The goal of this thesis is to present and describe a transformation approach to reduce the barriers in writing formal specification. By providing a semi-formal specification language (MER and KFDs) which does not require a specialist in formal notations, the formalization stage of the Requirements Analysis phase becomes less rigorous. At the same time, a formal specification will be automatically generated.

The second chapter provides a short introduction to VDM. It is not a complete description of the VDM language but it explains the basic notations which are sufficient to understand the specifications included in this thesis.

The third chapter is concerned with the data modeling aspects. A review of the well known Entity-Relationship model is given and the components of the E-R diagram are explained. The proposed MER data model is then presented. The details about the MER components and their differences with the E-R model concepts are outlined.

In the forth chapter, the KFDs are introduced. The list of the valid keywords of the language for describing an operation is given and their semantics are explained. The format of a KFD is described in detail. Finally, the syntax of the KFD specification language is presented.

Chapter 5 is concerned with the transformation of the semi-formal description into

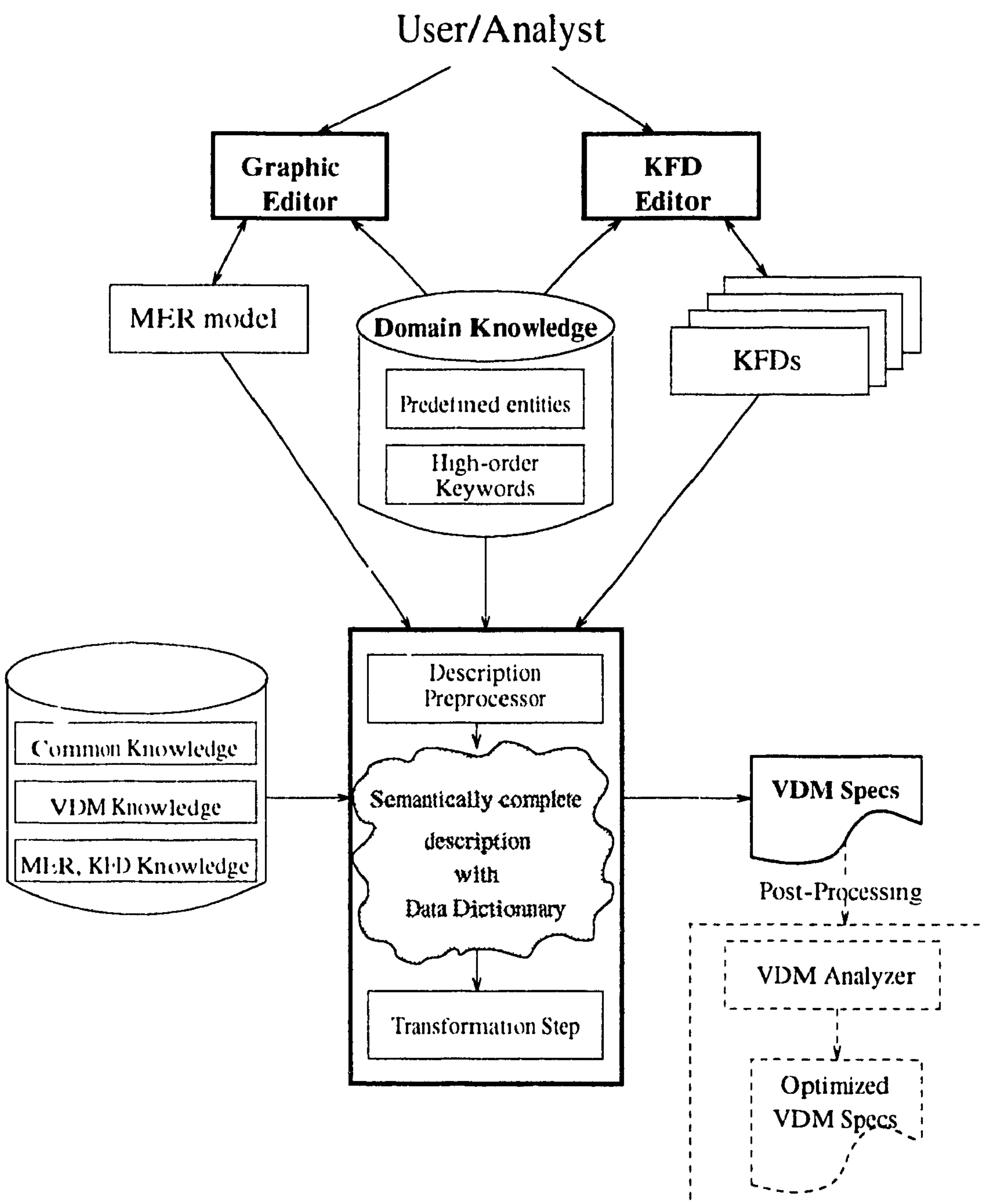


Figure 1.2: System architecture

a formal specification. First, the tools to build the MER and KFDs are described in detail with their user interface. We emphasize how domain facts are used in the semi formal description. Then, we explain the role of the description processor and how an intermediary form is produced. Finally, we go through the steps of the transformation process. The set of transformation rules are presented. The explanation of each subsystem is illustrated through an example.

Chapter 6 compares this thesis work with what was already done by others and presents the conclusion and future work.

Chapter 2

VDM Specification Language

The Vienna Development Method (VDM) is a model-based approach for system specification. In this approach, descriptions of systems are given as models. VDM has been developed in the 1970s by the IBM Vienna Research Laboratories. Apart from being a specification language, VDM provides rules and procedures to be followed in the various stages of system development. The constituents of models are data objects representing the inputs, outputs and internal 'state' of the system; and operations and functions which manipulate the data. Data types define classes of data objects requirements. The VDM standard used in this thesis is taken from [Cohen'85]. There are new versions of VDM as of today but for the purpose of the work presented here, the notations given in [Cohen'85] are sufficient. A more detailed description about VDM can be found in [Jones'90b].

Basically, there are two major parts in a VDM specification. First, abstract data types are introduced to define abstract 'variables' which are used to represent the internal state of the system as a model. The second part is the definition of operations and functions which act on the variables. The operations are the services available to the users of the system being specified.

2.1 Variables and Types

The basic notation in VDM specification language is a *set*. A set is a collection of similar items, such as the set of integers or strings. Each item in a set 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 the variable may take. At any time, the variable may assume one of the values in the set. The basic built in types provided in VDM are:

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

New variable types can be created by defining the set of all possible values the variable can take. This is how the built-in *Bool* type is defined. For example:

$Colour = \{\text{RED, BLUE, YELLOW}\}$

$Bool = \{\text{TRUE, FALSE}\}$

Types can be manipulated in several different ways. We may give an alternate name to a type or create a subtype of a numerical type. For example:

$Width = Nat0$

$Deck = \{1{:}52\}$

The VDM convention is to have variable names in upper case letters in the declarations and types starting with an upper case letter (the rest is lower case). A variable is defined (declared) by providing a name and its type, as follows:

$LIGHT = Colour$

$CARD = Deck$

2.1.1 Powerset Types

Variables can take as their value a set of objects of some type. For this, VDM provides the powerset type which is the set of all subsets of the base set (type). The suffix *-set* appended to a type name creates the powerset for that type. For example, the powerset of the type *Colour* would be

$$Colour\text{-}set = \{ \{RED, BLUE, YELLOW\}, \{RED, BLUE\}, \{RED, YELLOW\}, \{BLUE, YELLOW\}, \{RED\}, \{BLUE\}, \{YELLOW\}, \{\} \}$$

A variable of the powerset type *Colour-set* will take as value any combination of colours or the empty set.

VDM provides set operators to manipulate sets and create new ones. These are **union** ($\cup$), **intersection** ($\cap$), and **difference** (-). Boolean operations are also available to operate on sets. These are tests for set **membership** ($\in$) or **subset relationship** ($\subset$). Finally, the **cardinality** (**card**) of a set returns the number of elements in the set.

2.1.2 List Types

VDM provides a useful data structuring facility called a list type which is an **ordered** collection of values. The suffix *-list* attached to a type name creates the set of finite lists which can be made from the base type. The list type is an infinite set as shown in the example below:

$$\begin{aligned} Colour\text{-}list = \{ & <>, \quad /^{*} \text{ the empty list } ^{*}/ \\ & < RED >, < BLUE >, < YELLOW >, \\ & < RED, RED >, < RED, BLUE >, \ldots \\ & < RED, RED, BLUE >, \ldots \\ & < RED, BLUE, YELLOW, RED >, \ldots \ \} \end{aligned}$$

The operators provided for manipulating lists are **hd**, **tl**, **len** and **elems**. These operators yield the 'head' of the list (first element), the tail of the list (the rest of the list after the head is removed), the length of the list (number of listed items) and the set of elements of the list, respectively. Lists can be concatenated with the '||' operator and an element of the list can be accessed by referring to its position in the sequence. These operations are illustrated in the following examples:

assume l_1 = <A, B, C, B, A> and l_2 = <C, D, D>

then

hd l_1 = A

tl l_1 = <B, C, B, A>

len l_1 = 5

elems l_1 = {A, B, C}

$l_1 \| l_2$ = <A, B, C, B, A, C, D, D>

$l_1(4)$ = B /* the fourth element in the list l_1 */

2.1.3 Record Types

The record type allows a variable to assume a fixed length sequence of values drawn from different types as its value. Each value in the sequence is called a **field** of the record. VDM distinguishes record type definitions by the '::' symbol. The valid format for a record type definition is shown in the following example:

Person :: NAME : *String*
AGE : *Nat0*
SEX : {M, F}
MARRIED : *Bool*

In this example, a variable of type *Person* will be defined by a quadruple composed of a name, age, sex and indication of marital status.

Two operations are available to handle variables of record type. First, we can access the fields of the record and second, we can construct an instance of a record type with the make (**mk**) function which coerces discrete values into a record structure in a special notation. It is also possible to compare (test for equality or inequality) two records. For example,

$pers_1$ = **mk-Person** ('Joe Blow', 28, M, FALSE)

creates an instance variable of type *Person* named Joe Blow, who is 28 years old, who is a male, and is not married. Then, we can access each field individually to obtain the values for variable $pers_1$ as shown below.

NAME($pers_1$) = 'Joe Blow'

$AGE(pers_1) = 28$

$SEX(pers_1) = M$

$MARRIED(pers_1) = FALSE$

2.1.4 Mapping Types

Finally, the last data structure provided by VDM is the mapping type. A mapping type has a finite domain and provides an alternative way to define functions where elements on the domain set is mapped into elements of the range set. The difference between mappings and functions is that a mapping has a finite domain set. We can explicitly define mappings as shown below where elements of *Colour* are mapped to elements of the set $\{1, 2, 3\}$.

$$map = [red \rightarrow 1, blue \rightarrow 2, yellow \rightarrow 3]$$

Mappings are uniquely defined for elements of the domain. VDM provides operators to manipulate mappings. Operators **dom** and **rng** yield the domain and the range set of the mapping, respectively. The binary operator **overwrite** denoted as '†' combines two mappings and yields a new mapping. When a domain element appears in both the given mappings, the range element of the second operand determines the range of the resultant mapping. In the case where there is no duplication of domain elements in both mappings then the resulting mapping is a simple union of the two sets of mappings. The **restrict by** and the **restrict to** operators, denoted respectively by '\' and '/' take a mapping and a set as their operands and create a new mapping as their result. The **restrict by** removes from the mapping domain the elements specified in the given set. The **restrict to** keeps only the elements specified in the given set as the domain elements of the resulting mapping. The following examples illustrate the application of the operators described above.

assume $m_1 = [A \rightarrow 3, B \rightarrow 2, D \rightarrow 2]$ and

$m_2 = [C \rightarrow 1]$ and

$m_3 = [A \rightarrow 2, E \rightarrow 1]$

and $s = \{A, B\}$

then

$\mathbf{dom}\ m_1 = \{A, B, D\}$

$\mathbf{rng}\ m_1 = \{2, 3\}$

$m_1 \dagger m_2 = [A \to 3, B \to 2, C \to 1, D \to 2]$

$m_1 \dagger m_3 = [A \to 2, B \to 2, D \to 2, E \to 1]$

$m_1/s = [A \to 3, B \to 2]$

$m_1 \backslash s = [D \to 2]$

We can create powersets of mapping types in VDM as shown below. This defines a type called *Map* and a variable of that type has as its value a set of mappings with domain values of type *Colour* and range values of type *Int*, or the empty mapping set denoted by [].

$Map = Colour \to Int$

2.2 Predicate Logic

In VDM the operations are defined using the predicate logic. The predicate logic operators are simply listed here. A complete description of predicate logic can be found in [Jones'90b].

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

VDM also provides a facility to introduce textual substitution into a predicate. The **let ... in** construct allows shorter predicates to be written. The scope of the variables introduced in the **let** clause is the predicate which follows it. For example:

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

v = d ** 2 + d + 1

is shorter than writing the following expression:

v = (a + b) ** 2 + (a + b) + 1

2.3 States, Invariants and Operations

As mentioned before, a variable type is an abstraction of a physical object and instances of variable types are used to represent the internal states of the objects. In VDM, the set of variables form the system model which characterize the system state.

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

The variables defined in the system state are known in the context of every operation defined in VDM. Hence, those variables are global to all the specified operations. Variables created within an operation or which are the parameters of the operation are local variables for that operation. Operations specify changes to the values of some global variables defined in the state. The specification of an operation consists of four parts:

1. The name of the operation and any input or output parameters it takes.

2. The **external** clause indicates which part of the state the operation needs to access. For each state component accessed, read only or read/write access is specified by the keywords **rd** and **wr** respectively.

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

4. The **post**-condition part, which indicates how the values of the variables are affected by the operation and possibly, how the output parameters are 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 suffixed with a prime.

The following is a simple example taken from [Cohen'85] showing a sample VDM specification. The example contains the system state and three basic operations.

State :: UNMARRIED : *Person-set*
MARRIED : *Person-set*
Person : /* The type definition given in a previous example */

INIT ()
ext UNMARRIED : **wr** *Person-set*
MARRIED : **wr** *Person-set*
post $unmarried' = \{\} \wedge married' = \{\}$

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

MARRY (M: *Person*, W: *Person*)
ext UNMARRIED : **wr** *Person-set*
MARRIED : **wr** *Person-set*
pre $m \in unmarried \wedge w \in unmarried$
post **let** $couple = \{m, w\}$ **in**
$married' = married \cup couple \wedge$
$unmarried' = unmarried - couple$

In this example, one invariant which is clear from the specification is that no individual person should be present in both the set of unmarried clients and the set of married clients. This is specified by adding the following predicate to the specification:

$$\textbf{inv-}\mathit{State} \triangleq \mathit{unmarried} \cap \mathit{married} = \{\}$$

It can easily be proven that this predicate will always hold after the execution of each operation.

2.4 Why VDM?

The advantages of using VDM stem from the fact that it is a model-oriented approach. In that sense, it is close to the user's way of conceiving the problem. The VDM state defines types from which it is possible to automatically extract classes of objects and derive an object-oriented design. VDM encourages a layered top-down development of systems, by supporting abstraction at the uppermost levels of description. At the top level, a specification is given as an abstract model which captures only those system concepts necessary to explain the required functions of the system. This top-down development process is also encouraged with the MER and KFDs. In VDM, the data objects are specified using very abstract mathematically oriented data types. Then, at each refinement stage, it is easy for the designer to refine the abstract data structures into more implementation-oriented data structures like trees, arrays and so on. This facilitates the task of the programmer.

Moreover, automatic proof checking systems exist to verify a VDM specification. The Mural System is one of them [Johnson'92]. It is primarily concerned with providing support for the construction of formal mathematical proofs to VDM specifications. It can be used to verify the internal consistency of a specification by discharging the appropriate proof obligations. The Mural system can also be used to validate a VDM specification against an informal description of the system being specified (in our case, the MER and KFDs). This is done by stating and proving the properties the designers believe the system should exhibit. The Mural system works under a window-based environment and is recognized to support the writing of correct VDM specification.

Chapter 3

MER Model

In specifying software system requirements, we consider two aspects: data modeling and specification of operations. In the first case, all the entities of interest to the system are captured. In the second case, all activities required to be performed by the system are described. This chapter covers the first step, i.e. the modeling of entities in the system requirements description and presents the MER model for that purpose. The MER or Modified Entity-Relationship diagram is based on the well known Entity-Relationship approach to data modeling. In section 3.1, the E-R components are reviewed. The modified model is then introduced. The similarities and differences of the modified model with the traditional E-R model will be covered.

3.1 Review of the Entity-Relationship Model

The E-R model is a conceptual schema of an enterprise. It is a data modeling approach to capture the objects and the relationships among these objects which are of interest to the organization. Each class of objects as well as each relationship has properties called attributes. The E-R model does not possess a notation to specify operations on data. The model is independent of any design decisions. In practice, the E-R model is used for the high-level design of databases and to describe their logical structure.

3.1.1 Entity Sets

An entity is a thing that exists and is distinguishable. Entities are basic units used in modeling classes of concrete or abstract objects. An entity can have a concrete existence or it may denote ideas or concepts. An entity set is a group consisting of all similar entities. In the E-R approach, we characterize similar entities by a collection of attributes. The attributes are common to all entities in the entity set. Objects in the entity set are distinguished by their unique identifier (a key) which is formed by a subset of the entity attributes. An entity set can also be called an entity type. For example, a *Car* can be an entity type with the attributes being *brand name*, *number of doors* and so on. An instance of that set is obtained by assigning values to the entity attributes. For example, a *Ford Escort* with *2 doors* is an instance of the entity set *Car*.

3.1.2 Attributes

Attributes are properties of an entity set. Each attribute of an entity takes a value from the domain of values of that attribute. Attributes have simple domain values like integers, real numbers, strings, characters and so on. When the value of an attribute (or a group of attributes of the entity set) uniquely identify each entity of the set, we call it a **key**.

3.1.3 Relationships

A relationship represents an association between entity sets. A relationship set is a collection of relationships of the same type. If **R** is defined as a relationship among entities E_1, E_2, E_3, ..., E_n then an instance of **R** would be a set of tuples (e_1, e_2, e_3, ..., e_n) where e_i is an instance of the entity set E_i and **R** is called a **n-ary relationship**. The most common type of relationship is a binary relationship between two entity types. Relationships can also be characterized by attributes.

3.1.4 Is-a Hierarchy

The **Is-a** component is a special relationship used for abstraction purposes. If A **Is-a** B between entity sets A and B, then A inherits the properties of B. Thus, all the attributes defined in B also exist in A but A can also have some additional attributes that don't exist for entities in B. In that case, we say that B is a **generalization** of entity set A or that A is a **special kind** of entity set B. Generalization means viewing sets of objects as a single general class by concentrating on the common characteristics (attributes) of the constituent sets while ignoring their differences. A high-level entity type is produced by the union of lower-level entity types. For example, consider the entity types of figure 3.1. The high-level entity *Vehicle* is a generalization of the lower-level entities *Bicycle* and *Automobile*. On the other hand, specialization means introducing new characteristics to an existing class of objects in order to create a new class of objects. Thus, low-level entities are produced by adding characteristics to an existing higher-level entity type. For example, entity *Bicycle* is a specialization of entity type *Vehicle*. It inherits attributes *Price* and *NbWheels* from *Vehicle* but possesses the additional property *NbSpeeds* which further characterizes a bicycle.

3.1.5 The Entity-Relationship Diagram

The E-R diagram graphically represents the entities, their attributes and the relationships among entities. The components of the E-R diagram are:

RECTANGLE A rectangle represents an entity set. The rectangle is labelled with the name of the entity set.

OVAL Attributes are represented with ovals. They are linked to their corresponding entity sets by **undirected** edges. Key attributes are sometimes underlined.

DIAMOND Relationships are represented with diamonds. They are connected to entity sets by edges (undirected or directed, depending on the notation used to indicate the functionality of the relationship, as explained in the next section). The order of the entity sets in a relationship can be indicated by num-

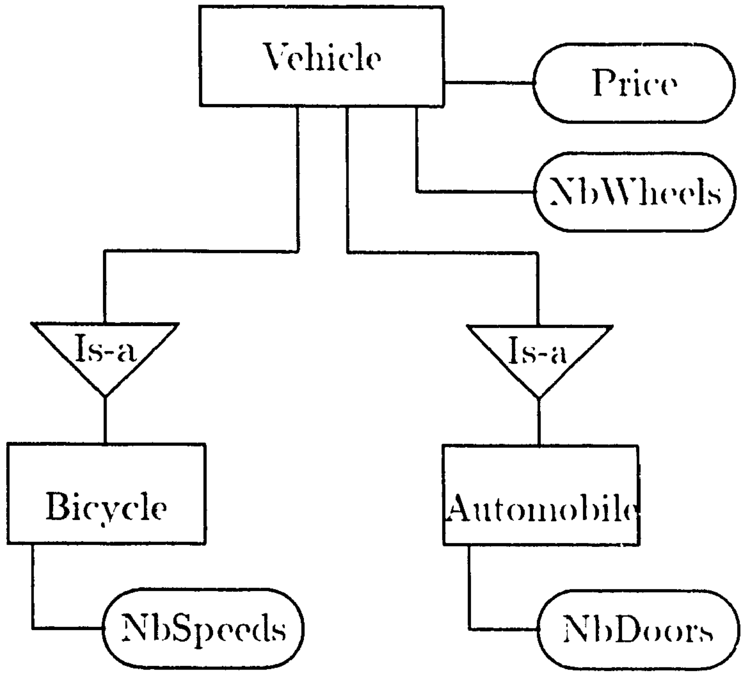


Figure 3.1: Example of Is-a relationship

bering edges. For example, the relationship **R** in figure 3.2 is a set of tuples (e_1, e_3, e_2, e_1) where e_i is an entity in the entity set E_i.

TRIANGLE The **Is-a** relationship is represented by a triangle. If entity A **Is-a** B then A and B are connected to the **Is-a** triangle and the triangle points towards the entity A where A is a special kind of B.

3.1.6 The Functionality of a Relationship

In the real world, we can have x number of entities related to y number of other entities. Thus, we need a notation to indicate what is the functionality of the relationship. Broadly speaking, there exist three kinds of relationships in the E-R model.

One-to-One A one-to-one relationship between any two entity sets means that for each entity in either set, there is at most one associated member of the other set. Graphically, this is represented as two edges connecting the relationship

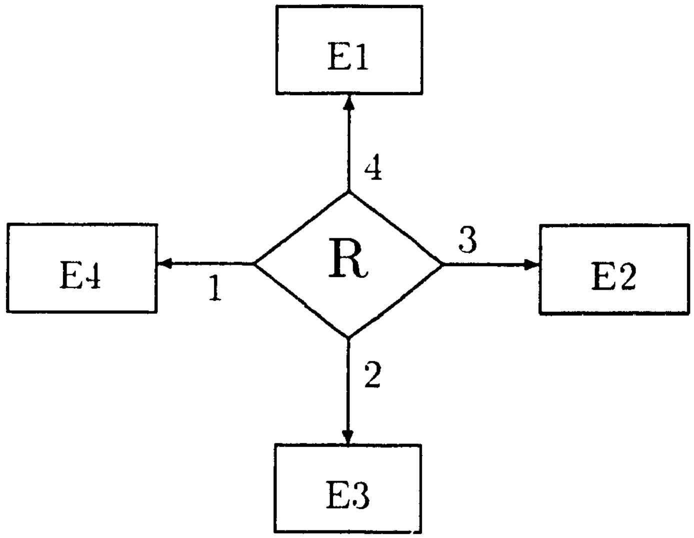


Figure 3.2: A relationship among 4 ordered entity types

diamond to each entities. For a one-to-one relationship, the edges are not labelled.

Many-to-One A many-to-one relationship between any two entity sets E_1 and E_2 means that one entity in set E_2 is associated with zero or more entities in set E_1. However, each entity in E_1 is associated with at most **one** entity in E_2. Graphically, this is represented with a "M" labelling the edge connecting the relationship diamond to E_1 and with a "1" labelling the edge connecting the relationship diamond to E_2.

Many-to-Many A many-to-many relationship between entities E_1 and E_2 has no restrictions as to the number of entities in set E_1 associated with an entity in the set E_2 and vice versa.

3.2 The Modified Entity-Relationship Model

This section presents the approach chosen for the modeling of the entities in the system requirements description. The model is a simple enhancement of the E-R approach and is called the MER model. The MER model describes the system entities, attributes and relationships at various levels of abstraction [D'Almeida'92]. The Entity-Relationship approach models the most primitive data. In this sense, it fails to model higher level entities which are composed of relationships between low level entities. In the original E-R approach, an entity set is characterized by attributes only. We call them as primitive entities. The MER model captures composite objects or entities. A composite entity (also called **higher order abstract** entity) is defined as a collection of attributes and/or relationships among other entities. The same components of the E-R model (i.e. entities, attributes, relationships) exist also in the MER model.

3.2.1 Entities

An entity in the MER model has the same meaning as in the E-R model. The only difference is that entities are seen as basic units in the E-R approach while they can be seen at higher levels of abstraction in the MER model. In the original model, an entity is characterized by its attributes. In the new model, the entity is characterized by its attributes, if any, and possibly by a set of relationships among other entities which construct this higher order abstract entity. Thus, each entity is composed of zero or more attributes and zero or more relationships. An MER entity can be described by means of other entities and relationships, and hence abstracts the concept of entity association in their definition.

3.2.2 Entity Sets

An entity set (also called entity type) is a group of similar entities, as defined in the E-R model. Similar entities are characterized by a set of properties which is a collection of attributes and/or relationships among other entities. Entity sets are represented with a rectangle as in the original approach. In the MER diagram, the properties of an entity set (i.e. what defines it) is determined by all its *outgoing* arrows. An arrow leaving the entity type points to either an attribute (ellipse), a relationship (diamond) or the *Is-a* relationship (triangle). Consider the example in

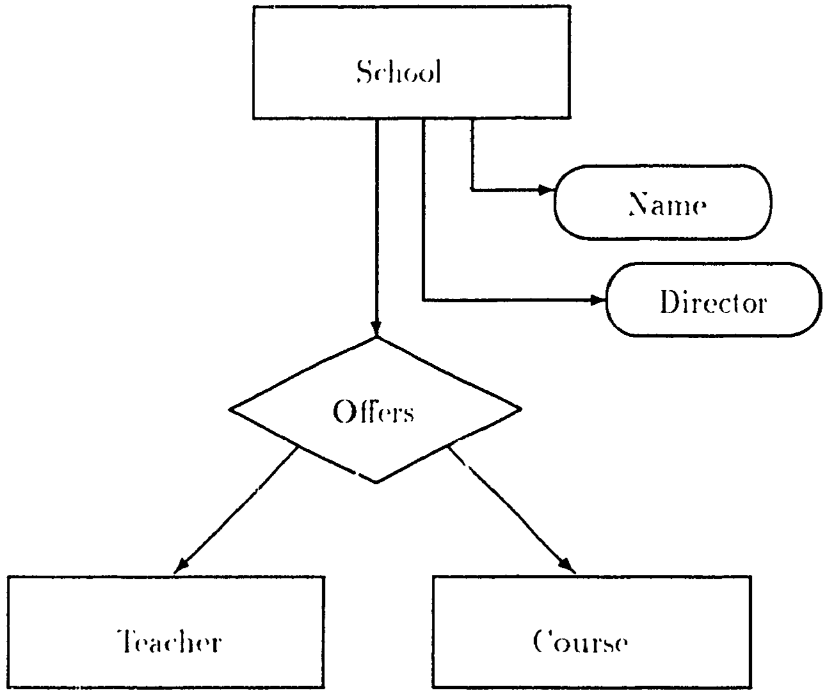


Figure 3.3: MER entity composed of two attributes and a relationship

figure 3.3. The entity set SCHOOL is defined with two attributes and a relationship between entity types TEACHER and COURSE. By definition, entity SCHOOL is a **higher order abstract** entity since it is defined by a relationship between lower level entities. Relationships will be further explained in 3.2.4. Entity types which are defined only by attributes are called **primitive** entities . Those are equivalent to entity types in the E-R model. Entity types defined by one or more relationships are called **higher order abstract** entity types. They will be further discussed in section 3.2.3.

An entity set can have a single attribute. In such cases, the attribute is denoted by the entity name. We call it a **single-value** entity set.

3.2.3 Higher order Abstract Entity Sets

Higher order abstract entity sets are created mainly to abstract relationships of lower level entities. Before describing the concept of a relationship, let us first understand

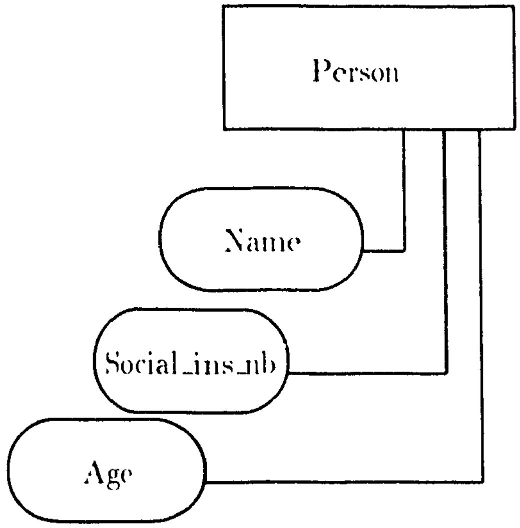


Figure 3.4: E-R model of entity PERSON

the relations that may exist among the attributes of an entity. In the E-R model, entity sets are characterized by a set of attributes. This group of attributes forms a record which is the representation of the entity. For example, the entity in figure 3.4 is described as follows:

entity set PERSON

Name : **string**

Social_ins_nb : **numeric**

Age : **numeric**

Consider the attribute (relative_name, relationship) of the entity PERSON, as shown in figure 3.5. This would give the names of the relatives and their relationship with the person (brother, cousin, and so on). This new attribute can possess multiple values for one person as shown in the record of figure 3.6. One name is associated with one age and one social_ins_nb but many values of the attribute (relative_name, relationship).

The attribute with multiple values is called the **repeating group**. To handle cases of such one-to-many association between sets of attributes, we separate them

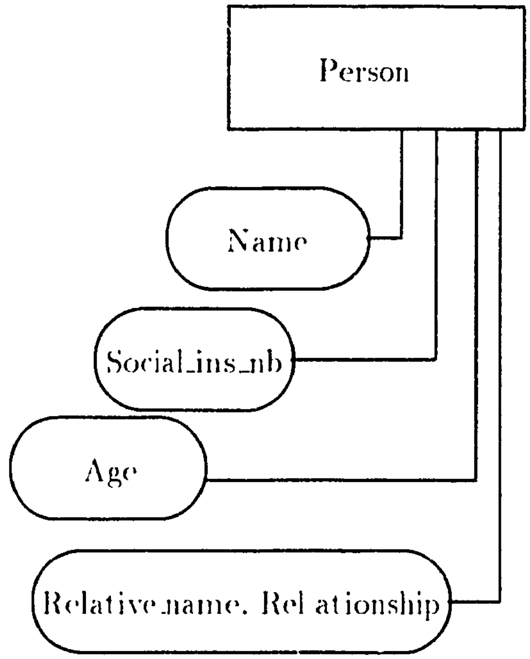


Figure 3.5: E-R model of entity PERSON and possible attributes

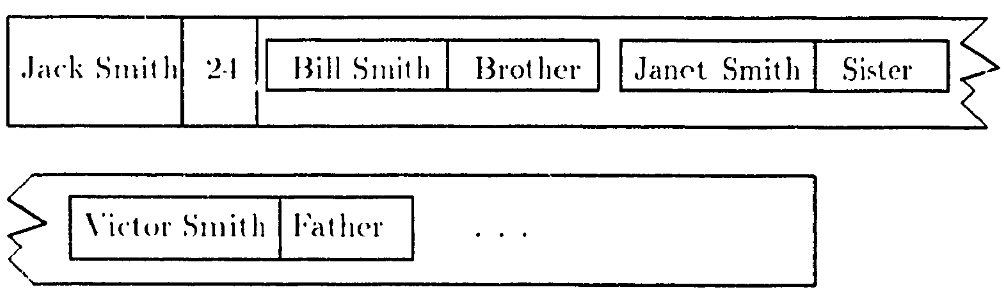


Figure 3.6: Record for the entity PERSON

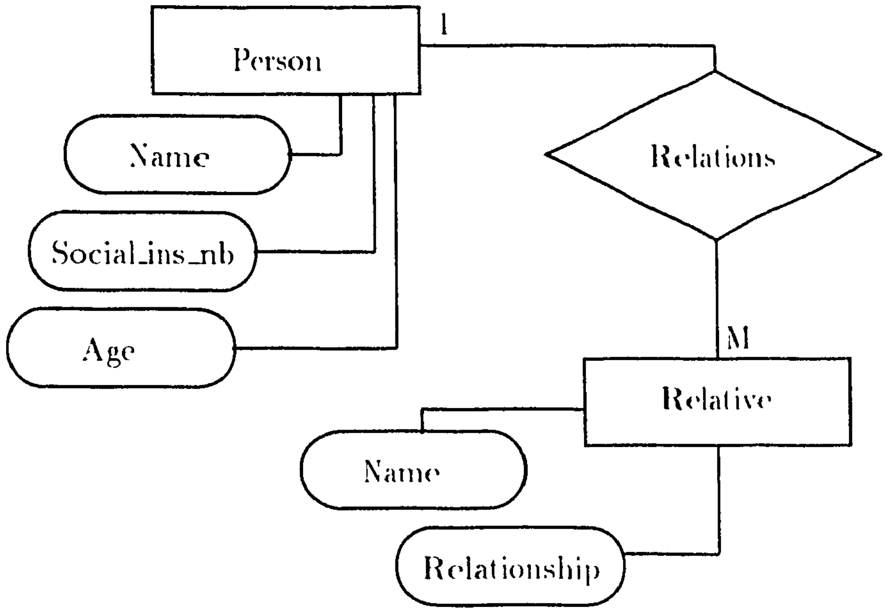


Figure 3.7: E-R model of entity PERSON after separating the repeating group

into another entity set as shown in figure 3.7. In such a case, the entity person is not only defined by its attributes but also by its relationship with the entity RELATIVE. We can view the entity PERSON having attributes name, age, social_ins_nb and (relative_name, relationship) or having attributes name, age, social_ins_nb and a relationship with a distinct entity RELATIVE. In both cases, (relative_name, relationship) is an attribute that brings more detail about a person. For that reason, in the MER model we call this type of entity **higher order abstract** entity, that is, an entity which is defined by attributes, if any, and by one or more relationships with other entities. In the MER approach, the outgoing arrows of an entity set can be read as **has**. For a higher order abstract entity like PERSON, we say that a person **has** a name, an age and **has** many relations, as shown in figure 3.8. We distinguish whether an entity is higher order abstract entity or not by its outgoing arrows. If there is an arrow from an entity to a relationship, then that entity set is **higher order abstract** entity and the relationship becomes part of the definition of that entity. It is hard to distinguish between a one-to-many relationship among entities and a relationship

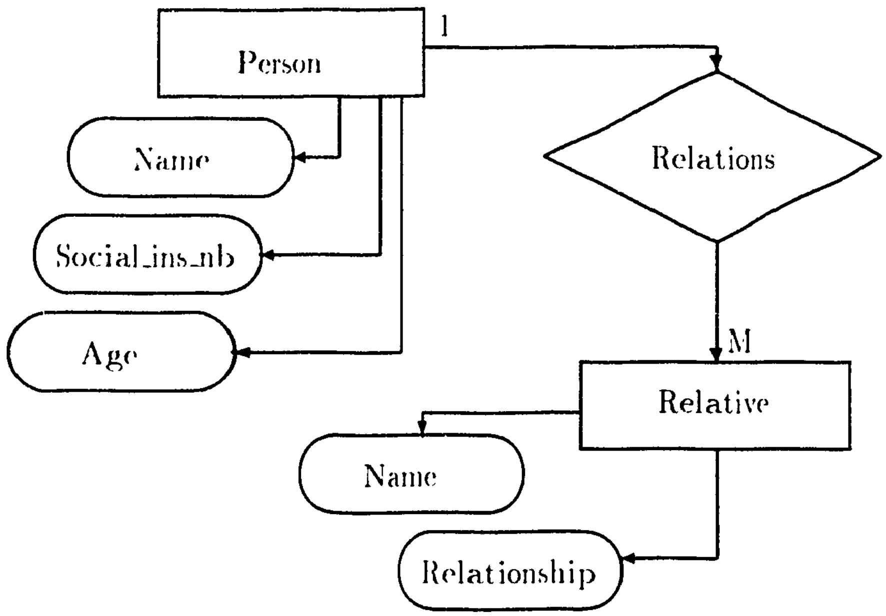


Figure 3.8: MER model of entity PERSON after separating the repeating group

derived from a one-to-many association among attributes. Again, it depends on the requirements of what is to be modeled. Both ways are valid. It is only a matter of how to organize the information to best match the requirements.

We have seen so far how a higher order abstract entity is created from a one-to-many association among attributes. Another case where a higher order abstract entity is used is for the concept of aggregation. In the real world, a relationship between objects is something abstract. We need to refer to the whole relationship and view it as a higher-level object. Aggregation is the process of grouping details and abstracting to a higher-level object. In the MER approach, every relationship is part of the definition of a higher order abstract entity type. Therefore, for any relationship **R** in the E-R model, there will exist a higher order abstract entity type **E** in the corresponding MER such that **R** is one of the properties of **E**. Consider the E-R model example in figure 3.9 and the relationship **R** between entity types E_1 and E_2. **R** possesses attributes $attr_1$ and $attr_2$. In the MER approach, we aggregate all the concepts in figure 3.9 and create a higher order abstract entity type **E**. This is

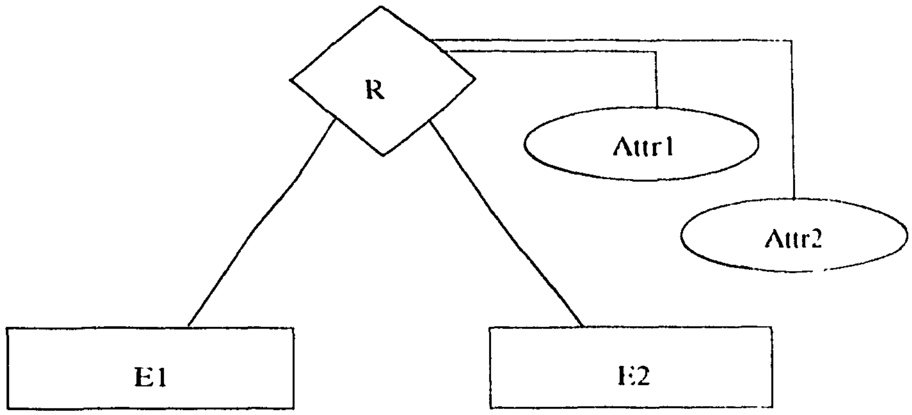


Figure 3.9: E-R relationship with attributes

shown graphically in figure 3.10. The direction of the arrows in figure 3.10 indicates that the relationship **R** between E_1 and E_2 is part of the definition of entity set **E**, as well as attributes $attr_1$ and $attr_2$. Therefore, for each entity e in **E**, e will be defined by $attr_1$, $attr_2$ and a set of tuples (e_{1i}, e_{2i}) where e_{1i} is an entity of type E_1 and e_{2i} is an entity of type E_2.

In the MER model, the entities of a relationship will be those specified by the outgoing arrows of the relationship diamond. An entity linked by an incoming arrow to a relationship diamond is the **higher order abstract** entity being defined and is not considered part of the relationship as such. Thus, complex relationships are encapsulated in the definition of a higher o.der abstract entity.

3.2.4 Relationships

As we have just explained, the concept of a relationship in the MER approach is interpreted differently from the E-R approach. In MER, a relationship can be defined as a set of entities of one entity type (refer to figure 3.8). There are two other ways: as a set of tuples associating two entities from two entity types; as a set of tuples associating an entity of some type to another relationship set. Graphically a relationship is represented by a diamond. The arrows leaving a relationship diamond indicate the entities and/or other relationships which are part of the current

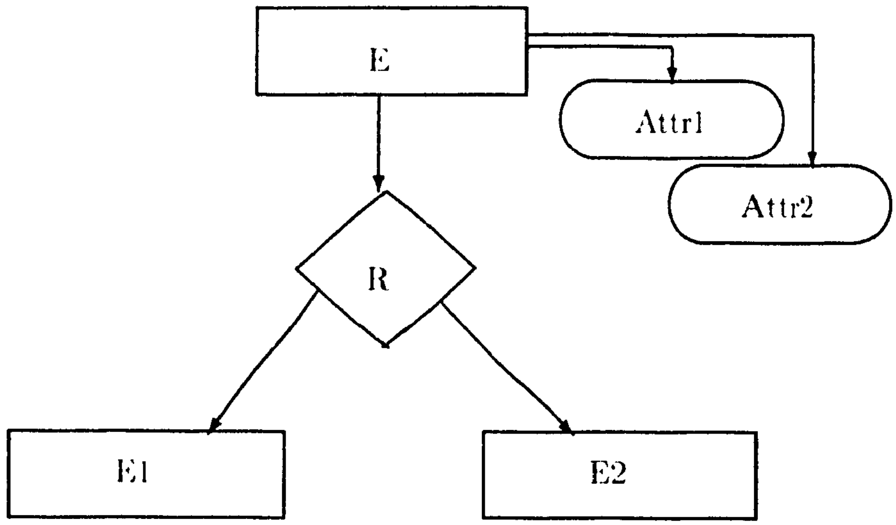


Figure 3.10: High abstract entity in the MER model

relationship set. If a relationship has only one *outgoing* arrow, we call it a **primitive** relationship. If a relationship has two *outgoing* arrows pointing to two entity types E_1 and E_2, we call it a **second-order** relationship. If a relationship has two *outgoing* arrows, one pointing to an entity type and the other pointing to another relationship diamond, we call it a **high-order** relationship. These are the only allowable types of relationships in the MER model. A relationship can associate only two things, either two entity types or an entity type to another relationship. Let us explain formally the distinction between the different types of relationships in MER.

PRIMITIVE A **primitive** relationship **R** is defined as a set of entities of one entity type. If **R** is defined as a primitive relationship of E then an instance of **R** would be a set $e_1, e_2, \ldots, e_n$ where any e_i is of type E. Graphically, there is an arrow from the primitive relationship diamond **R** to the entity type rectangle E. If the outgoing arrow from **R** is labelled with "M" then the relationship is a normal set. If the arrow is labelled with "MO" then the relationship is an ordered set of entities (a list) and the entity set is treated as a multi-set. Figure 3.11 shows the two types of primitive relationships.

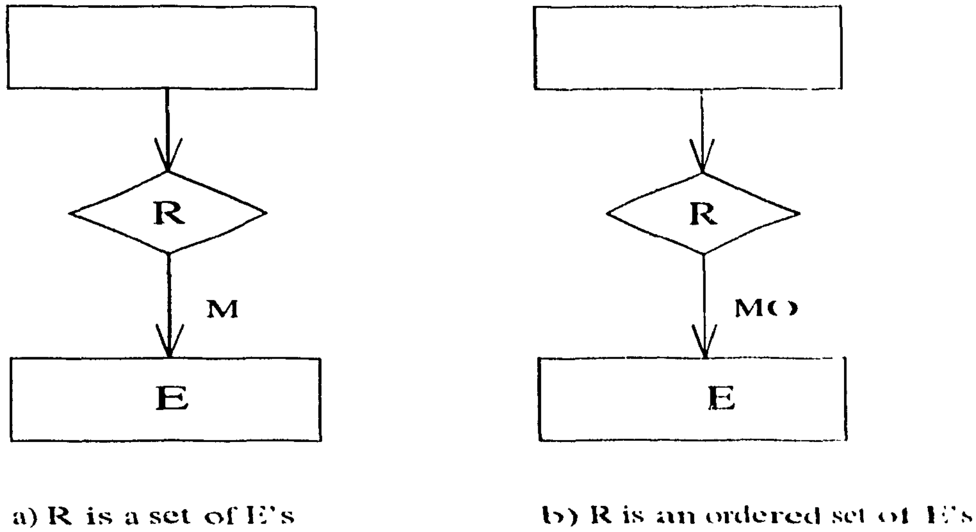


Figure 3.11: Two types of primitives relationships

SECOND-ORDER A **second-order** relationship **R** defined between two entity types E_1 and E_2 is a set of 2-tuples (e_1, e_2) where e_1 is an entity of type E_1 and e_2 is an entity of type E_2; or is a set of 2-tuples (e_1, s) where e_1 is an entity of type E_1 and s is a set of entities of type E_2. Graphically, there is an arrow from the second-order relationship **R** to each entity sets E_1 and E_2. One arrow is labelled "1" (points to the first entity type in the tuple) and the other is labelled "2", or "M" or "MO" (points to the second entity type in the tuple, whether it is a single entity or a set or a multi set). Figure 3.12 shows two possible second-order relationships.

HIGH-ORDER A **high-order** relationship **R** is defined as a set of 2 tuples (e, r) where e is an entity of the entity set E and r is an instance of a relationship **R'**. Then, **R'** must be a second-order or a high-order relationship. Figure 3.13 shows a high-order relationship. We have seen previously that a relationship in the E-R approach is a set of n-tuples $(e_1, e_2, e_3, \ldots, e_n)$. For any relationship of this form, there is an equivalent relationship in the MER approach: the n entity sets connected to the relationship diamond is transformed in the MER model into n-2 **high-order** relationships and *1* **second-order** relationship as

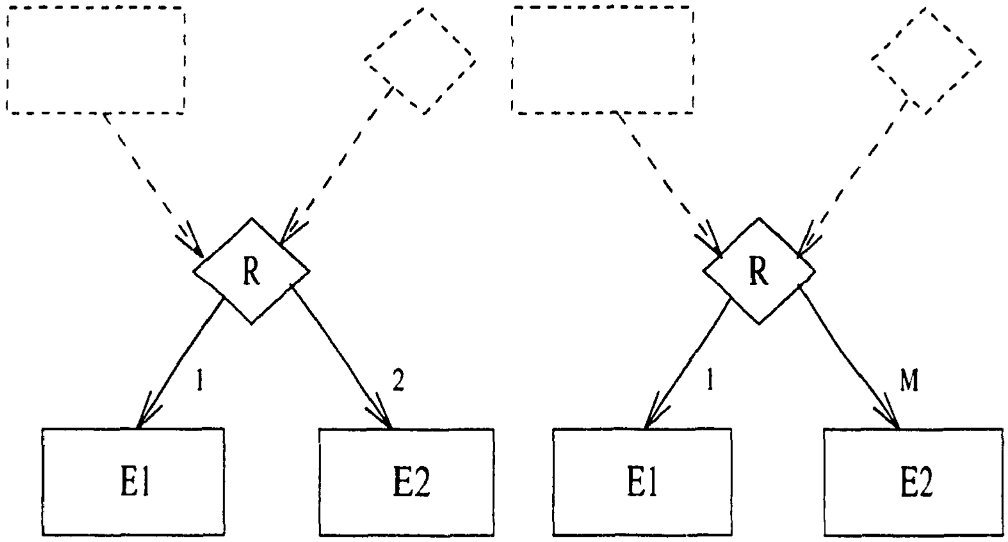


Figure 3.12: Second-order relationships

shown in figure 3.11.

We define the **depth** of a relationship hierarchy as being the number of diamonds starting from the top most high-order relationship to the bottom most second-order relationship. In our previous example, the **depth** of the relationship hierarchy is *n-1*.

3.2.5 Functionality of Relationships in the MER Diagram

One-to-One To represent a one-to-one relationship **R** between two entity types, the outgoing arrows of **R** are labelled with "1" and "2". The label indicates the order of the entity types in the 2-tuple. Note that only second-order relationships can be one-to-one.

One-to-Many To represent a one-to-many relationship **R**, the two outgoing arrows of the diamond of **R** are labelled with "1" and "M" or "MO". If the edge labelled with "1" points to entity set E_1 and the edge labelled with "M" points to entity set E_2, then one entity in E_1 can be associated with zero or more entities in E_2. This produces an association between an entity and a set of entities ("MO"

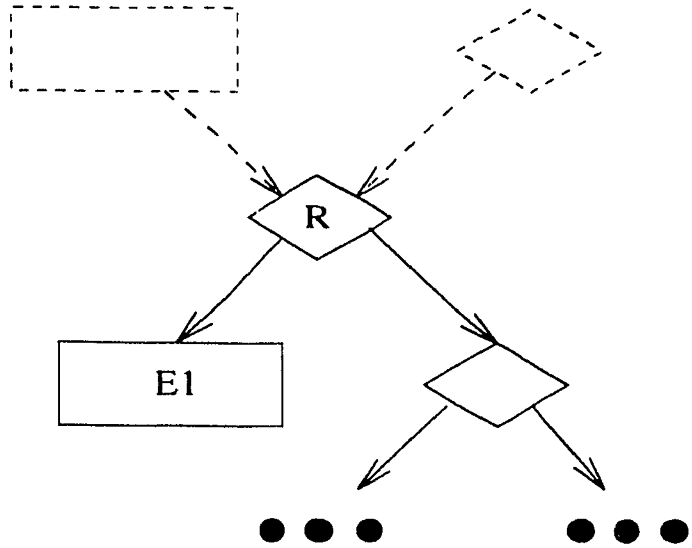
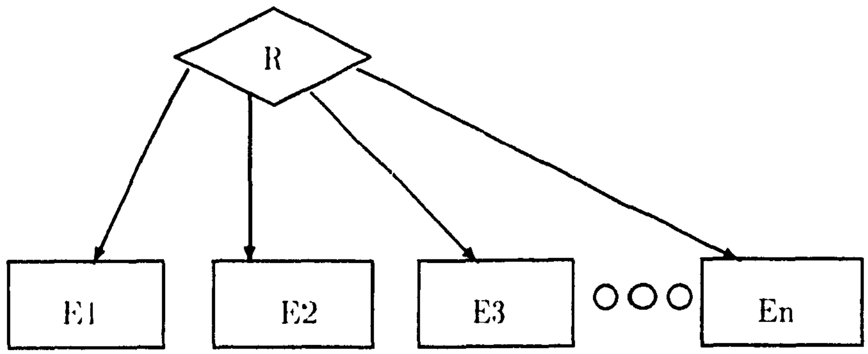


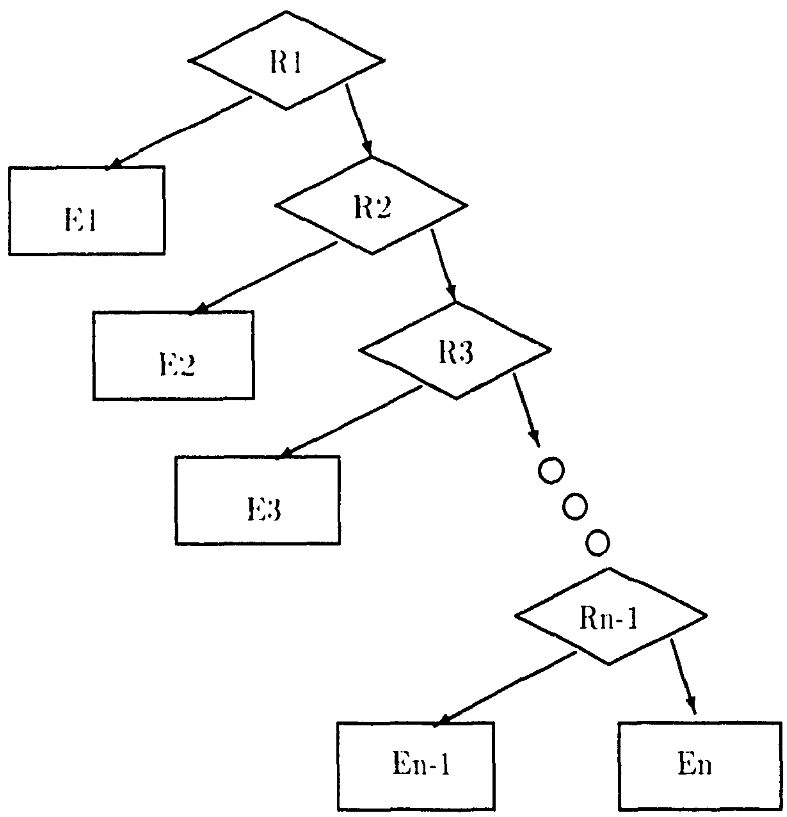
Figure 3.13: A high-order relationship

refers to many-ordered entities, in which case we have an association between an entity and an ordered multi-set). Second-order relationships can be one-to-many. High-order relationships are *always* one-to-many since they *always* associate an entity to a set. For that reason, there is no need to label the arrow edge "1" and "M" in this last case.

Many-to-Many A many-to-many relationship between entity sets E_1 and E_2 is represented differently, depending on the requirements of the system being modeled. In the real world, this type of relationship means that each entity in E_1 can be associated with zero or more entities in E_2 and vice versa. However, according to the requirements of the system, it might not be necessary to know both facts even if a many-to-many relationship exists in reality. While modeling with the MER, the two way correspondences of a many-to-many relationship are to be stated explicitly. This will be captured by two one-to-many relationships as shown in figure 3.15. Therefore, for a many-to-many relationship **R**, the higher order abstract entity of type **E** will possess two properties: one relationship R_1 mapping a set of entities in E_2 to one entity in E_1 and a relationship R_2 mapping a set of entities in E_1 to one entity in E_2.



E-R model



MER model

Figure 3.14: Relationship in the E-R model and its MER equivalent

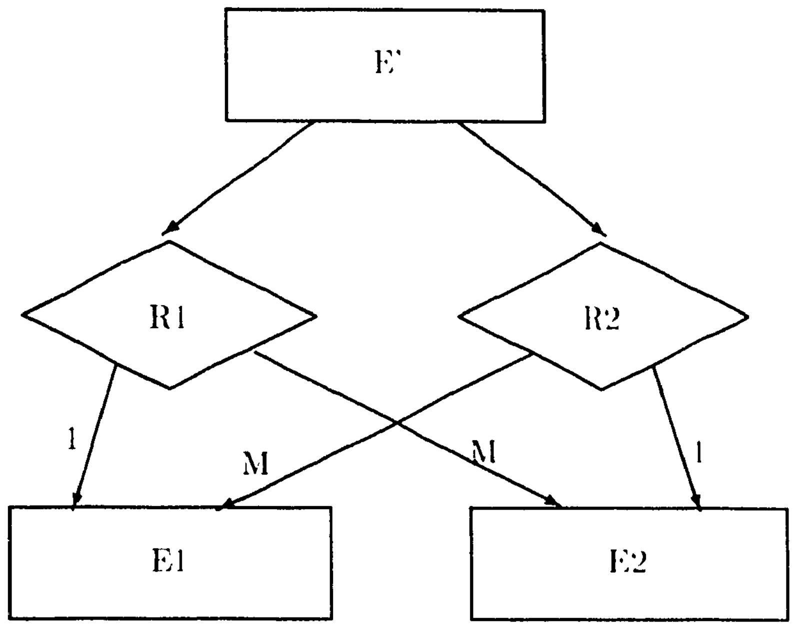


Figure 3.15: Many-to-many relationship with both functionalities required

3.2.6 Inheritance

The MER approach provides the concept of inheritance through the **Is-a** relationship, as defined in the original E-R model. If **A Is-a B** then **A** will inherit all the attributes and/or relationships defined in **B** in addition to its own attributes and/or relationships. The **Is-a** relationship can be used to form a hierarchy of entities where any entity at any level inherits the properties of the higher level entities. This concept of inheritance is particularly useful to gain properties from predefined entity types. In brief, an entity type can be either user-defined or predefined in which case, the definition of the predefined entity type is known to the system and made available to the user to directly include it as an entity type in the MER model or to create more detailed entity types inheriting general characteristics from the predefined entity type.

3.2.7 Attribute Types

The MER model presented here defines entity types. To complete this model, the user needs to provide a data dictionary which will describe the types of all the attributes and single-value entity sets. As in the E-R approach, a number of built-in simple types exist. These are *Natural*, *Integer*, *Text* and *Boolean*. For each attribute and single-value entity, the user must specify its type using any of the built-in types, predefined entity types or by another user-defined entity type of the MER model. For example, for the entity type PERSON of figure 3.4, a type should be provided for attributes **Name**, **Social_ins_nb** and **Age**. The possible types for these attributes are *Text*, *Natural* and *Natural*, respectively.

Attribute types of the E-R model can only be of simple built-in types. However, it is possible in the MER model that an attribute of an entity type be defined by another entity type, as long as there is no recursivity in the definition of the two types. This means the following cases are not allowed: 1) entity type E_1 has an attribute of type E_2 and entity type E_2 has an attribute of type E_1, 2) entity type E_1 has an attribute of type E_2 and E_2 is a higher order abstract entity type which is defined by a relationship (of any type) involving E_1. The process of developing the data dictionary is interactive in nature. It will be explained fully in section 5.3 when describing the graph tool used to build the MER. For now, it is enough to know that the user must provide a valid type for each attribute.

3.3 Defining the System Entities

The MER model captures the different classes of objects of interest to the analyst. In that sense, the MER model creates categories of objects defined by certain properties. They are equivalent to types. The entities which compose the system are to be declared by the user. These entities are instances of the entity types defined in the MER model. The user is required to provide the entity names and their corresponding types. These entities, which together define the system, will be referred to as the *system entities* or the *global entities*. They are global because they are known within the scope of any system activity that must be provided. The concept of global and local entities are similar to the global and local variables in VDM. They will be further explained in chapter 4 when the description for operations will be introduced

Chapter 4

Keyword-based Formatted Description for Operations

While modeling a system, we need to describe data models and operations of the system to develop. These operations are described in terms of manipulations of the system entities. For the purpose of describing these operations, we use a **K**eyword-based **F**ormatted **D**escription language [D'Almeida'92]. A KFD has a name and follows a specific format. A set of KFDs form the functional requirements. The format of a KFD is given below:

Operation name:	OPNAME
Operands:	$partype_1, partype_2, \ldots, partype_n$ [$\rightarrow$ **[M/MO]** $returntype$]
Syntax:	$param_1, \ldots, param_i, \ldots$ OPNAME, $\ldots, param_j, \ldots, param_n$
Constraints:	A list of constraints
Semantics:	A list of actions
Description:	Natural language description for human communication.

In the **operation name**, we give the name of the operation the KFD is describing. KFDs are uniquely identified by this name. The **Operands** clause gives a list of entity types which are the types of the operands for the operation. These types must be defined in the MER model or must be predefined entity types. Optionally, if the operation returns a value, the type of the value returned is specified by the *returntype*, after the right arrow. A set of values or an ordered set of values can also be returned when "M" or "MO" is specified. In the **Syntax** clause, we give the list of input parameters for the operation. The order of the parameters and the operation name is important since it settles the exact format to invoke the operation. A $param_i$ in the **Syntax** part is an instance variable of entity type $partype_i$ in the **Operands** section. Then, we specify the constraints which are boolean conditions that must be satisfied in order to be able to execute that operation. A set of keywords exists to specify conditions on the parameters of the operation or the global entities. Similarly, the **Semantics** clause describes the meaning of the operation as a set of actions to be performed on the parameters or global entities. Action keywords performing changes on entities are available for that purpose.

The only entities (instance variables) known in the context of an operation are the input parameters, the entities created by the operation itself and the user defined system entities provided with the MER. The input parameters and the entities created by the operation are local to that operation. The system entities defined by the user as being the system's main components are global to every operation. An instance variable can be created in the context of an operation using the keyword **create**. The keywords used to in the **Constraints** and **Semantics** clauses have their fixed syntax and semantics. They are explained in the following sections. The operation is executed only if all the constraints are satisfied. When the operation is not executed, the states of the entities remain unchanged and the operation has no effect.

4.1 Syntax for Constraints and Actions

This section defines constraints and actions. We define a term by means of other terms and/or keywords. This process of defining terms is done recursively until we reach a level at which the term can be described with keywords only or with concepts which are assumed to be defined in the MER model (entity types, relationship sets or attributes). The convention assumed in this thesis, views the definition of a term as an equation. On the L.H.S of the equation, we specify the term name. One the R.H.S of the equation, the definition for that term is given. In addition, the following conventions are assumed in defining a term:

- Anything within square brackets [ ] is optional.
- ⟨a-term-name⟩ in the definition part.
- Anything within braces { } can be repeated 0 or more times.
- Anything separated by a vertical bar | is one possible definition for the term. Thus, if **term** $= def_1 \mid def_2 \mid \ldots \mid def_n$ then **term** is defined as def_1 or def_2 or ... def_n.
- **Bold** words in the definition of a term (R.H.S) are keywords.

4.1.1 Definition of Terms

Constraints = ⟨single-constraint⟩ [⟨logical-op⟩ ⟨constraints⟩]

Actions = ⟨single-action⟩ [**and** ⟨actions⟩]

Single-constraint = ⟨simple-constraint⟩ | ⟨conditional-constraint⟩ | ⟨universal-constraint⟩

Logical-op = **and** | **or**

Single-action = ⟨create-action⟩ | ⟨add-action⟩ | ⟨remove-action⟩ | ⟨set-action⟩ | ⟨return-action⟩ | ⟨conditional-action⟩ | ⟨universal-action⟩

Simple-constraint = ⟨simple-obj⟩⟨operator⟩⟨simple-value⟩ | ⟨obj⟩ **is** [**not**] ⟨obj-value⟩ |
⟨obj-value⟩ [**not**] **is-in** ⟨group⟩

Conditional-constraint = **if** ⟨conditions⟩ **then** ⟨constraints⟩ [**else** ⟨constraints⟩] **end-if**

Universal-constraint = **for-all** ⟨ent⟩ **in** ⟨group⟩ **do** ⟨constraints⟩ **end-for**

Create-action = **create** ⟨ent⟩ : ⟨ent-type⟩ **with** ⟨obj-value⟩ {, ⟨obj-value⟩}

Add-action = **add** ⟨obj⟩ | (⟨ent⟩, ⟨ent⟩) **in** ⟨group⟩

Remove-action = **remove** ⟨obj⟩ **from** ⟨group⟩

Set-action = **set** ⟨obj⟩ **to** ⟨obj-value⟩

Return-action = **return** ⟨ent⟩ [**such-that** ⟨constraints⟩]

Conditional-action = **if** ⟨conditions⟩ **then** ⟨actions⟩ [**else** ⟨actions⟩] **end-if**

Universal-action = **for-all** ⟨ent⟩ **in** ⟨group⟩ **do** ⟨actions⟩ **end-for**

Simple-obj = ⟨single-ent⟩ | ⟨simple-attr⟩ | **number-of** ⟨group⟩ |
first-of ⟨simple-group-list⟩

Simple-value = **empty** | ⟨num-constant⟩ | ⟨text-constant⟩ | ⟨bool constant⟩ | ⟨simple-obj⟩

Operator = **is** [**not**] | [**not**] < | [**not**] >

Obj = ⟨ent⟩ | ⟨attribute⟩ | **first-of** ⟨group-list⟩ | ⟨group⟩ | ⟨simple obj⟩

Obj-value = ⟨simple-value⟩ | ⟨obj⟩

Conditions = ⟨simple-constraint⟩ | [⟨logical-op⟩ ⟨conditions⟩]

Ent = an instance variable name *ent* of some entity type *ENT* |
$rel(e_1)(e_2)...(e_n)$ **of** *ent*, refers to the second entity of a 2-tuple in the second order relationship set $rel(e_1)(e_2)...(e_{n-1})$ where *rel* is a second or high order relationship set defined in *ent*. The depth of the relationship hierarchy is *n - 1*.

Ent-type = An entity type name *ent-type* defined in the MER model | a built in type | a predefined entity type.

Single-ent = an instance variable name *ent* of some SINGLE VALUE entity type *ENT*
|
$rel(e_1)(e_2)...(e_n)$ **of** *ent*, refers to the second SINGLE-VALUE entity of a 2 tuple

in the second-order relationship set $rel(e_1)(e_2)...(e_{n-1})$ where rel is a second or high-order relationship set defined in ent. The depth of the relationship hierarchy is n - 1.

Simple-attr = $attr$ [**of** ⟨simple-attr⟩] **of** ent, where $attr$ is a SIMPLE TYPE attribute of some entity ent or $attr$ is a SIMPLE TYPE sub-attribute of another complex type attribute ⟨simple-attr⟩ of the entity ent.

Simple-group-list = same as ⟨group-list⟩ except that the relationship is a set of "MO" SINGLE-VALUE entities.

Num-constant = a numerical constant including the nil value.

Text-constant = a text constant including the empty string.

Bool-constant = a true or false value (nil when the variable is not defined).

Attribute = $attr$ [**of** ⟨attribute⟩] **of** ent. where $attr$ is an attribute of some entity ent or $attr$ is a sub-attribute of another complex type attribute ⟨attribute⟩ of the entity ent.

Group-list = rel **of** ent. refers to a primitive relationship set of "MO" entities in the entity ent |
$rel(e_1)(e_2)...(e_n)$ **of** ent. refers to a relationship set of "MO" entities where rel is a second or high-order relationship set defined in ent. The depth of the relationship hierarchy is n - 1.

Group = rel **of** ent. where rel is the relationship set (of any type) defined in the entity ent | $rel(e_1)(e_2)...(e_n)$ **of** ent, refers to a relationship set (of any type) where rel is a second or high-order relationship set defined in ent. The depth of the relationship hierarchy is **at least** n - 1.

The syntax given here is context-free. It is simple to derive the production rules of the grammar from the definition given above. At this point, we did not spend time to build a parser for this language because we prefer to emphasize on other more important parts of the work. However, we believe a parser can be easily created using a *Parser Generator* like YACC, given the appropriate context-free production rules.

4.2 List of Keywords

We describe in this section the list of possible keywords which can be used to access, test or change entity values. The semantic aspects of each keyword is given i.e. we explain their meaning and how they can be used. The parameters for the keywords are terms defined in section 4.1 where details about their syntax can be found. The keywords are classified into three basic categories. The keywords used to refer to any type of entity value are called **reference** keywords. The keywords used to test a condition and return a true or false value are called **test** keywords. Finally, the keywords changing an entity value are called **action** keywords.

4.2.1 Reference Keywords

Reference keywords are used to access entity values. This may be a single value, a relationship set or the complete entity record. It is also used to specify constant values. Reference keywords can be used in the **Constraints** and **Semantics** clauses of a KFD.

- **Empty**
  Refers to the empty set (for all types of relationship sets), the empty string (Text) or a nil value (Integer, Natural).

- **First-of** *group-list*
  Refers to the first entity of the specified ordered set (for primitive relationships of "MO" entities). *Group-list* follows the syntax definition given in section 4.1.

- **Number-of** *group*
  Refers to the number of entities in the specified group. In mathematical terms, it refers to the cardinality of the relationship set (for any type of relationship set). *Group* follows the syntax definition given in section 4.1.

- *comp* **of** *ent*

  Refers to the value of one the components of the specified entity. *Comp* can be an attribute name of the entity *ent* or it may refer to the name of a relationship set defined in *ent*. For a relationship r defined in *ent* where r is a set of tuples (e_1, e_2) or a set of tuples (e_1, r_1), we can also refer to the entity e_2 or the set r_1 associated to an entity e_1 by simply using the function notation $r(e_1)$ **of** *ent*. If r_1 is a second order or a high-order relationship, we can go down the hierarchy of relationships with $r(e_1)(e_i)$... **of** *ent* and so on. Refer to **Attribute**, **Ent** and **Rel-set** in section 4.1 for more detail about the use of keyword **of**.

4.2.2 Test Keywords

Test keywords are used to test some condition and return true or false. The keywords in brackets are optional. Test keywords can be used in the **Constraints** clause of a KFD. It can also be used in the **Semantics** clause but only when specifying the conditions of a *conditional* action.

- *obj* **is** [**not**] *obj-value*

  Returns true if the value of the first object is [not] equal to the value of the second object, false otherwise. *obj* and *obj* must be of the same type. *obj* and *obj-value* follow the syntax definition given in section 4.1.

- obj_1 [**not**] $<$ obj_2 or obj_1 [**not**] $>$ obj_2

  Returns true if the value of the first object is [not] smaller/greater than the value of the second object, false otherwise. obj_1 and obj_2 must be of the same type. obj_1 and obj_2 follow the syntax definition of **simple-obj** given in section 4.1.

- *obj* [**not**] **is-in** *group*

  Returns true if the object is [not] part of the specified group. The group is a relationship set defined in some higher order abstract entity. There are three possibilities in which the **is-in** can be used: 1) *obj* refers to an entity and *group* specifies a set of entities of the same type as *obj*, 2) *obj* is an entity of type T_1

and *group* specifies a set of 2-tuples where the type of the domain element (first entity in the tuple) is T_1. 3) *obj* is an entity of type T_1 and *group* specifies a set of entities of another type, say T_2, and there exists an attribute defined in T_2 which is of type T_1. In the last case, *obj* **is-in** *group* means that there exist an entity in the specified *group* having an attribute with the same value as *obj*. *obj* and *group* follow the syntax definition given in section 4.1.

- **if** $conditions_1$ **then** $conditions_2$ [**else** $conditions_3$] **endif**

  When the **else** part is not specified, returns true if the $conditions_1$ is true and the $conditions_2$ is true; or returns true if the $conditions_1$ is false; otherwise returns false. When the **else** part is specified, returns true if the $conditions_1$ is true and the $conditions_2$ is true; or returns true if the $conditions_1$ is false and the $conditions_3$ is true; otherwise returns false. The $conditions_i$ follow the syntax definition given in section 4.1.

- **for-all** *ent* **in** *group* **do** *constraints* **end-for**

  Perform the *constraints* for each entity in the specified *group*. If the group specifies a set of entities e_i or a set of tuples (e_j, e_k), the constraints must be satisfied (if applicable) for each e_i in the set or for each tuple in the set. Only when they are satisfied for all of them can the *for-all* constraint return true. Otherwise, returns false. *Ent*, *group* and *constraints* follow the syntax definition given in section 4.1.

4.2.3 Action Keywords

Actions keywords are used in the **Semantics** section only. We distinguish several basic actions that can be done in an operation: create a new instance of an entity, add/remove an entity in/from a relationship set, or change an attribute value of an entity. Any combination of these basic actions can be applied recursively on one or more entities as the goal of the operation requires.

- **create** *ent : entype* **with** $val_1, val_2, \ldots, val_n$

  Creates a new instance variable *ent* of type *entype* and assigns val_i to the i^{th}

attribute/relationship defined in *entype*. The type of val_i must correspond to the type of the i^{th} attribute/relationship. *Entype* is a record of n fields (attributes or relationship sets). The new entity *ent* becomes known in the rest of the context of the operation. The syntax of *ent* and *entype* is given in section 4.1. Each val_i refers to some object values and follows the syntax definition of **obj-value**.

- **add** *obj* **in** *group*

  Adds an object to a group of objects of the same type. The *group* must refer to a primitive or second order relationship. If the group specifies a set of entities of type E_1 then *obj* must be an entity of type E_1 or a set of entities of type E_1. If the group refers to a second order relationship (a set of 2-tuple entities of type E_1 and E_2) then *obj* must be a tuple (ϵ_1, ϵ_2) or a set of tuples $(\epsilon_{1_i}, \epsilon_{2_j})$ where ϵ_{1_i} is of type E_1 and ϵ_{2_j} is of type E_2. *Obj* and *group* follow the syntax definitions given in section 4.1. The addition process is interpreted differently depending on the type of relationship the group specifies. The following are the rules guiding the **add** action:

1) If the group is a primitive relationship of "M" entities of type E then *group* will be the union of *group* and *obj* whether *obj* is an entity of E or a primitive relationship set of entities of type E.

2) If the group is a primitive relationship of "MO" entities of type E then *obj* will be appended to *group* whether *obj* is an entity of type E or a primitive relationship set of entities of type E.

3) If the group is a second order relationship r then *obj* must be a tuple (ϵ_1, ϵ_2) or a set of tuples $(\epsilon_{1_i}, \epsilon_{2_j})$ that will be added in the set of tuples specified by the group.

3.1) If r is one-to-one then

Add every $(\epsilon_{1k}, \epsilon_{2k})$ in the group.

If (e_{1k}, x) already exists in the group then

it will be replaced by the new tuple (e_{1k}, e_{2k}).

3.2) If r is one-to-many then

Every entity e_{1k} is associated with a set s_k. Tuple (e_{1k}, s_k) in the relationship set will be changed for $(e_{1k}, s_k \cup \{e_{2k}\})$.

If (e_{1k}, s_k) did not exist before then

tuple $(e_{1k}, \{e_{2k}\})$ will be created in the relationship set.

Note: **add** e_2 **in** $r(e_1)$ or **add** s' **in** $r(e_1)$

are other equivalent ways to add an entity or a set of entities to the set s associated to e_1. This will produce $(e_1, s \cup \{e_2\})$ or $(e_1, s \cup s')$. However in this case, entity e_1 must exist in the domain of the relationship r (tuple (e_1, x) must exist in the set whatever x is).

4) If the group is a high-order relationship then

the action **add** is not valid for high-order relationships. For a high order relationship r, we must go down to the hierarchy of relationships with $r(e_1)$, (e_2),... until we reach a second order relationship where a 2-tuple can be added.

- **remove** *obj* **from** *group*

Removes the entity object *obj* from the *group* of entities of some type T. *obj* may be an entity of type T or a set of entities of type T. *obj* and *group* follow the syntax definition given in section 4.1. If any entity in *obj* does not exist in the group, nothing is done.

1) If the group is a primitive relationship of "M" or "MO" entities then

every entity specified in *obj* is removed from the *group* set.

2) If the group is a second order relationship r then

2.1) If r is one-to-one then

If *obj* is an entity e then

tuple $(e,\ x)$ is removed from r whatever x is.

If *obj* is a set of entities e_i then

every (e_i, x) existing in r is removed.

2.2) If r is one-to-many then

If *obj* is an entity e then

the tuple (e, s) (where s is a set of entities) is removed from r.

If *obj* is a set of entities e_i then

every (e_i, s) (where s is a set) is removed from r.

Note: we can remove a particular entity e' from a set s associated to e in relationship r with **remove** e' **from** $r(e)$.

- **set** obj_1 **to** obj_2

  Assigns the value of obj_2 to obj_1 and they must be of the same type. They follow the syntax definition of **obj** in section 4.1.

- **return** *ent* [**such-that** *conditions*]

  Returns an entity called *ent* to the system external environment. If *ent* is known in the scope of the operation, there is no *conditions* specified. If the instance variable *ent* is not known, *ent* is created and becomes known in the rest of the operation description. The entity *ent* is of type *returntype* where *returntype* is the type specified in the **Operands** section of the KFD. When a new entity *ent* is created, it must satisfy the *conditions* otherwise the entity returned is undefined. The keyword **return** is only valid for operations returning an object.

i.e. having an arrow $\rightarrow$ in the **Operands** section.

- **if** *conditions* **then** $actions_i$ [**else** $actions_j$] **endif**

  Performs $actions_i$ if the conditions are true otherwise nothing is done unless an **else** part exists in which case, $actions_j$ will be performed. The *conditions* and *actions* follow the syntax definition given in section 4.1.

- **for-all** *ent* **in** *group* **do** *actions* **end-for**

  Perform the *actions* on each entity in the specified *group*. If the group specifies a set of entities e_i or a set of tuples (e_j, e_k), the actions are applied (if applicable) to each e_i in the set or to each tuple in the set. *Ent*, *group* and *actions* follow the syntax definition given in section 4.1.

4.3 KFD Limitations

The format of the KFD is a first cut proposal and is by no means final. The syntax and semantics of the KFD can go through further refinements, enhancements and modifications. The set of keywords available now are general in nature. They view everything as being objects or group of objects, whatever types these objects are. For that reason, the constraints or actions might not be written in the most concise way. In certain cases, using more specialized keywords could result in a better operation description. By having keywords carrying different meanings depending on the type of entities they are applied to, we only need to keep a small set of keywords. This way, it is easier for the user to learn the specification language.

No analysis has been made about how convenient the KFDs are to capture functional requirements. We applied the KFD technique on two different examples and we realized that due to the generality of the KFDs, they are easily adaptable to any type of problem. However, for large problems, the set of keywords might not be appropriate since they are basically primitive operations on sets of entities. The description could become too long and tedious to write with primitive keywords only. One way to overcome this problem would be to rely heavily on high-order keywords provided by the domain expert. However, the high-order keywords are domain dependent and would require to be changed each time an application in a different domain is given.

At this stage, we recognize that more experience in the use of KFDs in different domains is required to further refine and improve the language, and possibly create a more flexible format and syntax than what we have proposed in this work. However, the role of KFDs will remain the same.

Chapter 5

Detailed Description of the System

This chapter describes in detail the complete system presented section 1.2. Each component of the system will be described regardless of whether it has been implemented or not, at this point. The user interface of each component will also be presented whenever the component has to deal with the user. The subsystems described here covers the complete process from the writing of the semi formal system description to the actual generation of VDM specifications.

The analyst and/or user is responsible for providing the MER and KFDs which form the system description. Two distinct editors are provided to help the user to interactively develop the semi-formal specification. The MER is built using a graph editor and the KFD is developed using a text editor. The implementation of both editors is not completed at this stage but is under development. The descriptions of both editors given here should be considered as informal requirements of these subsystems. The description preprocessor validates the input description and generates an intermediate form of the specification. This subsystem is not implemented presently but the process has been simulated manually. The details about the description preprocessor includes the specification of the intermediate form. A prototype of the transformation process has been implemented. This subsystem is described by explaining the algorithms used to generate the VDM specification. Each subsystem description given in this chapter is not intended to force any design or implementation issues but rather to explain what are the required functions of the various subsystems.

Throughout this chapter, we will make use of the mailing system example taken from [Cohen'85]. We will show a possible semi-formal description derived from the given informal requirements of the mailing system a· now a VDM specification will be generated from it. The next section presents the informal requirements of the mailing system.

5.1 The Mailing System Example

The mailing system described in [Cohen'85] is part of an electronic 'office' which allows users to prepare and file documents and to transmit them to each other. In this simplified example, each user is provided with a personal terminal which serves as an electronic 'desk'. Hence, the complete system can be regarded as a network of desks, where each desk can send and receive documents to and from every other desk. This forms a ring as illustrated in figure 5.1. Each user's desk is conceptually divided into a number of work areas, as indicated in figure 5.2. At the center of the desk is a 'pad' where documents are placed and can be edited (created or changed). Documents are transferred between the pad and the other components of the desk. For example, a user can read or write mail messages using the pad which gets a message from or puts a message in the appropriate trays (queues) of the mailing system. The file subsystem is responsible for holding documents on a long-term basis and the printer subsystem produces a hard copy of the documents in the appropriate printer. It is not required to cover here the complete electronic office system. We will restrict ourselves to the mailing subsystem which is sufficient to illustrate the work presented in this thesis.

The electronic mail system uses three trays for each user as shown in figure 5.2. The out-tray contains documents waiting to be sent (moved from the pad). The in tray contains incoming documents transmitted by other users. The pending tray contains copies of those incoming documents for which replies are awaited. For each mail item, a header will be generated when passing the item from the pad to the

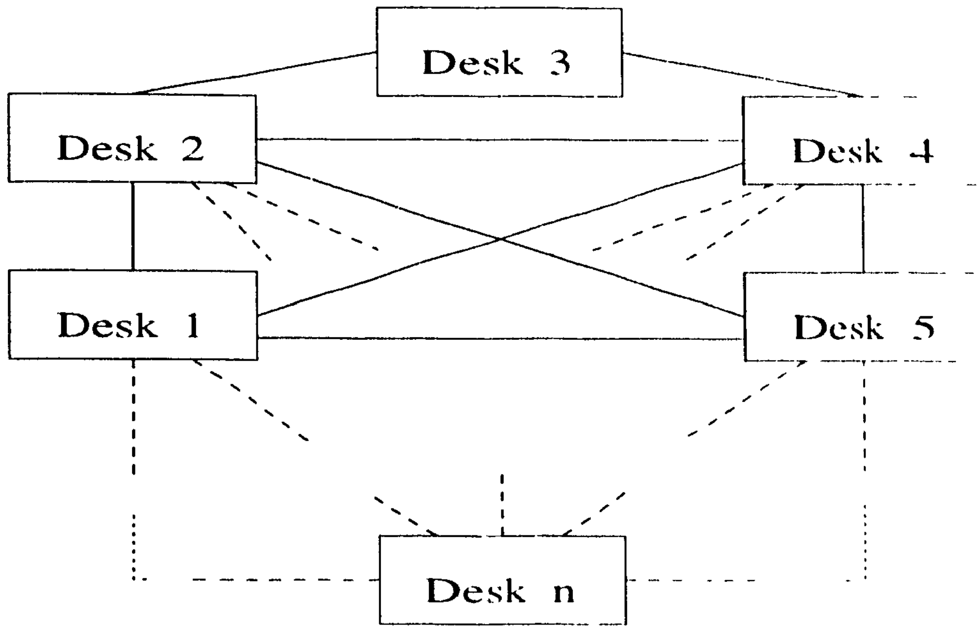


Figure 5.1: The electronic office system

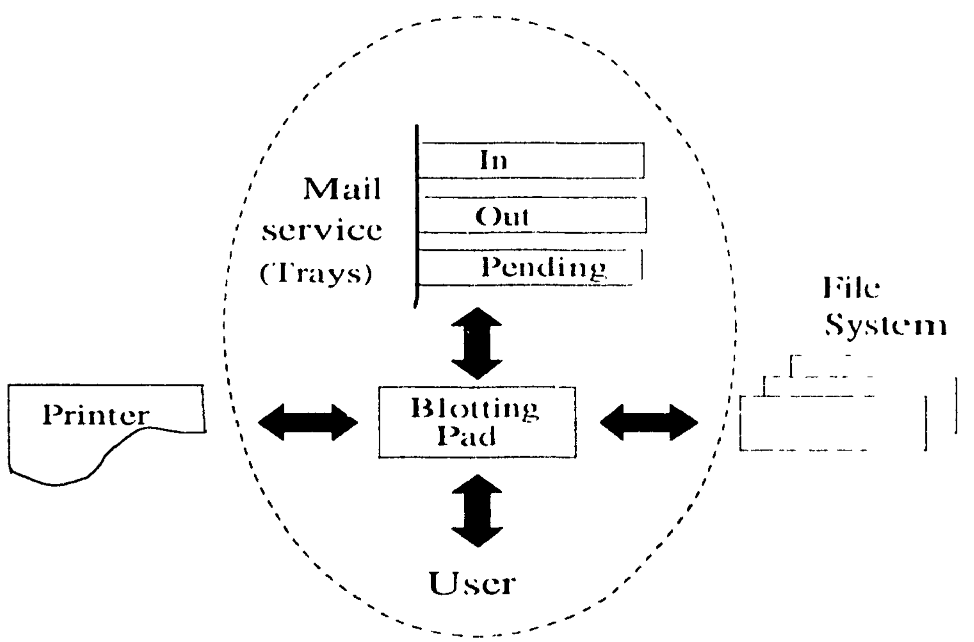


Figure 5.2: The components of a user's desk

out-tray. The user's pad is considered as the input/output device where the user reads/writes a mail. The header of the mail should contain the following information:

- a list of names of the recipients of the document.
- a list of names of persons to receive a copy of the document.
- the name of the person sending the mail item.
- the subject of the mail message.
- a flag indicating whether a reply is required.
- a possible reference to one or more pending items, for which this mail item is a reply.
- the time and date at which the mail item was transmitted
- a system-generated reference number for the mail

The above fields of a mail item are provided by the user who writes the mail (sending to the out tray), except the last two, which are generated by the system when the mail is sent and placed in the in-tray of the recipients. Additionally, when a reply is required, the mail system will also place the mail item in the pending tray of the recipients. It will be removed from the pending tray only when a reply is sent. The first two lists must preserve their order so that the recipients receive the mail in the specified order. Aliases can be used in these two lists of names. Therefore, each user maintain a directory which maps aliases to standard names by which each user is known to the system. Aliases and standard names must be unique.

The following are the names and the outline descriptions of the basic operations permissible in the mailing system:

- POST transfers the mail item from the user's pad into the out-tray. The mail item consists of a document and partially completed mail system header.

- CLEAR empties the user's out-tray and copies the mail into the appropriate in-trays of the intended recipients, and into their pending trays if replies are required. Only the recipients on the TO list can be required to reply (see [Cohen'85]). If any of the transmitted items are replies to pending items, the pending items are automatically deleted.

- COLLECT transfers a mail item identified by reference number from a user's in-tray to his pad. This deletes the item from the in-tray.

- READ copies a mail item identified by reference number from a user's pending tray to his pad. This does not delete the mail item, since pending items are deleted only when replies are sent.

- ADDALIAS adds a new name to the user's directory of aliases.

- DELALIAS deletes a name from the user's directory.

The details of the configuration specifying how each desk is connected to others need not to be taken care at the mailing subsystem level but should be considered for the complete electronic office system. From the above informal description, we need to identify the system entities and the relationships among entities in order to produce the MER model.

5.2 The MER Graph Editor

The current working version of the MER graph editor contains only the basic functions. The tool described here is more complete. This section explains fully what services the graph editor is expected to provide, even if they are not implemented at this point, and proposes an appropriate user interface. The proposed graph editor is shown in figure 5.3. The MER graph editor works under the XWindows environment. It is a very simple editor which allows the user to build the MER diagram by drawing boxes, ellipses, diamonds, triangles and lines using the mouse. The unshaded buttons are functions used to **create** any MER component. The shaded buttons are functions to **operate** on objects created in the diagram. The large area under the buttons is the allowable drawing space for the MER diagram. The virtual space is larger than the displayed window. The vertical and horizontal scrollbars permit the display window to move around the drawing space. The following describes the purpose of each button and its effect on the drawing area.

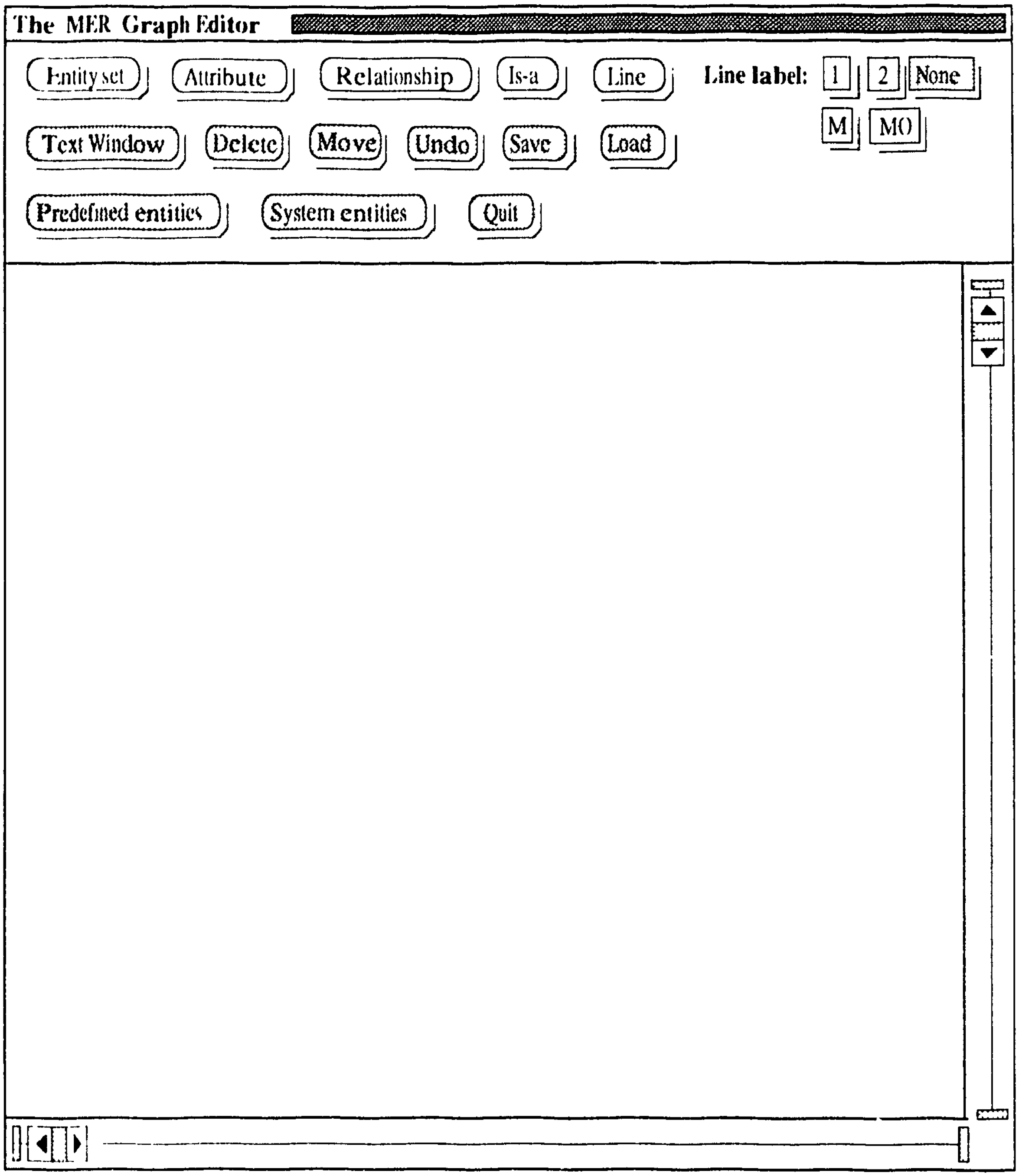


Figure 5.3: The graph editor for the MER diagram

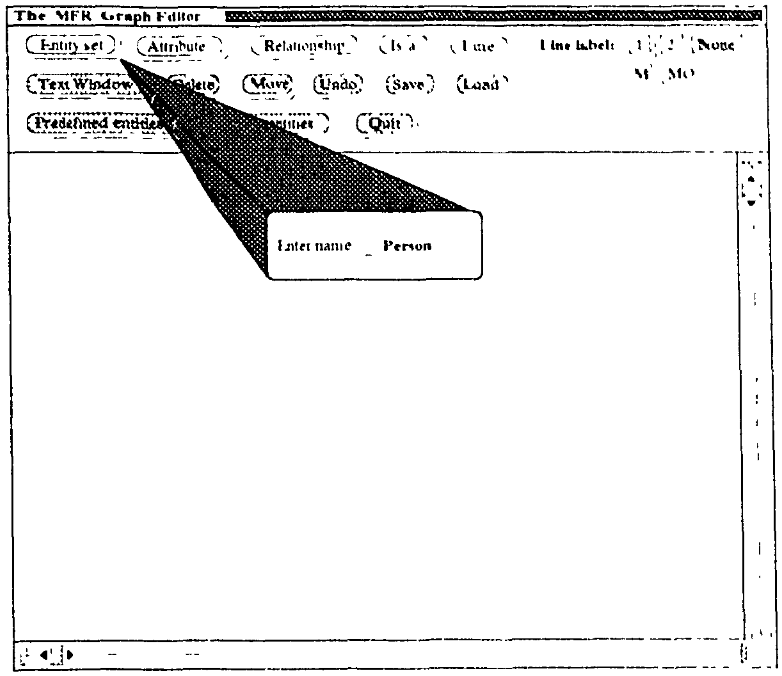


Figure 5.4: Creating an entity with the MER graph tool

Entity set By clicking on this button, a pop-up window is created requesting the user to provide a name for the entity set to create. This is shown in figure 5.4. As soon as a name is given, the pop up window disappears. Then the user moves the cursor in the drawing area, clicks on the desired location in the area and an entity set rectangle (labelled with the given name) is inserted in the diagram.

Attribute By clicking on this button, a pop-up window opens requesting the user to provide a name and a type for the entity. This is shown in figure 5.5. A value can be entered by clicking on the field and type the **Name**. In the case of the **Type** field, this will open a sub-window where all the valid types are listed in three groups, i.e. all the built-in types first, the predefined entity types in second and lastly the user-defined entity types created so far. The user simply selects the appropriate type with the use of the mouse. When both fields are given the pop-up window closes. Then, the user moves the cursor in the drawing area and clicks on the desired location where the attribute ellipse should be inserted

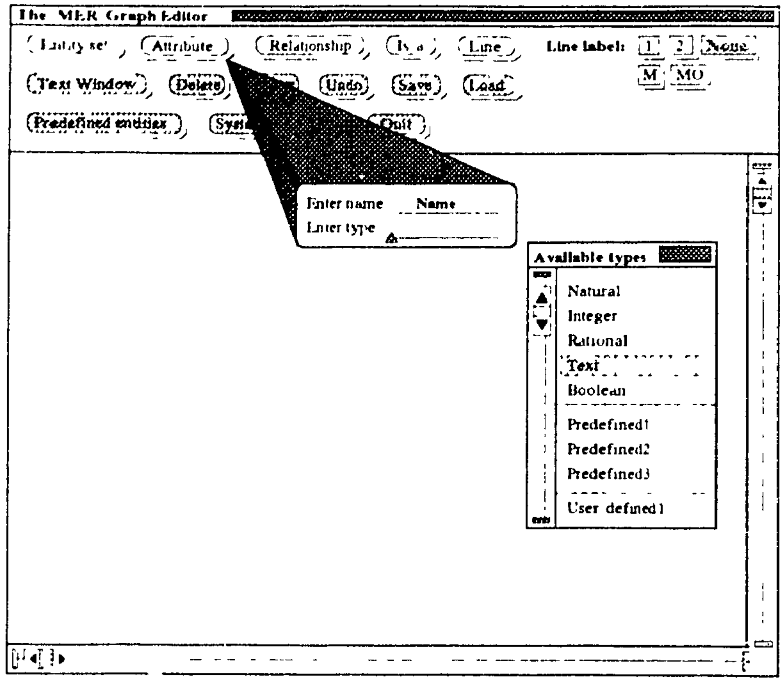


Figure 5.5: Creating an attribute with the MER graph tool

Attributes are shown with their types, as illustrated in figure 5.6.

Relationship The functionality of this button is the same as the "Entity set" button except that it inserts a relationship diamond in the diagram.

Is-a This button creates the Is-a relationship. As soon as the user presses this button, the program is waiting for the user to click in the drawing area to indicate where to insert the **Is-a** triangle.

Line This button draws a directed line in the diagram. The line must connect two MER components. The line possibly carries a label. The line label is the button currently pressed (shaded button), specified by the "Line label" field. The user clicks with the leftmost mouse button to indicate the points where segment lines are connected until the middle mouse button is pressed to indicate the last point to connect. The line ends at this point with an arrow. This function allows to draw a line only between an entity set and an attribute, an entity set and a relationship (including the **Is-a** relationship), or two relationships. The

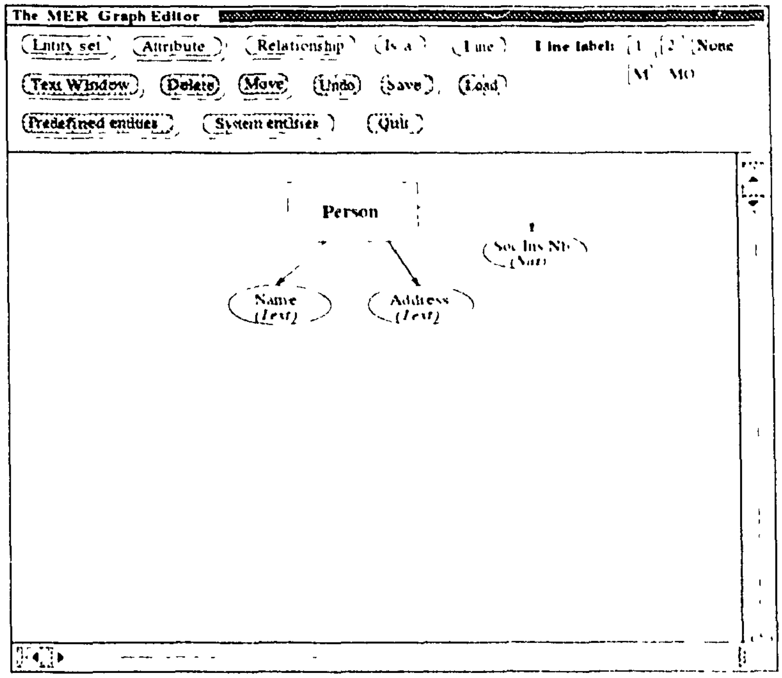


Figure 5.6: A MER diagram in the MER graph tool

other cases are not valid. A labelled line is allowed only between a relationship diamond and an entity set rectangle. An unlabelled line (Line label None) has no restrictions.

Line label This function assigns a label to a line of the diagram when the function "Line" is invoked. The user can select one of the five multiple choice label buttons with a mouse click. The choices are mutually exclusive. The default label in "None" in which case, the line function draws an unlabelled line.

Text window This function opens a text window to assign a textual description (a comment) to an object of the diagram. This description can be any type of information the user might need to know and can refer to while building the MER diagram or viewing it at a later stage.

Delete This function deletes any MER component created with the unshaded buttons. It is possible to delete an entity set rectangle, an attribute ellipse, a relationship diamond, the **Is-a** triangle, or a line. Upon clicking this button the user must select with the mouse the MER component to delete from the drawing space.

Move This function moves within the drawing area any component previously created with the first line of buttons. It is possible to move an entity set rectangle, an attribute ellipse, a relationship diamond, the **Is-a** triangle, or a line. Upon clicking this button, the user must select with the mouse the MER component to move within the drawing space.

Undo This function cancels the last effect created by any of the buttons described here.

Save This function saves the current MER diagram. Upon clicking this button, the user is requested to provide a file name under which the current information about the MER diagram will be saved.

Load This function loads an MER diagram into the drawing area. Upon clicking this button, the user is requested to provide a file name which includes the MER diagram to load.

Predefined entities This function allows the user to view and possibly use predefined entity types in the MER diagram. This function is fully explained in section 5.2.1.

System entities This functions permits to define the system global entities, as explained in section 3.3. Upon clicking this button, a sub-window will open where the user can enter the list of system entities and their corresponding types. This is shown in figure 5.7. When the list is completed, the "Ok" button will close the sub-window and accept the given input as the current system entities.

Quit This function quits the MER graph editor. If the last changes made to the MER diagram have not been saved, the user will have the possibility to save the changes before quitting or to quit immediately without saving (changes are cancelled, if any).

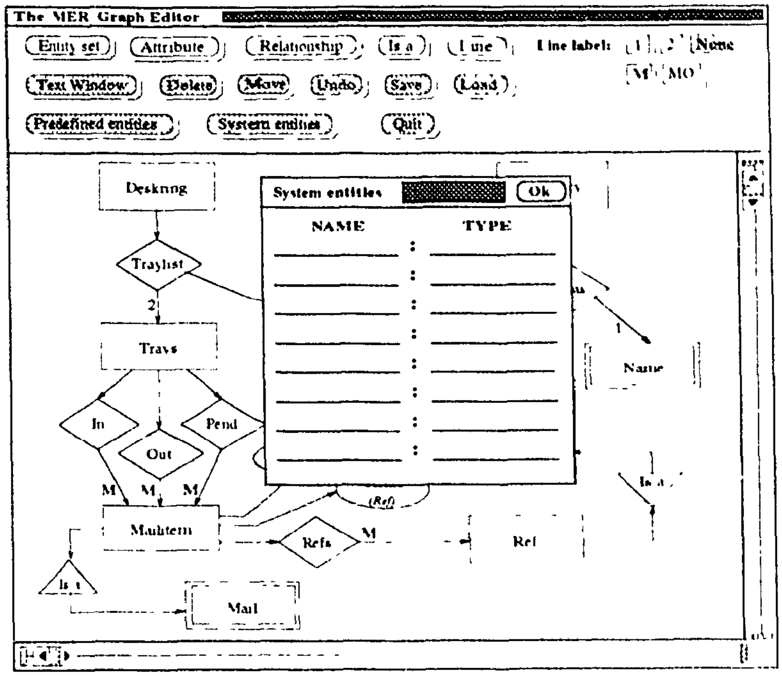


Figure 5.7: Defining system entities

5.2.1 Using Predefined Entities

Predefined entities are a priori known facts about the domain application. They are chunks of domain knowledge the user can use to describe the software system through the MER diagram. There are two ways in which predefined entity types can be used. One way is to directly include them in the MER model. The other way is to indirectly incorporate them through inheritance, that is, a new entity type will be created in the MER model and will inherit the properties of a predefined entity type in addition to its own properties. This is the concept of specialization discussed in section 3.1.1. Thus, a predefined entity type can only be a generalization of a user defined entity type. A predefined entity type cannot inherit properties from a user-defined entity type but it can inherit from other predefined entity types. The reason for this is that predefined entity types are stored information about the domain which cannot be changed or extended by other user-defined entity types. User-defined entities can inherit the properties of a predefined entity and not have additional properties of its own. This might be the case when we need to inherit the definition of a predefined entity type but give it another name.

To view the predefined entity types, the user clicks on the "Predefined entities" button. This will open a sub-window which will list all the names of the entity types already defined for that domain. The user can view any entity type properties (attributes and relationships) using the view function. This is illustrated in figure 5.8.

Another sub-window will display the MER model of the predefined entity type selected, showing its attributes and its relationships between other predefined entity types, if any. In the case of the mailing system, we will assume that entity types *Mail*, *Unique-entity* and *Name* are predefined and have the properties as shown in the second sub-window of figure 5.8. A mail is usually composed of the originator or author (*From*), a list of destinators (*To*), a list of people to whom a copy is forward (*Cc*), a subject, the date the mail was sent (*Whensent*), and the actual text message (*Body*). These are the minimum properties the domain expert who provided this knowledge assumed that a mail should possess. We can notice from the diagram that entity type *Mail* is defined by two primitive relationships with another predefined entity type (*Name*). *Name* is a single-value entity type which is simply defined as a string. Notice that all attributes of a predefined entity type and the single-value entity types are specified along with their types. The MER diagram of the predefined entity type *Unique-entity* is shown in figure 5.9. It has an attribute *Id* which is a unique identification number of the entity. The predefined entity type *Date* is a single-value entity which represents a system-generated date. The type of *Date* is machine-dependent. In this example, we will assume it is a *String*.

When the user has finished examining the characteristics of a predefined entity type, a click on the "Ok" button will make the second sub-window disappear and the control returns to the first sub-window. If the user needs to include a predefined entity type in the MER diagram, she must click on the "Use" button after having selected the desired predefined entity, then move the cursor in the drawing space of the MER diagram and click again where the predefined entity type rectangle should be inserted. Predefined entity types are represented by a double line rectangle. When a predefined entity type is inserted in the user-defined MER diagram, its properties

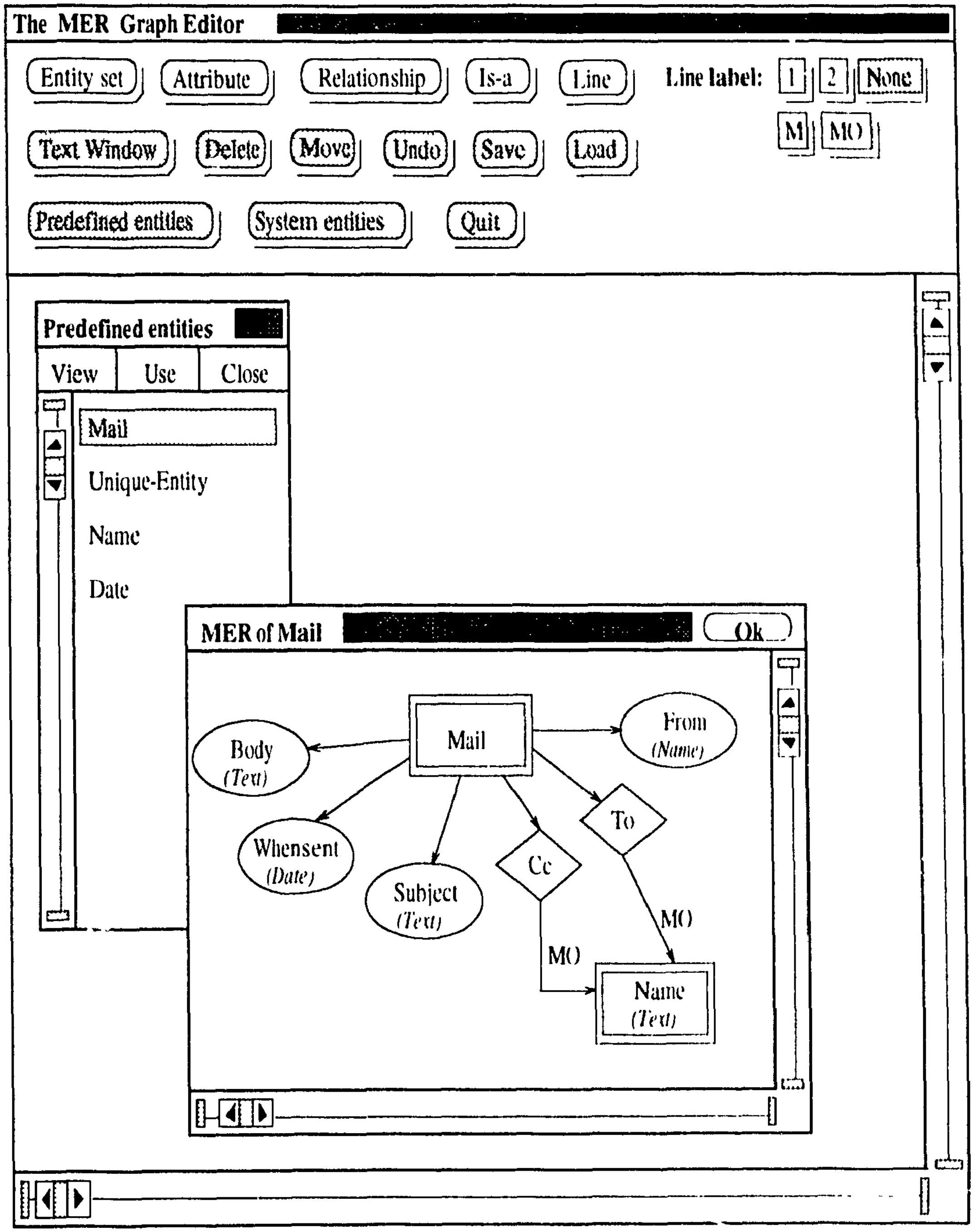


Figure 5.8: Viewing Predefined entity type *Mail*

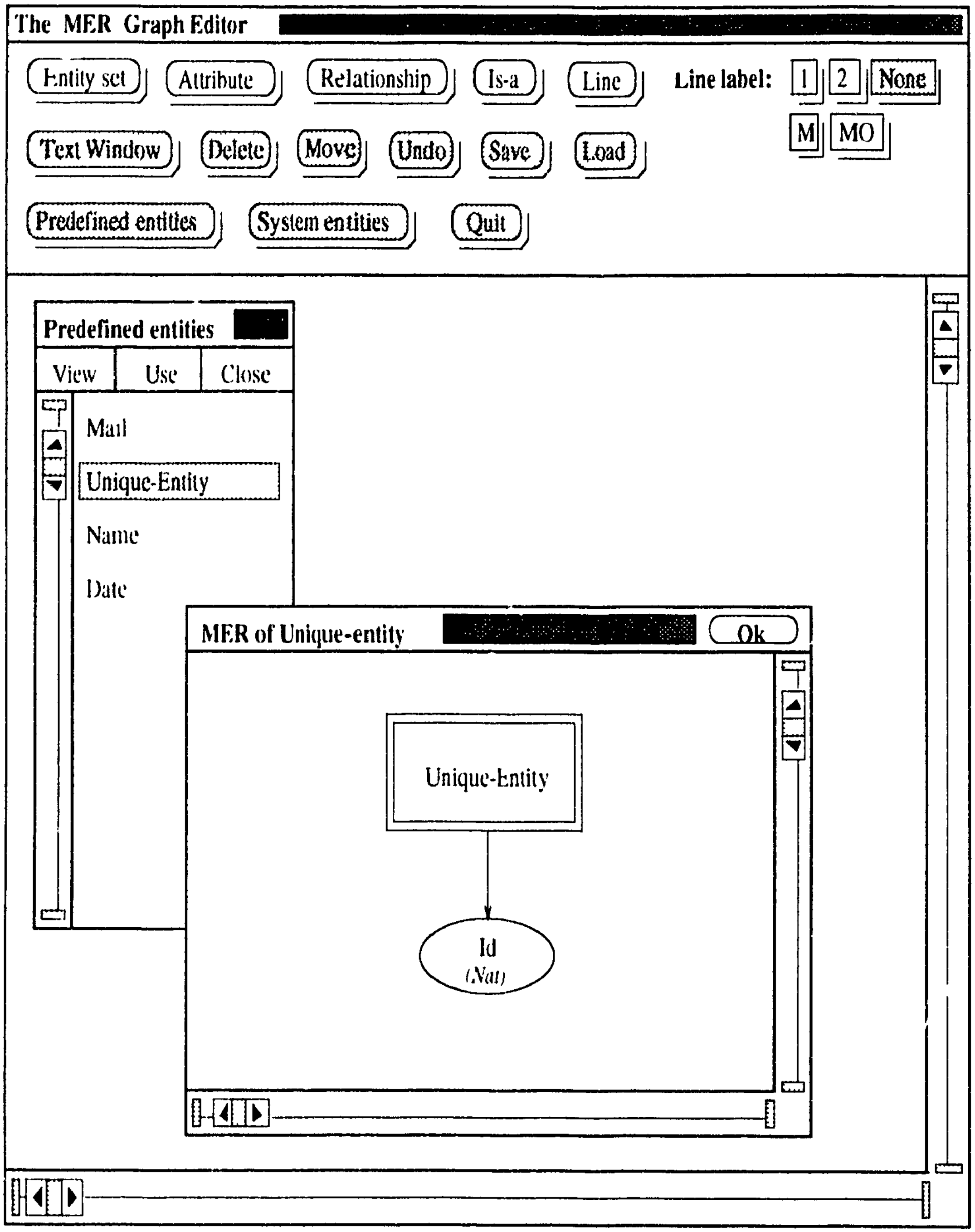


Figure 5.9: Viewing Predefined entity type *Unique-entity*

are not shown but are still assumed to be acquired. The "close" button will remove the two sub-windows of the "Predefined entities" function.

5.2.2 The MER Diagram of the Mailing System

Assuming the predefined entity types described in the previous section, the MER diagram for the mailing system is shown in figure 5.10. From the requirements given in section 5.1, the following entity types are identified by the user: *Deskring*, *Directory*, *User*, *Trays*, *Mailitem*, and *Ref*. *Deskring* is defined by a one-to-one relationship between *User* and *Trays*. It maps each user with a set of trays as stated in section 5.1. Entity type *User* inherits from the predefined entity type *Unique-entity* which characterizes an entity uniquely identified by an id number. Hence, entity type *User* is simply defined as a unique user identification number. *Trays* is defined by three primitive relationships which correspond to the three sets of trays. A tray is a set of mail items.

The *Mailitem* entity type inherits its main characteristics from the predefined entity type *Mail*. Additionally, the requirements specify a flag for replies and a reference number uniquely identifying each mail. This corresponds to the two attributes *Reply_req* and *Refno*, respectively. The relationship *Refs* defines a set of mail reference numbers to which this mail is a reply to. Then entity type *Ref* is a unique identification number for a mail item because it inherits from *Unique-entity*. Therefore, attribute *Refno* must also be of type *Ref*. The entity type *Directory* is defined by one high-order relationship called *Dirlist*. This relationship associates a user with the relationship set *Alias*. The *Alias* relationship is a one-to-one relationship between a name and a user (the name is an alias for that user). Hence, entity type *Directory* associates a directory of aliases for each user.

The mailing system is composed of two main entities, the ring of desk which includes the trays for each user, and a directory of user's aliases. The two system entities will be called DESKR and DIRECT which are of type *Deskring* and *Directory*, respectively. This information is not shown in the diagram of figure 5.10 but is assumed entered by the user through the "System entities" function. These two variables,

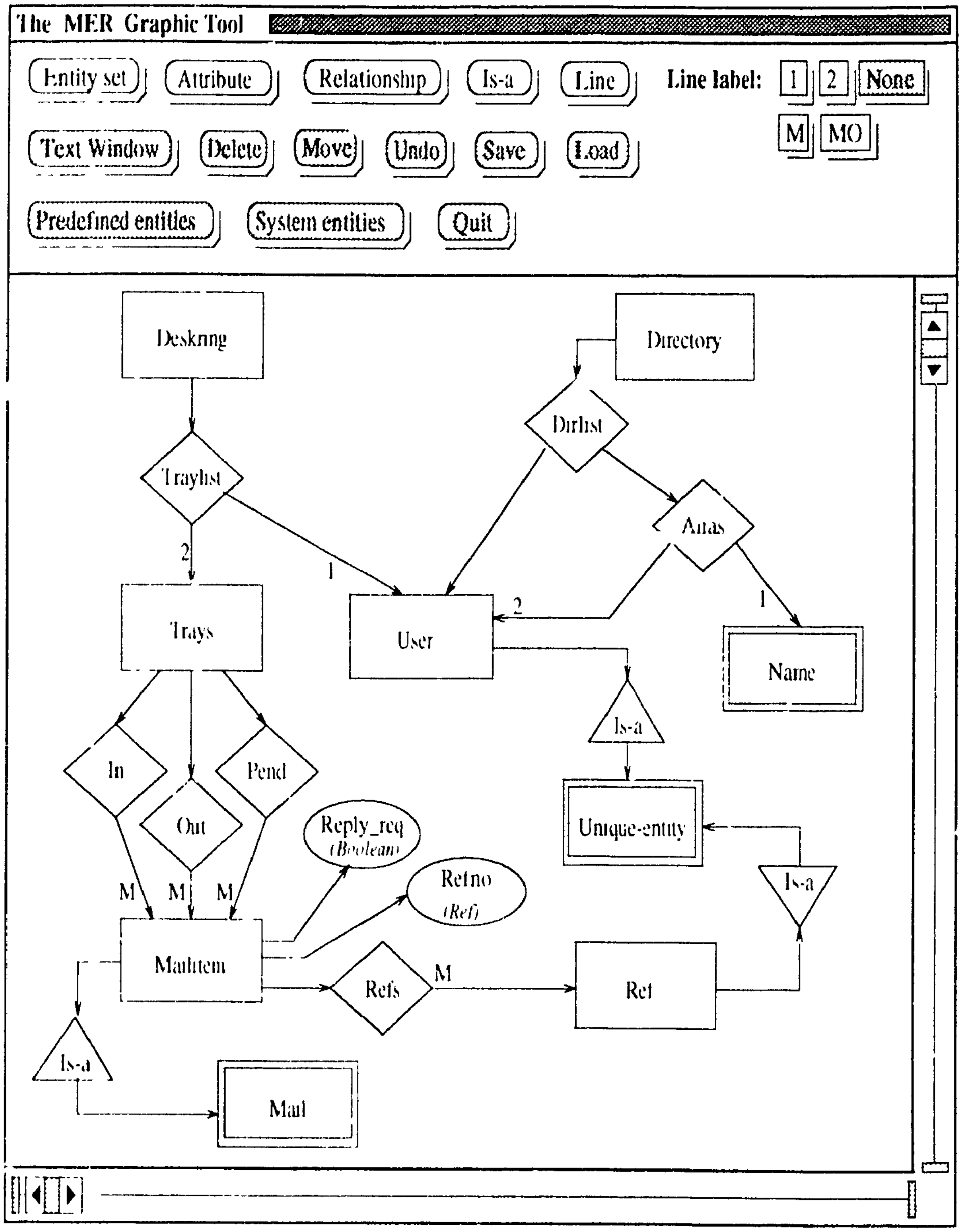


Figure 5.10: User-defined MER diagram of the mailing system

from their structure, include all the necessary information we need to know about our mailing system. Thus, DESKR and DIRECT are global entities to all operations

5.2.3 Current State of the MER Graph Tool

The MER graph editor presented has been built by [Nguyen]. Currently, the MER tool has all the functions to draw any MER component. The "Text window", "Delete" and "Move" functions are also implemented. The implementation of the MER graph editor is not complete yet. An MER diagram contains a wealth of information: entity types and their corresponding attributes and relationships; characteristics of relationships; the type of each relationship; the entity types which are specializations of general entity types; the system entities and their corresponding types, etc. These information are collected in the form of a table and fed as input to the description preprocessor. An example of this proposed table, pertaining to the mailing system, is given in figure 5.11. Each entity type has an entry in the MER table and is referred by that entry number. For each entity type, the attributes are listed with their names and their corresponding types. The type could be either an entry in the MER table or a code for a built-in type. The built in types are coded as follows

Integer : 66

Natural : 77

Text : 88

Boolean : 99

For each entity type, the relationships are listed by specifying the name, the type of the relationship (specified by a code), the domain entity type and the range entity type. The codes for the relationships are the following:

Primary (many) : 1

Primary (many-ordered) : 2

Second-order (one-to-one) : 3

Second-order (one-to-many) : 4

Second-order (one-to-many-ordered) : 5

High-order : 6

The domain and range entity types of a relationship are entry numbers in the MER table. However, for a high-order relationship, the range field specifies another relationship entry defined for the current entity. For example, if the relationship code is 6, the domain is 3 and the range is 2, then 3 is an entry in the MER table and 2 is the 2^{nd} relationship defined in the current entity. Finally, the inherited entity is an entry number in the MER table which specifies the entity type from which the current entity type inherits.

5.3 The KFD Input Tool

The KFDs can be developed with an ordinary text editor. However, since a KFD has a specific format, it would require an additional validation step. A general purpose text editor does not force any syntax on the textual description. The validation step will assume that a description follows the format of the KFD. Presently, the KFDs are written using a text editor and we assume the input is syntactically correct. This section proposes an appropriate editor to write KFDs which would be developed under a mouse-driven windowing environment. The important features the editor should possess are emphasized here rather than how they are provided.

5.3.1 The Proposed KFD Editor

Since the KFD possesses a fixed format, the editor should force the user in following that format. The proposed editor is shown on figure 5.12. and is relatively simple to use. The first row includes basic functions which are common to any editor. "Save" stores the currently edited KFDs under the specified name. "Load" retrieves a file of KFDs for editing purposes. Before explaining the second row of functions, let us look at how the user enters a KFD. A KFD has six parts which correspond to the six divided regions of the editors. These regions can be referred to as sub-windows. The user can start entering input for any of the parts by moving the cursor in the appropriate sub-window and clicking the mouse. When the sub-window is full, it automatically extends line by line as we keep on typing. The scrollbar permits to

MER Table								
Entry no.	*Entity-name*	*Attributes*		*Relationships*				*Inherited-entity*
		Name	*Type*	*Name*	*Code*	*Domain*	*Range*	
1	MailItem	Reply_req	99	Refs	1	4	Nil	10
		Refno	4					
2	Trays	Nil	Nil	In	1	1	Nil	Nil
				Out	1	1	Nil	
				Pend	1	1	Nil	
3	Name	Nil	Nil	Nil	Nil	Nil	Nil	88
4	Ref	Nil	Nil	Nil	Nil	Nil	Nil	6
5	User	Nil	Nil	Nil	Nil	Nil	Nil	6
6	Unique-entity	Id	77	Nil	Nil	Nil	Nil	Nil
7	Deskring	Nil	Nil	TrayList	3	5	2	Nil
8	Directory	Nil	Nil	DirList	6	5	2	Nil
				Alias	3	3	5	
9	Date	Nil	Nil	Nil	Nil	Nil	Nil	88
10	Mail	From	3	To	2	3	Nil	
		Subject	88	Cc	2	3	Nil	
		Whensent	9					
		Body	88					

Figure 5.11: MER table for the mailing system

move only within one KFD specification, when the KFD does not fit in the main window.

When the current KFD is completed, the function "Confirm" files the KFD in memory (so that it can be viewed later) and clears the editor to input a new KFD. The function "Delete" clears the editor and destroys whatever was specified in the current KFD. "Prev" and "Next" move around the KFDs entered so far (all those that have been confirmed). The "Search" function looks for a specific KFD previously entered. Upon clicking this function, a pop-up window opens requesting the name of the KFD to search for. Finally, the function of the "Keywords" button is purely informative. It gives an informative description of all valid keywords as well as high-order keywords. The next section explains it in more detail.

The inputs given in each of the six parts of a KFD are in free style, i.e. there is not restrictions on what the user writes. The KFDs are saved as a text file and only at a later stage will the syntax of the KFD be verified (refer to section 5.4) according to the grammar given in section 4.1.

5.3.2 High-order Keywords

High order keywords are stored operations to manipulate predefined entity types. These keywords can be used in the *Constraints* and *Semantics* clauses of a KFD. High order keywords are domain dependent and are provided by the domain expert. The "keywords" function opens a sub-window in which the keywords descriptions are given in two groups: the high-order keywords first and the general KFD keywords next. This is illustrated in figure 5.13. The description of the general keywords are brief since the user is assumed to know the language at this point. The "Prev" and "Next" buttons at the bottom of the sub-window allows the user to move around the pages of the keywords description. The "Ok" button closes the description window.

In our mailing system example, high-order keywords are: *now()* and *newid()*. The high-order keyword *now()* is associated with the predefined entity *Date*. It is a function which returns the current date. At this point, it is not necessary to know the representation of the value returned but rather to know that the value returned

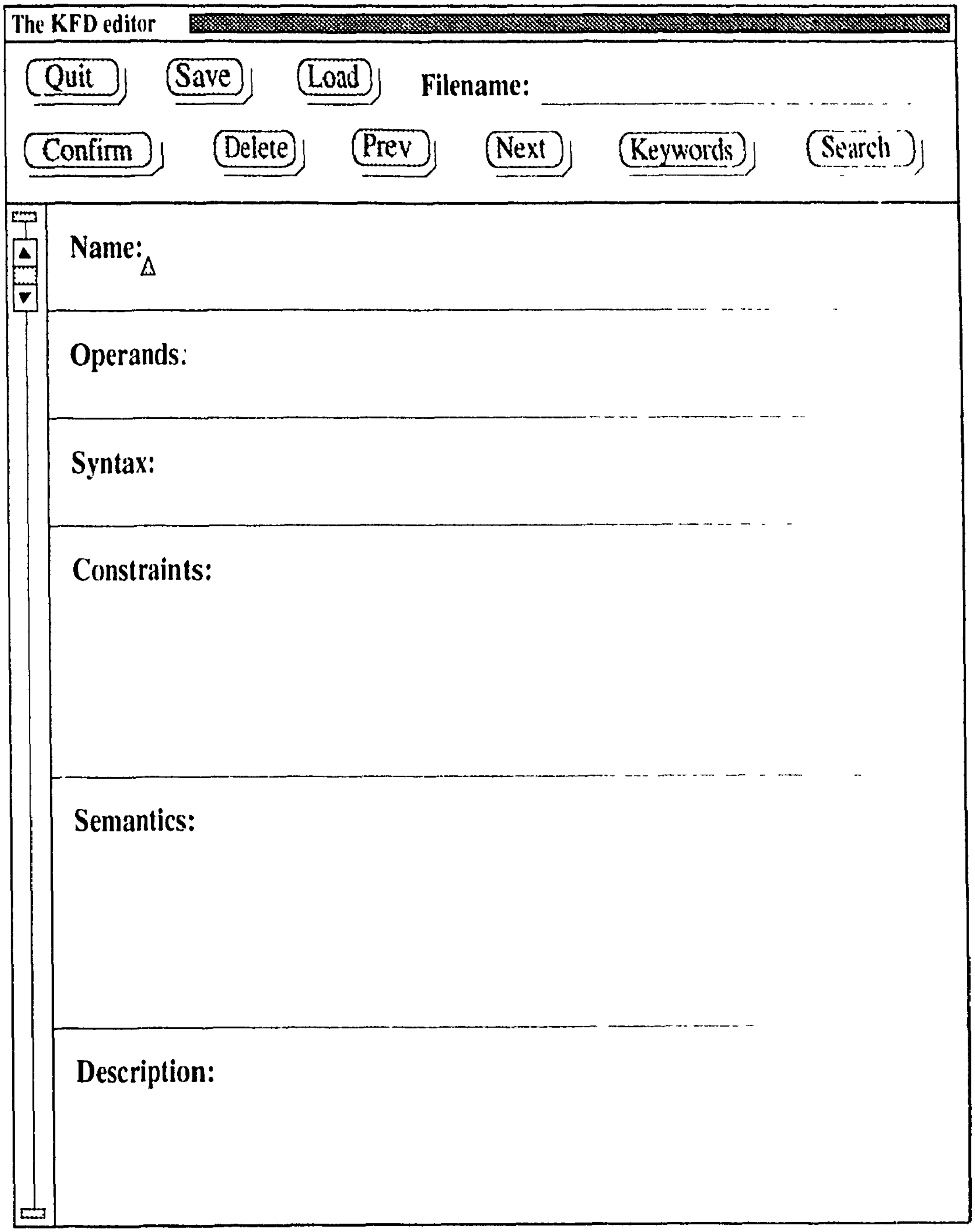


Figure 5.12: The KFD editor

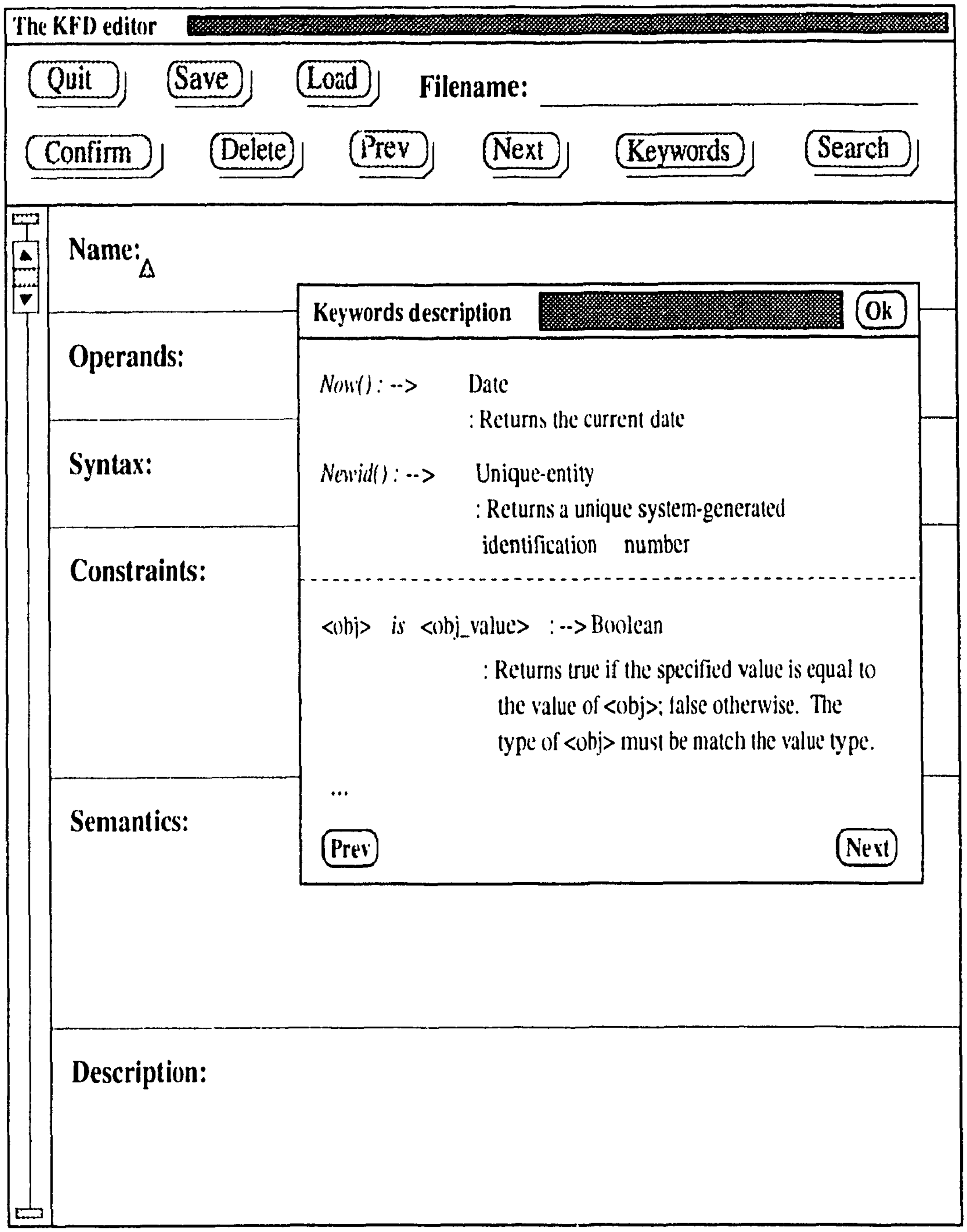


Figure 5.13: The keywords description function

is of type *Date*. The user can use *now()* in the description of the constraints or semantics of a KFD whenever the current date is requested without having to bother about the details of its representation. The second high-order keyword is associated with the predefined entity type *Unique-entity*. *newid()* generates a unique identifier. Again, how the function is implemented is not the issue; it might be assumed to be a system-generated function. The value returned is of type *Unique-entity*. However, the value might be assigned to an entity of a different type. This is valid as long as that entity type inherits from *Unique-entity*. Thus, if USERX is of type *User* in the mailing system, then *newid()* can be used to assign an unique identification number to USERX since *User* inherits from *Unique-entity*.

5.3.3 The KFDs of the Mailing System

The requirements described in 5.1 identifies six operations which must be provided by the mailing system. Each operation is described by a KFD. The KFDs for the mailing system are shown in figures 5.14 and 5.15. This section will go over each KFD and explain their important aspects.

Operation POST has two inputs of types *User* and *Mailitem*. The parameter variables are USERX and MAILX which are entities of the corresponding types. The operation is invoked by specifying which user is posting, the operation name, and which mail to post. The constraints are simple to understand since they are very close to English. It requires that attributes TO, SUBJECT, and BODY of MAILX are not empty prior to posting the mail. On the other hand, WHENSENT and REFNO of MAILX should be empty. It is important to notice that certain attributes have different types but the same keyword "empty" is used whether we refer to an empty string or a nil integer value. Another required condition is that USERX is a valid user in the desk ring, i.e. USERX is part of the relationship defining DESKR. DESKR is a system entity which is global to all operations. The relationship TRAYLIST associates each valid user to a group of trays. Thus, from the constraint "userx *is-in* Traylist(userx) *of* deskr" the system must understand that there must exist a tuple in the relationship where the first entity is USERX. This level of details is not required from the user but is expected to be deduced by the system. Finally, the *Semantics* clause describes the effect of the POST operation by adding MAILX in the out-tray associated to USERX.

The operation READ has two inputs and returns a value of type *Mailitem*. The two input parameter variables are USERX and MAILID. MAILID is of type *Ref*, which is a unique identification mail number. The constraints require that the given mail (identified by MAILID) exist in the pending tray associated to USERX in the desk ring. We notice here that PEND is a set of elements of type *Mailitem* and that MAILID is of type *Ref*. The system must then understand that what the user means is that there must exist a certain mail (of type *Mailitem*) in the PEND tray associated to USERX to which the reference number (REFNO) is equal to the given MAILID number. That level of details is not required from the user and type incompatibility is allowed. The effect of the operation is to return that specified mail item to the user's pad (to read). This is done with the keyword *return* which returns an entity of the return type, specified in the *Operands* clause. Thus, MAILX is an instance variable of type *Mailitem*, created by the *return* keyword. MAILX is defined as the mail in PEND associated to USERX which is identified by MAILID (REFNO = MAILID).

The operation COLLECT works the same way as READ except that the mail is taken from the IN tray (a received mail) and removed from it after it has been returned to the user's pad. This is done with the *remove* keyword which removes the entity MAILX, created by the *return* function, from the IN tray associated to USERX of the desk ring.

Operation ADDALIAS has three input parameters of type *User*, *Name*, and *User* respectively. The operation "userx *addalias* namey usery" means that USERX is creating an alias for USERY with the name NAMEY. The constraints require that the given name NAMEY does not exist already in the directory of aliases, DIRLIST, associated to USER. DIRLIST is defined in the global entity DIRECT. Another constraint is that USERY is a valid user defined in the directory of users of DIRECT. The effect of the operation is to add the tuple (NAMEY, USERY) in the relationship set (the directory of aliases) associated to USERX.

DELALIAS does the inverse of ADDALIAS. Given a *User*, a *Name* and another

User respectively identified by USERX, NAMEY and USERY. DELALIAS removes the tuple (NAMEY, USERY) from the directory of aliases associated to USERX. Thus, the constraints ensure that prior to the execution of DELALIAS: 1) NAMEY is in the relationship set (directory of aliases) associated to USERX in the high-order relationship DIRLIST and 2) NAMEY is actually the alias for USERY in the directory of aliases associated to USERX. DIRLIST(USERX)(NAMEY) refers to the user associated to NAMEY in the relationship ALIAS which is associated to USERX in the relationship DIRLIST. The semantics of DELALIAS is to remove the tuple for NAMEY from the relationship set ALIAS associated to USERX in DIRLIST.

The CLEAR operation has only one input of type *User* which is referred as USERX. The condition for this operation to be executed is that the OUT tray associated to USERX in the relationship TRAYLIST must not be empty, i.e. USERX has some mails to be sent. The semantics of CLEAR is composed of two main actions. These are determined by the action keywords *for-all* and *set*. The first main action accesses all the mails in the OUT tray associated to USERX. They are referred to as MAILY. For each MAILY, the following sub-actions will be done:

1) a new mail is created using the attribute values of MAILY. Additionally, the new mail carries a date and a unique reference number. This new mail is called NEWMAIL.

2) For each of the names specified in TO of MAILY, the NEWMAIL will be added in the IN tray of the user associated to that name. DIRLIST(USERX)(NAMEI) gives the user identified by NAMEI in the directory of aliases of USERX. Let that user be called USERY. Then, TRAYLIST(USERY) refers to the set of trays of USERY. Also, if a reply to MAILY is required, NEWMAIL will also be added in the PEND tray of USERY.

3) Similarly, for each of the names specified in CC of MAILY, the NEWMAIL will be added in the IN tray of the user associated to that name.

4) For each mail M in the PEND tray associated to USERX, we will verify if MAILY is a reply to that mail. Thus, if the reference number of M is in the set of reference numbers REFS of MAILY, then MAILY replies to M. In that case, M will be removed from the PEND tray associated to USERX in the relationship TRAYLIST.

Finally, the OUT tray of USERX will be emptied since the mails have been dropped in the appropriate in-trays of the recipients.

5.4 The Preprocessing Step

The role of the preprocessor is to ensure that the input description is syntactically correct, that the MER and KFDs are consistent with each other, and that the input description is as complete as it can be, with reference to the "common knowledge". The description preprocessor will transform the correct input description into an intermediary form that will be understood by the transformation system. The task of the preprocessor is to ensure a proper input is given to the transformation step. At the present moment, the preprocessing is done algorithmically but manually. This section presents the job of the preprocessor so that it can be automated. The "common knowledge" assumed is about sets and functions and knowledge about the MER and KFD structure and semantics. Both types of knowledge are embodied in the preprocessing procedures and are not separate components as shown in figure 1.2. The important point is to recognize their existence and ultimately think about a method to represent them independently from the process.

5.4.1 Parsing the Input Description

The MER diagrams, if built with the proposed graph editor, cannot include syntactic or semantic errors. They can only possibly have logical errors which are totally dependent on the user. Naturally, the system cannot detect such errors. On the other hand, the KFDs might include syntactic, semantic and logical errors. Again, nothing can be done about a user's logical error. However, a parser can take care of the other two. It is easy to build a parser for KFDs from the syntax given in section 4.1. In fact for each KFD, the following checks must be done:

1. the operation name must be unique among the KFDs.
2. the types in the *Operands* section must be valid types defined in the MER diagram (includes predefined types) or built-in types.

Operation name:	POST
Operands:	User, Mailitem
Syntax:	userx *post* mailx
Constraints:	To *of* mailx *is not empty and* Subject *of* mailx *is not empty and* Body *of* mailx *is not empty and* Whensent *of* mailx *is empty and* Refno *of* mailx *is empty and* userx *is-in* TrayList *of* deskr
Semantics:	*add* mailx *in* Out *of* TrayList(userx) *of* deskr
Description:	Transfer the mail item "mailx" (from the user's pad) into the out-tray of "userx". The mail item consists of a document and partially completed mail system header

Operation name:	READ
Operands:	User, Ref → Mailitem
Syntax:	userx *read* mailid
Constraints:	mailid *is-in* Pend *of* TrayList(userx) *of* deskr
Semantics:	*return* mailx *such that* mailx *is-in* Pend *of* TrayList(userx) *of* deskr *and* Refno *of* mailx *is* mailid
Description:	Return (to the user's pad) a copy of the mail item identified by the reference number "mailid" from the pending tray of "userx". The mail item is not removed from the pending tray

Operation name:	COLLECT
Operands:	User, Ref → Mailitem
Syntax:	userx *collect* mailid
Constraints:	mailid *is-in* In *of* TrayList(userx) *of* deskr
Semantics:	*return* mailx *such that* mailx *is-in* In *of* TrayList(userx) *of* deskr *and* Refno *of* mailx *is* mailid *remove* mailx *from* In *of* TrayList(userx) *of* deskr
Description:	Return (to the user's pad) a copy of the mail item identified by the reference number "mailid" from the pending tray of "userx". The mail item is transferred to the user's pad thus, it is removed from the pending tray

Operation name:	ADDALIAS
Operands:	User, Name, User
Syntax:	userx *addalias* namey, usery
Constraints:	namey *not is-in* DirList(userx) *of* direct *and* usery *is-in* DirList *of* direct
Semantics:	*add* (namey, usery) *in* DirList(userx) *of* direct
Description:	Add alias "namey" for "usery" in the directory of aliases of "userx"

Figure 5.14: KFDs for the mailing system operations (I)

Operation name: DELALIAS
Operands: User, Name, User
Syntax: userx *delalias* namey, usery
Constraints: DirList(userx)(namey) *of* direct *is* usery
Semantics: *remove* namey *from* DirList(userx) *of* direct
Description: Delete the "namey" associated to "usery" from the directory of aliases of "userx".

Operation name: CLEAR
Operands: User
Syntax: *clear* userx
Constraints: Out *of* TrayList(userx) *of* deskr *is not empty*
Semantics:

```
for all mailx in Out of TrayList(userx) of deskr do
        create newmail : Mailitem with To of mailx, Cc of mailx, From of mailx,
                                       Subject of mailx, Reply_req of mailx, Refs of mailx,
                                       now(), newid(), Body of mailx
        for-all namei in To of mailx do
                add newmail in In of TrayList(DirList(userx)(namei) of direct) of deskr
                if Reply_req of mailx then
                        add newmail in Pend of TrayList(DirList(userx)(namei) of direct) of deskr
                end if
        end for
        for-all namej in Cc of mailx do
                add newmail in In of TrayList(DirList(userx)(namej) of direct) of deskr
        end for
        for-all m in Pend of TrayList(userx) of deskr do
                if Refno of m is-in Refs of mailx then
                        remove m from Pend of TrayList(userx) of deskr
                end if
        end-for
end for
set Out of TrayList(userx) of deskr to empty
```

Description: Empty the out-tray of "userx" and copies each mail in the appropriate in-trays of the intended recipients. If a reply for a mail is required, insert also a copy in the pending-tray of the intended user. If a mail sent was a reply to some previously received mail by "userx", then delete the mail from the pending-tray of "userx".

Figure 5.15: KFDs for the mailing system operations (2)

3. the instance variables in the *Syntax* section should not have the same name as any of the system entities.

4. the *constraints* should follow the syntax given in section 4.1.

5. the actions of the *Semantics* clause should follow the syntax given in section 4.1

6. the attributes and relationships of entities referred in the *Constraints* and *Semantics* section must correspond to the definitions given in the MER diagram

7. if a value is returned with the keyword *return*, the proper return type should be specified in the *Operands* clause. (If a return type is specified, there must exist a *return* action in the *Semantics* clause.)

8. for each keyword, the parameter types should be compatible with its semantics as described in section 4.2.

When the above checks are completed, we assume the KFDs are syntactically and semantically correct and that the MER and KFDs are consistent with each other. The scanning and the parsing processes embody the knowledge about the structure of MER and KFD.

5.4.2 Completing the Input Description

In writing the functional specification using KFDs, the user is given some freedom. For example, if we refer to *r(e)*, where *r* is a second order or high order relationship and *e* is the first entity in the two tuple, we must be sure that *e* is in fact part of the relationship set, otherwise *r(e)* is undefined. For the user, this is assumed so she doesn't have to bother to specify it. However, to be complete, the specification must explicitly state that *e* must be part of the domain elements of *r*. Therefore, whenever *r(e)* is specified for some relationship *r* in the KFD, the additional constraint that "e *is-in* r" must be derived first. This additional constraint is not necessary when *r(e)* is referred within the scope of a "*for-all* e *in* r *do*".

In the KFDs of the mailing system example, there are few cases where the user did not bother to verify that the entity is part of the second or high order relationship set before referring to it. Thus, for the operations READ and COLLECT, we must add the following constraint: "userx *is-in* TrayList *of* deskr". For operations ADDALIAS and DELALIAS, we add the following constraint: "userx *is-in* DirList *of* direct". For operation DELALIAS, we must also state that "namey *is-in* DirList(userx) *of* direct". Finally, in operation CLEAR, the constraints must also state that "userx *is-in* TrayList *of* deskr".

Common knowledge about sets and relationships permit us to notice such incompleteness in the description. However, if we want to store this knowledge and use it in an automated process, a method to represent knowledge is required. [Johnson'92] discusses several ways to represent knowledge. A rule based knowledge representation seems appropriate for this case. For example, the parser could refer to those rules when it recognizes a relationship and apply them, if necessary. Those rules would be independent of any application domain.

5.4.3 The Intermediate Form

The preprocessing step transforms the input description into an intermediate form which is easier to manipulate by the transformation process. Basically, the information given in the input is kept in tables. The table for the MER diagram keeps the same format, as introduced in section 5.2.3. The KFDs will be kept in a table of records where each record defines one KFD. The record will possess the following fields:

- *Opname* (The name of the operation)
- *Parameters* (A table of the parameters for the operation)
- *Return parameter* (The specification of the returned entity, if any)
- *Constraints* (A table of constraints)
- *Actions* (A table of actions)

- *Lst* (A local symbol table)

Each field is defined below.

Opname
Text

The name is a string value.

Parameters		
1	$Name_1$	$MerEntry_1$
2	$Name_2$	$MerEntry_2$
...	...	...

The *Parameters* field is a table where the name of each parameters is given along with its type. The type is specified as an entry in the MER table or as a built-in type code (refer to section 5.2.3).

Return parameter	
Name	MerEntry

The return parameter has a name and a type which is specified as an entry in the MER table or a built in type code.

Constraints/Actions									
1	$Kcode_1$	$Foper_1$	$Soper_1$	$From1_1$	$To1_1$	$From2_2$	$To2_1$	$From3_3$	$To3_3$
2	$Kcode_2$	$Foper_2$	$Soper_2$	$From1_2$	$To1_2$	$From2_2$	$To2_2$	$From3_2$	$To3_2$
3	...	...	...	...	...	...	...	...	...

There is a separate table, one for constraints and one for actions, following the above format. Each constraint or action is described by the following fields.

Kcode : A code to identify the keyword. The following codes are assumed.

0 -	No more constraints/actions	1 -	*is*
2 -	*is not*	3 -	>
4 -	*not* >	5 -	<
6 -	*not* <	7 -	*is-in*

8 - *not is-in* 9 - *add*

10 - *remove* 11 - *set*

12 - *create* 13 *if-then-else*

14 - *for-all* 15 - *return*

The rest of the fields have different interpretations depending on the keyword

- For keywords *is, is [not], > [not], < [not], [not] is-in, add, remove, set* :

  Foper : entry number in the *Lst* for the first keyword operand.

  Soper : entry number in the *Lst* for the second keyword operand.

  From1 : Nil.

  To1 : Nil.

  From2 : Nil.

  To2 : Nil.

  From3 : Nil.

  To3 : Nil.

- For the keyword *if-then-else* :

  Foper : Nil.

  Soper : Nil.

  From1 : entry number in the *Constraints* or *Actions* table for the first condition of the **if** part.

  To1 : entry number in the *Constraints/Actions* table for the last condition of the **if** part.

  From2 : entry number in the *Constraints/Actions* table for the first condition/action of the **then** part.

  To2 : entry number in the *Constraints/Actions* table for the last condition/action of the **then** part.

  From3 : entry number in the *Constraints/Actions* table for the first condition/action of the **else** part.

  To3 : entry number in the *Constraints/Actions* table for the last condition/action of the **else** part.

- For the keyword *for-all* :

  Foper : entry number in the *Lst* for the instantiated entity.

  Soper : entry number in the *Lst* for the relationship set.

  From1 : entry number in the *Constraints/Actions* table for the first condition/action to apply.

  To1 : entry number in the *Constraints/Actions* table for the last condition/action to apply.

  From2 : Nil.

  To2 : Nil.

  From3 : Nil.

  To3 : Nil.

- For keyword *return* :

  Foper : entry number in the *Lst* for the entity returned.

  Soper : Nil.

  From1 : entry number in the *Actions* table for the first condition, if any.

  To1 : entry number in the *Actions* table for the last condition, if any.

  From2 : Nil.

  To2 : Nil.

  From3 : Nil.

  To3 : Nil.

- For keyword *Create* :

  Foper : entry number in the *Lst* for the entity created.

  Soper : entry number in the *MER* table for the entity type.

  From1 : entry number in the *Lst* for the first value to assign.

  To1 : entry number in the *Lst* for the last value to assign.

  From2 : Nil.

  To2 : Nil.

  From3 : Nil.

  To3 : Nil.

The following tables describe the KFDs for the operations POST and READ. For want of space, the tables for the other KFDs are not presented here but can be derived similarly.

POST

Opname
POST

Parameters		
1	userx	5
2	mailx	1

Return parameter	
Nil	Nil

Constraints									
1	2	1	3	Nil	Nil	Nil	Nil	Nil	Nil
2	2	4	3	Nil	Nil	Nil	Nil	Nil	Nil
3	2	6	3	Nil	Nil	Nil	Nil	Nil	Nil
4	1	8	3	Nil	Nil	Nil	Nil	Nil	Nil
5	1	10	3	Nil	Nil	Nil	Nil	Nil	Nil
6	7	12	13	Nil	Nil	Nil	Nil	Nil	Nil

Actions									
1	9	2	14	Nil	Nil	Nil	Nil	Nil	Nil

Lst			
Lst Entry	*Code*	*Opstring*	*Type*
1	3	To	7
2	1	mailx	1
3	2	*empty*	5
4	3	Subject	2
5	1	mailx	1
6	3	Body	2
7	1	mailx	1
8	6	Whensent	9
9	1	mailx	1
10	6	Refno	4
11	1	mailx	1
12	1	userx	5
13	1	deskr	7
14	4	Out	6
15	5	userx	Nil
16	1	deskr	7

READ

Opname
READ

Parameters		
1	userx	5
2	mailid	1

Return parameter	
mailx	1

Constraints									
1	7	8	4	Nil	Nil	Nil	Nil	Nil	Nil
2	7	1	2	Nil	Nil	Nil	Nil	Nil	Nil

Actions									
1	15	5	Nil	2	3	Nil	Nil	Nil	Nil
2	7	5	2	Nil	Nil	Nil	Nil	Nil	Nil
3	1	6	1	Nil	Nil	Nil	Nil	Nil	Nil

Lst			
Lst Entry	*Code*	*Opstring*	*Type*
1	1	mailid	4
2	4	Pend	6
3	5	userx	Nil
4	1	deskr	8
5	1	mailx	1
6	3	Refno	4
7	1	mailx	1
8	1	userx	5

It is ideal to generate the above tables as a part of the scanning and parsing processes. The set of all tables is the output of the preprocessing step.

5.5 The Transformation Step

This section presents the procedures used to generate the VDM specification from the encoded MER and KFDs. The transformation process has been implemented in C. It expects an input in the form presented in section 5.4.3, generates a temporary LaTeX file and then outputs the VDM specification in a postscript file. We needed a type setting language that would be able to produce any of the VDM symbols. LaTeX satisfies these requirements. The temporary LaTeX file is compiled by the transformation program to produce the postscript file and the compilation is invisible to the user.

The algorithms presented here are in pseudo-code form. The algorithms are described in terms of MER and KFD concepts rather than using the code values and the table entries. This makes the algorithms more understandable. The actual program manipulates the appropriate tables.

5.5.1 From MER model to VDM State

The MER diagram and the system entities are the only information required to build the VDM state. In generating the VDM state, each entity of the MER diagram (user-defined or predefined) is transformed into a VDM record type. The attribute names and relationship names defining the entity type become fields of the VDM record. Primitive relationships are VDM sets or lists depending on whether the relationships is labeled "M" or "MO" respectively. Second-order relationships associate an entity type to another entity type or to a set or an ordered set of another entity type. They are translated to VDM as mapping types. The first entity type in the 2-tuple is mapped into the second entity type or into a set or list of entities of the second type. A high-order relationship associating entity of type E to a relationship R is defined as a mapping from E to R where R is another mapping type in VDM. For every entity type A inheriting from an entity type B, the VDM record **A** will include the fields of **B**.

A more formal specification of the algorithm used to generate the VDM state is given here. When specifying a record type in VDM, we use two colons (::). VDM also has simple types, denoted by the equal sign (=), which are types not composed of fields. Simple types are defined by another type (built-in or user-defined) or by a set or a list or a mapping of other types. The terms record type and simple type are used in the algorithm given below.

Algorithm to generate the VDM state

STEP 1, Transformation rules

1. For each user-defined entity *ent* in the model do

1.1 Create a RECORD type in VDM named **ent**

1.2 For each attribute *attr* of entity *ent* do

1.2.1 Create a field named **attr** in the RECORD **ent**.

The type of the field **attr** is the type of the attribute *attr*.

1.3 For each relationship *rel* defined in the entity *ent* do

1.3.1 Create a field named **rel** in the RECORD **ent**.

1.3.2 If *rel* is a primitive relationship pointing to entity *entx* then

1.3.2.1 If the arrow pointing to *entx* is labelled "M" then

1.3.2.1.1 the type of field **rel** is the POWERSET type **entx-set**

1.3.2.2 If the arrow pointing to *entx* is labelled "MO" then

1.3.2.2.1 the type of field **rel** is the LIST type **entx-list**.

1.3.3 If *rel* is a second-order relationship from *entx* to *enty* then

1.3.3.1 If *rel* is a one-to-one relationship from *entx* to *enty* then

1.3.3.1.1 the type of field **rel** is the MAPPING type **entx** $\rightarrow$ **enty**.

1.3.3.2 If *rel* is a one-to-many relationship from *entx* to *enty* then

1.3.3.2.1 If the arrow pointing to *enty* is labelled "M" then

1.3.3.2.1.1 the type of field **rel** is the MAPPING type
entx $\rightarrow$ **enty-set**.

1.3.3.2.2 If the arrow pointing to *enty* is labelled "MO" then

1.3.3.2.2.1 the type of field **rel** is the MAPPING type
entx $\rightarrow$ **enty-list**.

1.3.4 If *rel* is a high-order relationship from *entx* to relationship *relx* then

1.3.4.1 the type of field **rel** is the MAPPING type **entx** $\rightarrow$ **relx**.

1.3.4.2 Create a SIMPLE VDM type **relx** which will be defined according to rules 1.3.3 and 1.3.4 depending on the type of *relx*.

1.4 For each *is-a entx* relationship defined in the entity *ent* do

1.4.1 Include all fields of the VDM RECORD for *entx* into the VDM RECORD **ent** (i.e. all the fields of **entx** and their corresponding types will be added to the definition of **ent** also).

2. Create a VDM global variable for each system entity. The type of the variable is the system entity type.

Given the MER diagram of the mailing system, the above algorithm will generate the VDM state shown in figure 5.16. From the VDM state of figure 5.16, we can notice

State ::

DESKR	: *Deskring*		
DIRECT	: *Directory*		
Deskring	:: TrayList : *User* → *Trays*		
Directory	:: DirList : *User* → *Alias*		
Trays	::	IN	: *Mailitem-set*
		OUT	: *Mailitem-set*
		PEND	: *Mailitem-set*
Mailitem	::	TO	: *Name-list*
		CC	: *Name-list*
		FROM	: *Name*
		SUBJECT	: *Text*
		REPL_REQ	: *Bool*
		REFS	: *Ref-set*
		WHENSENT	: *Date*
		REFNO	: *Ref*
		BODY	: *Text*
Unique-entity	:: ID : *Nat*		
Mail	::	TO	: *Name-list*
		CC	: *Name-list*
		FROM	: *Name*
		SUBJECT	: *Text*
		WHENSENT	: *Date*
		BODY	: *Text*
Alias	= *Name* → *User*		
Ref	:: ID : *Nat*		
Date	::		
User	:: ID : *Nat*		
Name	::		

Figure 5.16: Generated VDM state for the mailing system

that it is possible to refine it further. We need to adjust the record types having no fields or one field to simple VDM types. This is done with the following refinement step.

STEP 2, Refinement rules

1. For each VDM RECORD type **ent** previously created for each entity *ent* do

1.1 If **ent** has no field (*ent* is a single value entity) then

1.1.1 Change the RECORD type **ent** for a SIMPLE VDM type.
The type of **ent** is the type of the single value entity.

1.2 If **ent** has only one field **f** defined with type **t** then

1.2.1 Change the RECORD type **ent** for a SIMPLE VDM type.

1.2.2 Remove the single field **f**.

1.2.3 The SIMPLE type **ent** is now defined as **t**.

1.2.4 In the KFDs, change every reference of attribute *f* **of** *ent* to *ent*, and of $f(x_1)(x_2)$... **of** *ent* to $ent(x_1)(x_2)$....

Then, the final state of the diagram is given in figure 5.17.

5.5.2 From Formatted Keyword-based Descriptions to VDM Operations

The KFDs are the main source of information used to generate the VDM operations. However, we also need to refer to the previously generated VDM state to observe how an entity is defined. In this section, the rules performing the transformation from KFD to VDM operation are presented. The terms used in the procedures are defined in section 4.1. The operation name of the KFD becomes the VDM operation name. The KFD names are ensured to be unique.

Generating the VDM operation header

The instance variables in the *Syntax* section of the KFD operation are the parameters of the VDM operation. Their corresponding types are listed in the *Operands* section of the KFD operation. For each KFD header:

State ::

DESKR	: *Deskring*		
DIRECT	: *Directory*		
Deskring	= *User* → *Trays*		
Directory	= *User* → *Alias*		
Trays	:: IN		: *Mailitem-set*
	OUT		: *Mailitem-set*
	PEND		: *Mailitem-set*
Mailitem	:: TO		: *Name-list*
	CC		: *Name-list*
	FROM		: *Name*
	SUBJECT		: *Text*
	REPL_REQ		: *Bool*
	REFS		: *Ref-set*
	WHENSENT		: *Date*
	REFNO		: *Ref*
	BODY		: *Text*
Unique-entity	= *Nat*		
Mail	:: TO		: *Name-list*
	CC		: *Name-list*
	FROM		: *Name*
	SUBJECT		: *Text*
	WHENSENT		: *Date*
	BODY		: *Text*
Alias	= *Name* → *User*		
Ref	= *Nat*		
Date	:: Text		
User	= *Nat*		
Name	:: Text		

Figure 5.17: Generated and refined VDM state for the mailing system

Operation name : *oper*

Operands : $ent_1, ent_2, \ldots, ent_n$ [$\rightarrow$ [M/MO] *retype*]

Syntax : $var_1 var_2 \ldots var_i oper var_j \ldots var_l$

we will generate :

OPER($VAR_1 : ent_1, VAR_2 : ent_2, \ldots, VAR_n : ent_n$) [R : *retype*/*-set lst;*]

If the operation returns a value, its type is specified after a right arrow ($\rightarrow$) following the list of input operands in the **operands** section. If "M" is specified, then the returning type is *retype-set*. If "MO" is specified, then the returning type is *retype lst*. In VDM, a variable (or a set) **R** of type *retype* will be returned where **R** will be determined by the *return* action in the *Semantics* section of the KFD for operation *oper* (refer to CASE 5 of the *Semantics* rules in section 5.5.2). Each VDM operation includes a clause which indicates which global state variables the operation needs to access. This is specified by the keyword **ext**. For every global system entity *ent* of type ENTYPE used in a KFD where *ent* is affected by the keywords *add*, *remove* or *set* in the *Semantics* clause of the KFD, we will generate in the VDM operation,

ext ENT : **wr** Entype.

If *ent* is only referred to but never affected by those action keywords in a KFD, then we will generate in the VDM operation:

ext ENT : **rd** Entype.

Generating the VDM preconditions

The constraints represent conditions that must hold before executing the operation In the *Constraint* clause, zero or more constraints are listed, each separated by the logical operator **and** or **or**. One constraint can correspond to more than one condition in the VDM **precondition** part. The logical operator **and** corresponds to $\wedge$ in VDM and the operator **or** corresponds to $\vee$. Hence, the list of constraints will be translated by a list of VDM preconditions, each separated by $\wedge$ or $\vee$. The information captured in the MER diagram is used in this transformation process.

Transformation of each constraint of the *Constraints* clause

In the following rules, read the arrow $\Longrightarrow$ as **is translated in VDM to.** The transformation rules are grouped into five major cases based on the **test** keywords. Each case is analyzed by considering the parameters of the keyword separately and in the context of the meaning of the keyword.

CASE 1 For each constraint of the form $\langle obj\rangle$ $\langle operator\rangle$ $\langle obj\text{-}value\rangle$ do

1. For the $\langle obj\rangle$ specification

1.1 If $\langle obj\rangle$ is an entity or a set referred as variable ent then

1.1.1 $\langle obj\rangle \Longrightarrow ent.$

1.2 If $\langle obj\rangle$ is an attribute or a sub-attribute defined as $attr_1$ **of** $attr_2$ **of** ... **of** $attr_n$ **of** ent then

1.2.1 $\langle obj\rangle \Longrightarrow ATTR_1(ATTR_2(...ATTR_n(ent)...))$

1.3 If $\langle obj\rangle$ refers to an entity or to a relationship set part of a higher level relationship R where R is $rel(e_1)(e_2)...(e_n)$ **of** ent then

1.3.1 $\langle obj\rangle \Longrightarrow \mathrm{REL}(ent)(e_1)(e_2)...(e_n)$

1.4 If $\langle obj\rangle$ is defined as the number **number-of** $\langle group\rangle$ where $\langle group\rangle$ is a relationship set referred by $rel(e_1)(e_2)...(e_n)$ **of** ent then

1.4.1 $\langle obj\rangle \Longrightarrow$ **card** $\mathrm{REL}(ent)(e_1)(e_2)...(e_n)$

1.5 If $\langle obj\rangle$ is defined as the entity or relationship set referred by **first-of** $\langle group\rangle$ where $\langle group\rangle$ is a relationship set referred by $rel(e_1)(e_2)...(e_n)$ **of** ent then

1.5.1 $\langle obj\rangle \Longrightarrow$ **hd** $\mathrm{REL}(ent)(e_1)(e_2)...(e_n)$

2. For the $\langle operator\rangle$ specification

2.1 If the $\langle operator\rangle$ is the keyword **is** then

2.1.1 $\langle operator\rangle \Longrightarrow =$

2.2 If the $\langle operator\rangle$ is the keyword **is not** then

2.2.1 $\langle operator\rangle \Longrightarrow \neq$

2.3 If the $\langle operator\rangle$ is the keyword $<$ then

2.3.1 $\langle operator\rangle \Longrightarrow <$

2.4 If the $\langle operator \rangle$ is the keyword **not** $<$ then

2.4.1 $\langle operator \rangle \Longrightarrow \not<$

2.5 If the $\langle operator \rangle$ is the keyword $>$ then

2.5.1 $\langle operator \rangle \Longrightarrow >$

2.6 If the $\langle operator \rangle$ is the keyword **not** $>$ then

2.6.1 $\langle operator \rangle \Longrightarrow \not>$

3. For the $\langle obj\text{-}value \rangle$ specification

3.1 If $\langle obj\text{-}value \rangle$ is the keyword **empty** then

3.1.1 If $\langle obj \rangle$ is a primitive relationship of "M" entities e then

3.1.1.1 $\langle obj\text{-}value \rangle \Longrightarrow \{\}$

3.1.2 If $\langle obj \rangle$ is a primitive relationship of "MO" then

3.1.2.1 $\langle obj\text{-}value \rangle \Longrightarrow ()$

3.1.3 If $\langle obj \rangle$ is a second order or high-order relationship then

3.1.3.1 $\langle obj\text{-}value \rangle \Longrightarrow [\,]$

3.1.4 If $\langle obj \rangle$ is of type *Text* then

3.1.4.1 $\langle obj\text{-}value \rangle \Longrightarrow$ ""

3.1.5 If $\langle obj \rangle$ is of a numerical type then

3.1.5.1 $\langle obj\text{-}value \rangle \Longrightarrow$ **NIL**

3.1.6 If $\langle obj \rangle$ is of type boolean then

3.1.6.1 $\langle obj\text{-}value \rangle \Longrightarrow$ **NIL**

3.2 If $\langle obj\text{-}value \rangle$ is any other numerical, text or boolean constant value then

3.2.1 $\langle obj\text{-}value \rangle \Longrightarrow \langle obj\text{-}value \rangle$

3.3 If $\langle obj\text{-}value \rangle$ is defined as $\langle obj \rangle$ then

3.3.1 Transform $\langle obj \rangle$ according to rules 1. Let o be the result.

3.3.2 $\langle obj\text{-}value \rangle \Longrightarrow o$

4. Let o be the transformation result of $\langle obj \rangle$. op be the result of $\langle operator \rangle$ and v be the result of $\langle obj\text{-}value \rangle$.

4.1 $\langle obj \rangle$ *langleoperator*$\rangle$ $\langle obj\text{-}value \rangle \Longrightarrow o\ op\ v$

CASE 2 For each constraint of the form $\langle obj \rangle$ **is-in** $\langle group \rangle$ do

1. Transform $\langle obj \rangle$ according to rules 1 of CASE 1. Let the result be o.

2. Transform $\langle group \rangle$ according to rules 1 of CASE 1. Let the result be g.

3. If $\langle group \rangle$ refers to a primitive relationship of "M" entities Ent then

3.1 If $\langle obj \rangle$ is an entity of type Ent then

3.1.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow o \in g$

3.2 If Ent is uniquely identified by attribute $attr$ and the type of $\langle obj \rangle$ is equal to the type $attr$ then

3.2.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow (\exists!\ e \in g)(ATTRID(e) = o)$

3.3 If the type of $\langle obj \rangle$ is equal to the type of only one attribute $attr$ in Ent then

3.3.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow (\exists\ e \in g)(ATTRID(e) = o)$

4. If $\langle group \rangle$ refers to a primitive relationship of "MO" entities Ent then

4.1 If $\langle obj \rangle$ is an entity of type Ent then

4.1.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow o \in$ **elems** g

4.2 If Ent is uniquely identified by attribute $attr$ and the type of $\langle obj \rangle$ is equal to the type $attr$ then

4.2.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow (\exists!\ e \in$ **elems** $g)(ATTRID(e) = o)$

4.3 If the type of $\langle obj \rangle$ is equal to the type of only one attribute $attr$ in Ent then

4.3.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow (\exists\ e \in$ **elems** $g)(ATTRID(e) = o)$

5. If $\langle group \rangle$ is a second order or high-order relationship from Ent to some other entity or some relationship then

5.1 If $\langle obj \rangle$ is an entity of type ent then

5.1.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow o \in$ **dom** g

5.2 If Ent is uniquely identified by attribute $attr$ and the type of $\langle obj \rangle$ is equal to the type $attr$ then

5.2.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow (\exists!\ e \in$ **dom** $g)(ATTRID(e) = o)$

5.3 If the type of $\langle obj \rangle$ is equal to the type of only one attribute $attr$ in Ent then

5.3.1 $\langle obj \rangle$ **is-in** $\langle group \rangle \Longrightarrow (\exists\ e \in$ **dom** $g)(ATTRID(e) = o)$

CASE 3 For each constraint of the form $\langle obj \rangle$ **not is-in** $\langle group \rangle$ do

Follow the rules of CASE 2 except that

rule 3.1.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow o \notin g$

rule 3.2.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow \sim(\exists!\, \iota \in g)(ATTRID(\iota) = o)$

rule 3.3.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow \sim(\exists\, \iota \in g)(ATTRID(\iota) = o)$

rule 4.1.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow o \notin \textbf{elems}\ g$

rule 4.2.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow \sim(\exists!\, \iota \in \textbf{elems}\ g)$
$(ATTRID(\iota) = o)$

rule 4.3.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow \sim(\exists\, \iota \in \textbf{elems}\ g)$
$(ATTRID(\iota) = o)$

rule 5.1.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow o \notin \textbf{dom}\ g$

rule 5.2.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow \sim(\exists!\, \iota \in \textbf{dom}\ g)$
$(ATTRID(\iota) = o)$

rule 5.3.1 becomes $\langle obj \rangle$ **not is-in** $\langle group \rangle \Longrightarrow \sim(\exists\, \iota \in \textbf{dom}\ g)$
$(ATTRID(\iota) = o)$

CASE 4 For each constraint of the form **if** $\langle conditions \rangle$ **then** $\langle constraints \rangle$
[**else**$\langle constraints \rangle$] **endif** do

1. Transform each condition (separated by **and/or**) in $\langle conditions \rangle$ according to rules of CASE 1, 2, and 3 of the *Constraints* clause. Let *cond* be the result.
2. Transform the $\langle constraints \rangle$ according to the rules for the *Constraints* clause. Let c_1 and c_2 be the result for the **then** and **else** part respectively.
3. **if** $\langle conditions \rangle$ **then** $\langle constraints \rangle$ [**else** $\langle constraints \rangle$] **endif** $\Longrightarrow$
if *(cond)* **then** (c_1) [**else** (c_2)]

CASE 5 For each constraint of the form
for-all *ent* **in** $\langle group \rangle$ **do** $\langle constraints \rangle$ **end-for**

1. Transform $\langle group \rangle$ according to rules 1 of CASE 1 for the *Constraints* clause. Let g be the result.
2. Transform the $\langle constraints \rangle$ according to the rules for the *Constraints* clause. Let *cons* be the result.

3. If $\langle group \rangle$ refers to a primitive relationships of "M" entities then

3.1 **for-all** *ent* **in** $\langle group \rangle$ **do** $\langle constraints \rangle$ **end-for** $\Longrightarrow$

$(\forall ent \in g)$ *(cons)*

4. If $\langle group \rangle$ refers to a primitive relationships of "MO" entities then

4.1 **for-all** *ent* **in** $\langle group \rangle$ **do** $\langle constraints \rangle$ **end-for** $\Longrightarrow$

$(\forall ent \in \textbf{elems}\ g)$ *(cons)*

5. If $\langle group \rangle$ refers to a second-order or high-order relationship then

5.1 **for-all** *ent* **in** $\langle group \rangle$ **do** $\langle constraints \rangle$ **end-for** $\Longrightarrow$

$(\forall ent \in \textbf{dom}\ g)$ *(cons)*

Generating the VDM postconditions

The *Semantics* clause simply describes the effects of the operation on the *Operands* or the system global entities. The actions of the *Semantics* clause are transformed into postconditions in VDM. The KFDs describe **actions** whereas the VDM postcondition clause describe the state of objects assuming the actions have been performed on them. Each action in a KFD is transformed into one or more VDM postconditions according to the following rules. Then the postconditions are combined by the logical connector $\wedge$. In this transformation process, knowledge from the previously generated VDM state is required.

Transformation of each action of the *Semantics* clause

The transformation rules are combined into seven major groups based on the **actions** keywords. The transformation is done by analyzing the parameters of the keyword separately and in the context of the meaning of the keyword. In the following rules, read the arrow $\Longrightarrow$ as **the action is translated in VDM as**.

CASE 1 For each action of the form **add** *object* **in** $\langle group \rangle$ do

1. If *object* is a 2-tuple specified as (e_1, e_2) where e_i is an entity name of type E_i then e_i corresponds to a VDM variable name and *object* corresponds to (e_1, e_2)

in VDM. Otherwise, transform *object* with the rules 1 of CASE 1 for the *Constraints* clause. Let *obj* be the result of transforming *object*.

2. If *group* is specified as a single variable v in the KFD then

2.1 If v : *E-set* in the VDM state and *obj* is of type E then

2.1.1 $\Longrightarrow v' = v \cup \{obj\}$ (if *obj* is an entity)

$\Longrightarrow v' = v \cup obj$ (if *obj* is a set)

2.2 If v : *E-list* in the VDM state and *obj* is of type E then

2.2.1 $\Longrightarrow v' = v \parallel \langle obj \rangle$ (if *obj* is an entity)

$\Longrightarrow v' = v \parallel obj$ (if *obj* is a set)

2.3 If $v : E_1 \rightarrow E_2$ in the VDM state and *obj* is a tuple (ϵ_1, ϵ_2) where ϵ_i is of type E_i then

2.3.1 $\Longrightarrow v' = v \dagger [\epsilon_1 \rightarrow \epsilon_2]$

3. If *group* is specified as a variable $v(\epsilon)$ in the KFD then

3.1 If $v : E \rightarrow E_1$*-set* in the VDM state and *obj* is of type E_1 then

3.1.1 $\Longrightarrow v' = v \dagger [\epsilon \rightarrow (v(\epsilon) \cup \{obj\})]$ (if *obj* is an entity)

$\Longrightarrow v' = v \dagger [\epsilon \rightarrow (v(\epsilon) \cup obj)]$ (if *obj* is a set)

3.2 If $v : E \rightarrow E_1$*-list* in the VDM state and *obj* is of type E_1 then

3.2.1 $\Longrightarrow v' = v \dagger [\epsilon \rightarrow (v(\epsilon) \parallel \langle obj \rangle)]$ (if *obj* is an entity)

$\Longrightarrow v' = v \dagger [\epsilon \rightarrow (v(\epsilon) \parallel obj)]$ (if *obj* is a set)

3.3 If $v : E \rightarrow R$ and $R = E_1 \rightarrow E_2$ in the VDM state and *obj* is a tuple (ϵ_1, ϵ_2) where ϵ_i is of type E_i then

3.3.1 $\Longrightarrow v' = v \dagger [\epsilon \rightarrow (v(\epsilon) \dagger [\epsilon_1 \rightarrow \epsilon_2])]$

4. If *group* is specified as a variable $attr(\epsilon)$ **of** v in the KFD and $v : r$ is defined in the VDM state where r is a record type having fields $attr_1, attr_2, \ldots, attr_i, \ldots, attr_n$ then

4.1 If $attr : E \rightarrow E_1$*-set* in the VDM state and *obj* is of type E_1 then

4.1.1 $\Longrightarrow ATTR(v)' = ATTR(v) \dagger [\epsilon \rightarrow (ATTR(v)(\epsilon) \cup \{obj\})]$

(if *obj* is an entity)

$\Longrightarrow ATTR(v)' = ATTR(v) \dagger [\epsilon \rightarrow (ATTR(v)(\epsilon) \cup obj)]$

(if *obj* is a set)

4.2 If $attr : E \rightarrow E_1\text{-}list$ in the VDM state and obj is of type E_1 then

4.2.1 $\Longrightarrow ATTR(v)' = ATTR(v) \dagger [e \rightarrow (ATTR(v)(e) \parallel \langle obj \rangle)]$

(if obj is an entity)

$\Longrightarrow ATTR(v)' = ATTR(v) \dagger [e \rightarrow (ATTR(v)(e) \parallel obj)]$

(if obj is a set)

4.3 If $attr : E \rightarrow R$ and $R = E_1 \rightarrow E_2$ in the VDM state and obj is a tuple (e_1, e_2) where e_i is of type E_i then

4.3.1 $\Longrightarrow ATTR(v)' = ATTR(v) \dagger [e \rightarrow (ATTR(v)(e) \dagger [e_1 \rightarrow e_2])]$

5. If $group$ is specified as a variable $attr_i$ of $v(e_1)$ in the KFD and $v : E_1 \rightarrow E_2$ is defined in the VDM state where E_2 is a record type having fields $attr_1, attr_2, \ldots, attr_i, \ldots, attr_n$ then

5.1 If $attr_i : E\text{-}set$ is defined in the VDM state and obj is of type E then

5.1.1
$$\begin{aligned} \Longrightarrow v' = v \dagger [e_1 \rightarrow \mathrm{mk}-\mathbf{E_2}(&ATTR_1(v(e_1)),\\ &ATTR_2(v(e_1)),\\ &\vdots\\ &ATTR_i(v(e_1)) \cup \{obj\},\\ &\vdots\\ &ATTR_n(v(e_1)))] \quad \text{(if } obj \text{ is an entity)} \end{aligned}$$

$$\begin{aligned} \Longrightarrow v' = v \dagger [e_1 \rightarrow \mathrm{mk}-\mathbf{E_2}(&ATTR_1(v(e_1)),\\ &ATTR_2(v(e_1)),\\ &\vdots\\ &ATTR_i(v(e_1)) \cup obj,\\ &\vdots\\ &ATTR_n(v(e_1)))] \quad \text{(if } obj \text{ is a set)} \end{aligned}$$

5.2 If $attr_i : E\text{-}list$ is defined in the VDM state and obj is of type E then

5.2.1
$$\begin{aligned} \Longrightarrow v' = v \dagger [e_1 \rightarrow \mathrm{mk}-\mathbf{E_2}(&ATTR_1(v(e_1)),\\ &ATTR_2(v(e_1)),\\ &\vdots\\ &ATTR_i(v(e_1)) \parallel \langle obj \rangle,\\ &\vdots \end{aligned}$$

$$ATTR_n(v(e_1)))] \quad \text{(if } obj \text{ is an entity)}$$

$$\Longrightarrow v' = v \dagger [e_1 \rightarrow \mathbf{mk-E_2}(ATTR_1(v(e_1)),$$
$$ATTR_2(v(e_1)),$$
$$\vdots$$
$$ATTR_i(v(e_1)) \parallel obj,$$
$$\vdots$$
$$ATTR_n(v(e_1)))] \quad \text{(if } obj \text{ is a set)}$$

5.3 If $attr_i : E_3 \rightarrow E_4$ is defined in the VDM state and obj is a tuple (e_3, e_4) where e_i is of type E_i then

5.3.1 $$\Longrightarrow v' = v \dagger [e_1 \rightarrow \mathbf{mk-E_2}(ATTR_1(v(e_1)),$$
$$ATTR_2(v(e_1)),$$
$$\vdots$$
$$ATTR_i(v(e_1)) \dagger [e_3 \rightarrow e_4],$$
$$\vdots$$
$$ATTR_n(v(e_1)))]$$

The previous rules are the major ones used in the transformation process. All other rules can be derived from the previous ones by applying them recursively. For example, if *group* is a variable $v(e_1)(e_2)$ and $v : E_1 \rightarrow R_1$ where $R_1 : E_2 \rightarrow R_2$ and $R_2 : E_3 \rightarrow E_4$, it will be transformed by following recursively the rule 3.3 (assuming *obj* is a tuple (e_3, e_4)).

CASE 2 For each action of the form **remove** *obj* **from** *group* do

The transformation is done according to the rules of CASE 1 for the **add** action except that the following rules are changed.

2.1.1 $\Longrightarrow v' = v - obj$ (*obj* is an entity or a set)

2.2.1 $\Longrightarrow v' = \mathbf{tl}\ v$ (*obj* cannot be a set)

2.3.1 $\Longrightarrow v' = v \setminus \{e_1\}$

3.1.1 $\Longrightarrow v' = v \dagger [e \rightarrow (v(e) - obj)]$ (*obj* is an entity or a set)

3.2.1 $\Longrightarrow v' = v \dagger [e \rightarrow (\mathbf{tl}\ v(e))]$ (*obj* cannot be a set)

3.3.1 $\Longrightarrow v' = v \dagger [e \rightarrow (v(e) \setminus \{e_1\})]$

4.1.1 $\Longrightarrow ATTR(v)' = ATTR(v) \dagger [e \rightarrow (ATTR(v)(e) - obj)]$

(obj is an entity or a set)

4.2.1 $\Longrightarrow ATTR(v)' = ATTR(v) \dagger [e \rightarrow (\mathbf{tl}\ ATTR(v)(e))]$

(obj cannot be a set)

4.3.1 $\Longrightarrow ATTR(v)' = ATTR(v) \dagger [e \rightarrow (ATTR(v)(e)\backslash\{e_2\})]$

5.1.1 $\Longrightarrow$

$$v' = v \dagger [e_1 \rightarrow \mathbf{mk-E_2}(ATTR_1(v(e_1)), ATTR_2(v(e_1)), \vdots, ATTR_i(v(e_1)) - obj, \vdots, ATTR_n(v(e_1)))]$$

(obj is an entity or a set)

5.2.1 $\Longrightarrow$

$$v' = v \dagger [e_1 \rightarrow \mathbf{mk-E_2}(ATTR_1(v(e_1)), ATTR_2(v(e_1)), \vdots, \mathbf{tl}\ ATTR_i(v(e_1)), \vdots, ATTR_n(v(e_1)))]$$

(obj cannot be a set)

5.3.1 $\Longrightarrow$

$$v' = v \dagger [e_1 \rightarrow \mathbf{mk-E_2}(ATTR_1(v(e_1)), ATTR_2(v(e_1)), \vdots, ATTR_i(v(e_1))\backslash\{e_3\}, \vdots, ATTR_n(v(e_1)))]$$

CASE 3 For each action of the form **set** $\langle obj \rangle$ **to** $\langle obj\text{-}value \rangle$ do

1. Transform $\langle obj \rangle$ according to rules 1 of CASE 1 for the *Constraints* clause. Let o be the result.
2. Transform $\langle obj\text{-}value \rangle$ according to rules 3 of CASE 1 for the *Constraints* clause. Let v be the result.
3. $\Longrightarrow o' = v$.

CASE 4 For each action of the form

create ent : ENT **with** $\langle obj_1 \rangle$, $\langle obj_2 \rangle$, ..., $\langle obj_n \rangle$ do

1. Transform each $\langle obj_i \rangle$ according to rules 1 of CASE 1 for the *Constraints* clause, if any. Let obj_i be the result.
2. $\Longrightarrow$ **let** ent = **mk-ENT** $(obj_1, obj_2, \ldots, obj_n)$ **in**

   (... include all the following actions in between parenthesis ...)

CASE 5 For each action of the form **return** ent [**such-that** $\langle constraints \rangle$] do

1. Transform $\langle constraints \rangle$ according to the rules for the *Constraints* clause. Let *cons* be the result.
2. Change the VDM operation header by creating a variable *r_ent* of type T that will be returned by the operation. If the operation name is *oper* then the operation header will look like :

   oper (...) r_ent : T

   where T is the type specified in the *Operands* section on the RHS of the arrow.
3. If *ent* is a *Unique-entity* then

3.1 $\Longrightarrow$ $(\exists!\ ent)($ ... include *cons* in parenthesis ... $\wedge\ r\_ent' = ent)$

4. If *ent* is not a *Unique-entity* then

4.1 $\Longrightarrow$ $(\exists\ ent)($ ... include *cons* in parenthesis ... $\wedge\ r\_ent' = ent)$

CASE 6 For each action of the form **if** $\langle conditions \rangle$ **then** $\langle actions \rangle$

[**else**$\langle actions \rangle$] **endif** do

1. Transform each condition (separated by **and/or**) in $\langle conditions \rangle$ according to rules of CASE 1, 2, and 3 of the *Constraints* clause. Let *cond* be the result.
2. Transform the $\langle actions \rangle$ according to the rules for the *Semantics* clause. Let a_1 and a_2 be the result for the **then** and **else** part respectively.
3. **if** $\langle conditions \rangle$ **then** $\langle actions \rangle$ [**else** $\langle actions \rangle$] **endif** $\Longrightarrow$

   if *(cond)* **then** (a_1) [**else** (a_2)]

CASE 7 For each action of the form

for-all *ent* **in** $\langle group \rangle$ **do** $\langle actions \rangle$ **end-for**

1. Transform $\langle group \rangle$ according to rules 1 of CASE 1 for the *Constraints* clause. Let g be the result.
2. Transform the $\langle actions \rangle$ according to the rules for the *Semantics* clause. Let *act* be the result.
3. If $\langle group \rangle$ refers to a primitive relationships of "M" entities then

3.1 **for-all** *ent* **in** $\langle group \rangle$ **do** $\langle actions \rangle$ **end-for** $\Longrightarrow$

$$(\forall ent \in g)\ (act)$$

4. If $\langle group \rangle$ refers to a primitive relationships of "MO" entities then

4.1 **for-all** *ent* **in** $\langle group \rangle$ **do** $\langle actions \rangle$ **end-for** $\Longrightarrow$

$$(\forall ent \in \textbf{elems}\ g)\ (act)$$

5. If $\langle group \rangle$ refers to a second-order or high-order relationship then

5.1 **for-all** *ent* **in** $\langle group \rangle$ **do** $\langle actions \rangle$ **end-for** $\Longrightarrow$

$$(\forall ent \in \textbf{dom}\ g)\ (act)$$

In section 5.5.2. we have presented algorithms to generate the VDM state and operations. Different algorithms have been developed to generate the **external**, **pre**condition, and **post**condition clauses of an operation. Invariants, which are integral parts of a VDM specification, need to be studied. Since invariants can be viewed as predicates applied on values of the state variables, we believe a method similar to the description of *Constraints* can be used.

If we apply the algorithms presented here to the KFDs : the mailing system example, we obtain the VDM operations shown in figures 5.18 and 5.19. If we compare them with the specifications given in [Cohen'85], we find that they are very similar. Operations POST, READ, COLLECT, ADDALIAS, and DELALIAS are equivalent to the specifications in [Cohen'85]. The only difference is that they use **if** a **then** b **else** *false* in the **pre**condition clause where our system generates $a \wedge b$. The specification generated for operation CLEAR is written in a simpler way than in [Cohen'85] but it can be proven that both are logically equivalent.

POST (USERX: *User*, MAILX: *Mailitem*)
ext DESKR: **wr** *Deskring*
pre TO(mailx) ≠ ⟨⟩ ∧ SUBJECT(mailx) ≠ "" ∧
BODY(mailx) ≠ "" ∧ WHENSENT(mailx) = "" ∧
REFNO(mailx) = NIL ∧ userx ∈ **dom** deskr
post desk' = deskr † [userx → mk-Trays(IN(deskr(userx)),
OUT(deskr(userx)) ∪ {mailx},
PEND(deskr(userx)))]

READ (USERX: *User*, MAILID: *Ref*) R_MAILX: *Mailitem*
ext DESKR: **rd** *Deskring*
pre userx ∈ **dom** deskr ∧
(∃! m ∈ PEND(deskr(userx))) (REFNO(m) = mailid)
post (∃! mailx)(mailx ∈ PEND(deskr(userx)) ∧ REFNO(mailx) = mailid ∧ r_mailx' = mailx)

COLLECT (USERX: *User*, MAILID: *Ref*) R_MAILX: *Mailitem*
ext DESKR: **wr** *Deskring*
pre userx ∈ **dom** deskr ∧
(∃! m ∈ IN(deskr(userx))) (REFNO(m) = mailid)
post (∃! mailx)(mailx ∈ IN(deskr(userx)) ∧ REFNO(mailx) = mailid ∧ r_mailx' = mailx) ∧
desk' = deskr † [userx → mk-Trays(IN(deskr(userx)) - mailx,
OUT(deskr(userx)),
PEND(deskr(userx)))])

ADDALIAS (USERX: *User*, NAMEY: *Name*, USERY: *User*)
ext DIRECT: **wr** *Directory*
pre userx ∈ **dom** direct ∧
namey ∉ **dom** direct(userx) ∧
usery ∈ **dom** direct
post direct' = direct † [userx → (direct(userx) † [namey → usery])]

Figure 5.18: Generated VDM operations for the mailing system(1)

```
DELALIAS (USERX: User, NAMEY: Name, USERY: User)
ext     DIRECT wr Directory
pre     userx ∈ dom direct ∧
        namey ∈ dom direct(userx) ∧
        direct(userx)(namey) = usery
post    direct' = direct † [userx → (direct(userx) \ {namey})]

CLEAR (USERX: User)
ext     DIRECT rd Directory
        DESKR: wr Deskring
pre     userx ∈ dom deskr ∧
        OUT(deskr(userx)) ≠ {}
post    (∀ mailx ∈ OUT(deskr(userx))
         (let newmail = mk-Mailitem (TO(mailx), CC(mailx), FROM(mailx), SUBJECT(mailx),
                                     REPLY_REQ(mailx), REFS(mailx), now(), newid(), BODY(mailx)) in (
         (∀ namei ∈ elems TO(mailx))
          (deskr' = deskr † [direct(userx)(namei) → mk-Trays(IN(deskr(direct(userx)(namei))) ∪ {newmail},
                                                             OUT(deskr(direct(userx)(namei))),
                                                             PEND(deskr(direct(userx)(namei))))] ∧
           if (REPLY_REQ(mailx))
           then (deskr' = deskr † [direct(userx)(namei) → mk-Trays(IN(deskr(direct(userx)(namei))),
                                                                  OUT(deskr(direct(userx)(namei))),
                                                                  PEND(deskr(direct(userx)(namei))) ∪ {newmail})] ) ) ∧
         (∀ namej ∈ elems CC(mailx))
          (deskr' = deskr † [direct(userx)(namej) → mk-Trays(IN(deskr(direct(userx)(namej))) ∪ {newmail},
                                                             OUT(deskr(direct(userx)(namej))),
                                                             PEND(deskr(direct(userx)(namej))))] ) ∧
         (∀ m ∈ PEND(deskr(userx)))
          (if (REFNO(m) ∈ REFS(mailx))
           then (deskr' = deskr † [userx → mk-Trays(IN(deskr(userx)),
                                                    OUT(deskr(userx)),
                                                    PEND(deskr(userx)) - m)] ) ) ) ) ∧
        OUT(deskr(userx))' = {}
```

Figure 5.19: Generated VDM operations for the mailing system(2)

Chapter 6

Conclusion and Future Work

6.1 Comparison of our Approach with Others

In [Fraser'91] the authors have attempted an automated approach to transform an informal specification based on structured analysis to a formal specification based on VDM. They use DFDs along with decision tables as means of informal specifications. The method they provide is only partially automatable since their rules do not take into account the abstract control flows which are not explicitly present in the conventional data flow diagrams. On the other hand the VDM specification has constructs such as sequence, decision and iteration to represent the control flows. The authors put a heavy responsibility on the part of analysts to identify the implicit control structure, if any, present in the informal specification. Apart from this, their informal specification which is based on decision tables is very "close" to the generated formal specification: thus mostly requiring pure syntactic manipulations to generate the VDM specifications.

In contrast, our KFD is not only easier for user to describe but also has explicit control constructs such as *if-then-else* and *for-all* to represent the control flows which are needed for describing operations at some lower level of abstraction. Also, the KFD descriptions provide flexibility by allowing partial descriptions. The partial descriptions are resolved at a later stage in the transformation process in one of three ways: use of the domain knowledge, use of the contextual information, or through interactive dialogue with the systems analyst. A study comparing the ease of use of

KFDs in comparison to VDM is to be carried out with software practitioners.

In [Plat'91] several strategies are discussed for mapping DFDs onto VDM constructs. For the sake of transformation what is called "low level DFD" are constructed. Our work, like [Plat'91] makes use of the knowledge of the problem domain; but our KFDs have no direct counterparts. Moreover, we believe, that successive refinement of DFDs to generate low level DFDs is a much more difficult job for a systems analyst than to specify KFDs. This remains to be tested through experiments in the future.

In [Dick'91], the authors describe a methodology similar to ours in certain aspects such as data modeling and rule-based transformation. In contrast, our MER embodies built in entities which form part of the problem domain knowledge in our knowledge base. Additionally, the inheritance structure we provide facilitates efficient reusability of built-in entity types, thereby helping the user to write the specification easier. The main difference in our approach resides in the way the operations are specified. We feel that our KFD has much more expressive power than the approach described in [Dick'91]. Our KFDs are more natural to a user's conceptual thinking than the forced upon state-based model as proposed by them.

In [Yonezaki'89] a methodology based on Montague grammar is presented to translate an informal specification in restricted natural language to a formal specification based on modal logic. The basic principle is to describe the semantics of each word, in natural language using less ambiguous and simpler words. The methodology is very complex and computation intensive. Also, it is necessary to describe the complete semantics of each complex word in the informal specification at subsequent levels of iteration, thus demanding a lot of effort from the user.

In [Neighbors'84], the author proposes a transformation scheme for constructing software program components from reusable abstract descriptions. Although, the methodology in general seems to be similar, the paper does not describe sufficiently the transformation process to compare with our work.

The work on Requirements Apprentice by Reubenstein et.al. [Reubenstein'91a] and our work have two things in common, namely, the need to deal with the side effects of informality and the interactive dialog with the systems analyst. The focus of their work is in requirements elicitation whereas our focus is in transformation to VDM. One way to deal with the side effects of informality is to use the "reasoning techniques" from AI. This is discussed in [Reubenstein'91b] and also in [Mirivala'91a]. The latter uses analogy based reasoning to handle certain sources of informality. Here again the informal specification is obtained from the systems analyst interactively. The informal specification is represented in the form of a structure tree. Its leaf nodes contain data objects and non-leaf nodes contain relations. The structure tree forms an input to the analogical reasoner. Based on such reasoning they show how the side effects of informality such as poor ordering, incompleteness, redundancy, and errors may be handled.

6.2 Conclusion

One of the contributions of this thesis is the development of a methodology for describing user requirements. To suit the two stages of data modeling and the specification of operations, we have developed the MER and KFDs respectively. The graphical aspects of the MER and the "English-like" keywords of the KFDs, we believe would make it easier for the user to express the requirements. The fixed format and the fixed set of keywords of KFDs and the MER semantics possess the necessary qualities for automating the transformation process.

Another contribution of this thesis is the set of algorithms developed to generate a VDM specification from the input MER and KFDs. Based on these algorithms, a software system has been implemented to show the feasibility of the automation of the transformation process. Two example systems (mailing system and library system) are provided to which the transformation process was applied. By integrating common knowledge in the preprocessing stage of the transformation, it was demonstrated that certain kinds of incompleteness can be accepted in the input description and resolved by the transformation system. There is a potential to use different sources of knowledge such as common knowledge about sets and functions and domain knowledge in the transformation process. This is the key to make a semi formal description technique to be less constraining on the user wherein the missing information is automatically inserted and inconsistencies resolved (whenever possible).

Not all software professionals are comfortable in writing formal specifications. The MER and KFDs proposed here are much easier to learn and use than the VDM specification language. Through the mailing system example presented in this thesis, we have shown the ease of use of MER and KFD. The automatic generation of VDM specification relieves the user from having to learn formal notations. In that sense, the work reported in this thesis is a step towards bridging the gap between informal and formal requirements specifications.

6.3 Suggested Future Work

The work presented in this thesis is a starting point of a large project. At this stage, there is a need for more work to realize the complete picture. Some of the important aspects left to be done and necessitating further analysis are listed below:

- The two examples, namely the mailing system and the library system, are not sufficient to show the general applicability of the approach presented here. The MER and KFDs should be used on many other test cases and their shortcomings, if any, should be discovered.

- Generation of invariants in the VDM specification needs to be studied. The necessary information for this purpose could come from the domain expert, possibly in the form of constraints specifications.

- The MER and KFD input tools are not fully implemented yet. The two editors should be able to refer to predefined entity types and high-order keywords while constructing the description of requirements.

- A scanner and a parser of the input description are required to verify the syntax and the semantics of the KFDs. Preferably, methods to integrate knowledge at this point would be convenient. As the description is parsed, checks for incompleteness and inconsistency could be applied and corrected, if possible. Otherwise, the interactive system should be able to request the user to clarify the encountered ambiguities. The result generated by the parsing stage should be the MER and KFDs encoded in the form of tables, as presented in section 5.4.3.

- In our transformation process, the knowledge about the target formal specification language is embodied in the procedures. Methods to separate this knowledge from the transformation procedures would be an important step towards making the system more adaptable. Then, the system would not be dependent on any formal specification language. Studies in this direction could justify the use of such a system in the requirements analysis phase.

- The VDM specification generated by the present system is not guaranteed to be the best description. A VDM post-optimizer could take as input the generated VDM specification and produce a more readable specification.

Bibliography

[Alagar'91] V.S. Alagar. *Lecture notes for COMP 648*. Concordia University, 1991.

[Balzer'78] Robert Balzer, Neil Goldman and David Wile. "Informality in Program Specifications", *IEEE Transactions on Software Engineering*, Vol. 4, pp. 94-103, March 1978.

[Balzer'83] Robert Balzer, Thomas E. Cheatham, Cordell Green, "Software Technology in the 1990's: Using a New Paradigm", *IEEE Computer Magazine*, pp. 39-45, November 1983.

[Chen'80] P. P. Chen, *Entity-Relationship approach to system analysis and design*, North Holland, Mass., 1980.

[Cohen'85] B. Cohen, W.T. Harwood, M.I. Jackson, *The Specification of Complex Systems*. Addison-Wesley publishing co., 1985.

[D'Almeida'92] Juliette D'Almeida, R. Achuthan, T. Radhakrishnan, V.S. Alagar, "Transformation of a Semi-formal Specification to VDM", in *Proceedings of 7^{th} Knowledge-based Software Engineering Conference*, Sept. 1992.

[Desai'90] Bipin C. Desai, *An Introduction to Database Systems*. West Publishing co., 1990.

[Dick'91] Jeremy Dick, Jérôme Loubersac, "Integrating Structured and Formal Methods: A Visual Approach to VDM", in *Proceedings of European Software Engineering Conference*, 1991.

[Neighbors'84] James M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components", *IEEE Transactions on Software Engineering*, Vol. 10, pp. 564-574, Sept. 1984.

[Fraser'91] Martin D. Fraser, Kuldeep Kumar, Vijay K. Vaishnavi, "Informal and Formal Requirements Specification Languages: Bridging the Gap", *IEEE Transactions on Software Engineering*, Vol. 17, pp. 151-166, May 1991.

[Gao'92] GAO Junming, "GAP — A Tool for Transforming From VDM Specification Into Object-Oriented Design", *Master's Thesis*, Department of Computer Science, Concordia University, April 1992.

[Johnson'92] W. Lewis Johnson, Martin S. Feather, David R. Harris, "Representation and Presentation of Requirements Knowledge", *IEEE Transactions on Software Engineering*, Vol. 18, pp. 853-869, Oct. 1992.

[Jones'90a] C.B. Jones, R.C. Shaw, *Case studies in Systematic Software Development*, Prentice-Hall International, 1990.

[Jones'90b] C.B. Jones, *Systematic software development using VDM*, Englewood Cliffs, NJ: Prentice-Hall International, 1990.

[Jones'91] C.B. Jones, K.D. Jones, P.A. Lindsay, R. Moore, *Mural: A Formal Development Support System*, Springer-Verlag, 1991.

[Loucopoulos'89] P. Loucopoulos, R. E. M. Champion, "Knowledge-based support for requirements engineering", *Information and Software Technology*, Vol. 31, pp. 123-135, April 1989.

[Miriyala'91a] Kanth Miriyala, "Intelligent assistance for formalizing software specification", Technical report UIUCDCS-R-91 1702, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1991.

[Miriyala'91b] Kanth Miriyala and Mehdi T. Harandi, "Automatic derivation of formal software specifications from informal descriptions", *IEEE Transactions on Software Engineering*, Vol. 17, pp. 1126-1142, Oct. 1991.

[Nguyen] Mai Lan Nguyen, "A knowledge-based graph editor", *Master's Thesis, in progress*, Department of Computer Science, Concordia University.

[Neighbors'84] James M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components", *IEEE Transactions on Software Engineering*, Vol. 10, pp. 564-574, Sept. 1984.

[Plat'91] Nico Plat, J. W. Katwjik, Kees Pronk, "A case for structured analysis/formal design", *VDM'91, Formal software development methods*. Eds. S.Prehn, W. J.Toetenel, Berlin: Springer-verlag, 1991.

[Reubenstein'91a] Howard B. Reubenstein, Richard C. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition", *IEEE Transactions on Software Engineering*, Vol. 17, pp. 226-240, March 1991.

[Reubenstein'91b] Howard B. Reubenstein, "Automated acquisition of evolving informal descriptions", Technical Report AI-TR 1205, MIT Artificial Intelligence Laboratory, June 1991.

[Rich'86] Charles Rich, Richard C. Waters, "Artificial Intelligence and Software Engineering", *Readings in Artificial Intelligence and Software Engineering*. Eds. Morgan Kaufman, Los Altos, CA, 1986.

[Sommerville'89] Ian Sommerville, *Software Engineering*, Addison-Wesley, Wokingham, England, 1989.

[Sowa'84] J. F. Sowa, *Conceptual structures: Information processing in mind and machine*, Addison-Wesley, Wokingham, UK, 1984.

[Wing'91] Jeannette M. Wing, "Unintrusive ways to integrate formal specifications in practice", in *VDM'91, Formal software development methods*, Eds. S.Prehn, W. J.Toetenel, Berlin: Springer-verlag, 1991.

[Yonezaki'89] Naoki Yonezaki, "Natural language interface for requirements specification", *Japanese Perspective in Software Engineering*, Ed. Matsumoto, Addison Wesley publishing co., 1989.

Appendix A

Library System

The library system example is taken form [Alagar'91].

A.1 Informal Specification

- A library management system will deal with the two major entities *users* or *borrowers* and *books*.
- All books must have been entered in the data base. A book can have multiple copies: however, every copy has a unique call number.
- It is assumed that every book acquired by the library will never be lost, meaning that it will be either in stack or loaned out.
- All users must have been registered in order to use the facilities of the library. Every user has a unique I.D. number.
- A faculty member can borrow a maximum of 20 books while the maximum limit for a student user is 10.
- Users can reserve books. If more than one user reserves the book, the users' requests are queued. All users have equal privileges in reserving a book.
- A user can renew a book if it is already loaned to that user and not requested by any other user.

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

From the given requirements, we can identify the following functionalities which must be provided by the library system: initialize the library management system, add a user, add a book, borrow a book, return a book, reserve a book, and renew a book. In this example, we do not consider other operations such as deleting a book from the database, deleting a user from the library system, dealing with "overdue" books, and shelving the reserved books separately.

A.2 The semi-formal description

A.2.1 The MER Model

The MER diagram of the library system is shown in figure A.1. As in the mailing system, the predefined entity types *Date* and *Unique-entity* are included in the model. Only one system entity is required to describe the library system. The system variable will be called *lib*. It is an instance of entity type *Library*.

A.2.2 The KFDs

Operation name:	INIT_LIB
Operands:	none
Syntax:	*init_lib*
Constraints:	none
Semantics:	*set* Registered-users *of* lib *to empty*
	set Catalogued-books *of* lib *to empty*
	set Loans *of* lib *to empty*
	set Reservations *of* lib *to empty*
Description:	Initialize the library system.

Operation name:	ADD_BOOK
Operands:	Text, Text

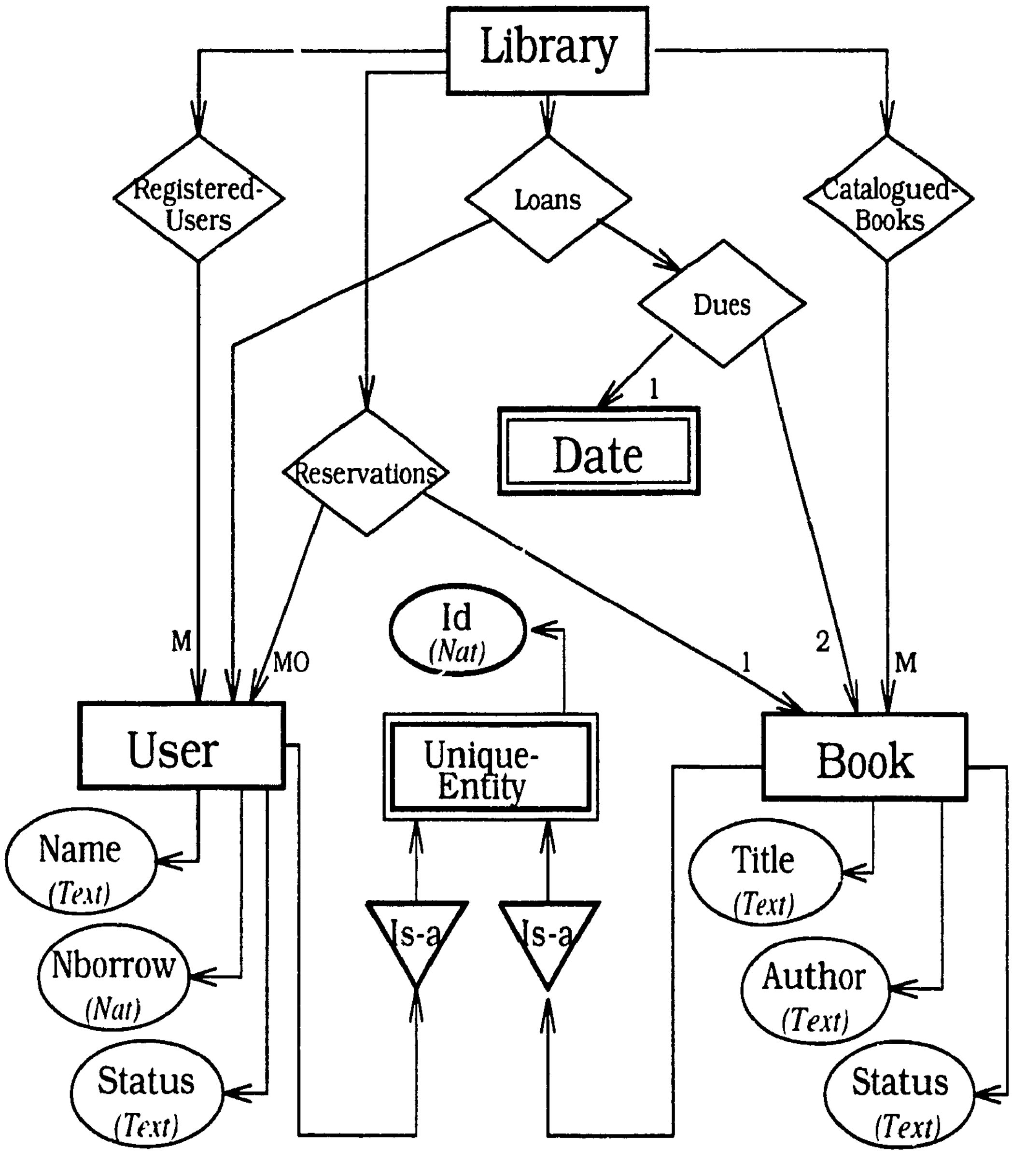


Figure A.1: The MER diagram of the library system

Syntax:	*add_book* titlex, authorx
Constraints:	none
Semantics:	*create* bookx : Book *with newid()*, titlex, authorx, "inshelf" *add* bookx *in* Catalogued-books *of* lib
Description:	Add a new book with title "titlex" and authors "authorx" to the collection of library books. The status of a book is initially in stack.

Operation name:	ADD_USER
Operands:	Text, Text
Syntax:	*add_user* namex, statusx
Constraints:	statusx *is* "faculty" *or* statusx *is* "student"
Semantics:	*create* userx : User *with newid()*, namex, 0, statusx *add* userx *in* Registered-users *of* lib
Description:	Add a user with name "namex" with "statux" in the set of users of the library. Users are classified into two categories (status): faculty or student.

Operation name:	BORROW
Operands:	User, Book
Syntax:	userx *borrow* bookx

Constraints:	userx *is-in* Registered-users *of* lib *and*
	bookx *is-in* Catalogued-books *of* lib *and*
	if Status *of* userx *is* "faculty" *and*
	  userx *is-in* Loans *of* lib *then*
	    Number-of Loans(userx) *of* lib < 20
	end-if and
	if Status *of* userx *is* "student" *and*
	  userx *is-in* Loans *of* lib *then*
	    Number-of Loans(userx) *of* lib < 10

end-if and

if bookx *is-in* Reservations *of* lib *then*

first-of Reservations(bookx) *of* lib *is* userx *or*

Reservations(bookx) *of* lib *is empty*

end-if and

Status *of* bookx *is* "inshelf"

Semantics: *add* (bookx, *now()*) *in* Loans(userx) *of* lib

set Status *of* bookx *to* "loaned-out"

remove userx *from* Reservations(bookx) *of* lib

Description: "Userx" borrows "bookx". A faculty member can borrow a maximum of 20 books while the maximum limit for a student if 10. There should not exist any reservations for that book unless the first user in the queue of reservations is "userx".

Operation name: RESERVE

Operands: User, Book

Syntax: userx *reserve* bookx

Constraints: userx *is-in* Registered-users *of* lib *and*

bookx *is-in* Catalogued-books *of* lib *and*

Status *of* bookx *is not* "inshelf"

Semantics: *add* (bookx, userx) *in* Reservations *of* lib

Description: "Userx" reserves "bookx". If more than one user reserve the book, the user's request are queued on a FIFO basis.

Operation name: RENEW

Operands: User, Book

Syntax: userx *renew* bookx

Constraints: userx *is-in* Registered-users *of* lib *and*

bookx *is-in* Catalogued-books *of* lib *and*

if (bookx *is-in* Reservations) *then*

Reservations(bookx) *of* lib *is empty end-if and*

bookx *is-in* Loans(userx) *of* lib

Semantics: *add* (bookx, *now()*) *in* Loans(userx) *of* lib

Description: "Userx" renews "bookx". "Bookx" is renewable if it has been previously loaned out to that user and there is no current reservations for that book.

Operation name: RETURN

Operands: User, Book

Syntax: userx *return* bookx

Constraints: userx *is-in* Registered-users *of* lib *and*

bookx *is-in* Catalogued-books *of* lib *and*

userx *is-in* Loans *of* lib *and*

bookx *is-in* Loans(userx) *of* lib

Semantics: *remove* bookx *from* Loans(userx) *of* lib

set Status *of* bookx *to* "inshelf"

Description: "Userx" returns "bookx". The book must have been previously loaned out by that user.

A.3 The generated VDM specification

A.3.1 The Generated VDM State

State ::

LIB	: *Library*		
Library	:: REGISTERED-USERS	:	*User-set*
	LOANS	:	*User* → *Dues*
	RESERVATIONS	:	*Book* → *User-list*
	CATALOGUED-BOOKS	:	*Book-set*
Dues	= *Book* → *Date*		
User	:: ID	:	*Nat*
	NAME	:	*Text*
	NBORROW	:	*Nat*
	STATUS	:	*Text*
Book	:: ID	:	*Nat*
	TITLE	:	*Text*
	AUTHORS	:	*Text*
	STATUS	:	*Text*
Unique-entity	= *Nat*		
Date	= *Text*		

A.3.2 The Generated VDM operations

INIT-LIB

ext LIB: **wr** *Library*

pre None.

post REGISTERED-USERS(lib)' = {} ∧

CATALOGUED-BOOKS(lib)' = {} ∧

LOANS(lib)' = {} ∧

RESERVATIONS(lib)' = {}

ADD BOOK (TITLEX: *Text*, AUTHORX: *Text*)

ext LIB: **wr** *Library*

pre None.

post **let** bookx = **mk-Book** (*newid()*, titlex, authorx, "inshelf") **in** (

CATALOGUED-BOOKS(lib)' = (CATALOGUED-BOOKS(lib) ∪ {bookx})

ADD-USER (NAMEX: *Text*, STATUSX: *Text*)

ext LIB: **wr** *Library*

pre statusx = "faculty" ∨

statusx = "student"

post **let** userx = **mk-User** (*newid()*, namex, 0, statusx) **in** (

REGISTERED-USERS(lib)' = REGISTERED-USERS(lib) ∪ {userx})

BORROW (USERX: *User*, BOOKX: *Book*)

ext LIB: **wr** *Library*

pre userx ∈ REGISTERED-USERS(lib) ∧

bookx ∈ CATALOGUED-BOOKS(lib) ∧

if (STATUS(userx) = "faculty" ∧

userx ∈ **dom** LOANS(lib)) **then**

(**card** LOANS(lib)(userx) < 20) ∧

if (STATUS(userx) = "student" ∧

userx ∈ **dom** LOANS(lib)) **then**

(**card** LOANS(lib)(userx) < 10) ∧

if (bookx ∈ RESERVATIONS(lib)) **then**

(**hd** RESERVATIONS(lib)(bookx) = userx ∨

RESERVATIONS(lib)(bookx) = ⟨⟩) ∧

STATUS(bookx) = "inshelf"

post LOANS(lib)' = LOANS(lib) † [userx →

(LOANS(lib)(userx) † [bookx → *now()*])] ∧

(STATUS(bookx) = "loaned-out" ∧

if (bookx ∈ RESERVATIONS(lib)) **then**

(RESERVATIONS(lib)' = RESERVATIONS(lib) † [bookx →
(**tl** RESERVATIONS(lib)(bookx))])

RESERVE (USERX: *User*, BOOKX: *Book*)

ext LIB: **wr** *Library*

pre userx ∈ REGISTERED-USERS(lib) ∧
bookx ∈ CATALOGUED-BOOKS(lib) ∧
STATUS(bookx) ≠ "inshelf"

post RESERVATIONS(lib)' = RESERVATIONS(lib) † [bookx →
(RESERVATIONS(lib)(bookx) || ⟨userx⟩)]

RENEW (USERX: *User*, BOOKX: *Book*)

ext LIB: **wr** *Library*

pre userx ∈ REGISTERED-USERS(lib) ∧
bookx ∈ CATALOGUED-BOOKS(lib) ∧
if (bookx ∈ **dom** RESERVATIONS(lib)) **then**
(RESERVATIONS(lib)(bookx) = {})
bookx ∈ **dom** LOANS(lib)(userx)

post LOANS(lib)' = LOANS(lib) † [userx →
(LOANS(lib)(userx) † [bookx → *now()*])]

RETURN (USERX: *User*, BOOKX: *Book*)

ext LIB: **wr** *Library*

pre userx ∈ REGISTERED-USERS(lib) ∧
bookx ∈ CATALOGUED-BOOKS(lib) ∧
userx ∈ **dom** LOANS(lib) ∧
bookx ∈ **dom** LOANS(lib)(userx)

post LOANS(lib)' = LOANS(lib) † [userx → (LOANS(lib)(userx) \ {bookx})] ∧
STATUS(bookx) = "inshelf"