Transformation of a Semi-formal Specification to VDM

Juliette D'Almeida, R. Achuthan, T. Radhakrishnan, V. S. Alagar
Department of Computer Science
Concordia University
Montréal, Canada, H3G 1M8

Abstract

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. Specifications written in a VDM like language are considered formal. In between these two ends we recognize several techniques for semi-formal specifications. In this paper we propose a technique for semi-formal specification. Through an example we present a knowledge assisted transformation process that can generate a formal specification from the given semi-formal specification. The formal specification language chosen is VDM.

1 Introduction

Considerable research is being done to introduce formal methods in all the stages of the software development process; yet it is not well accepted in practice. One reason for this reluctance to accept is in the extensive use of mathematical symbols, their strict semantic interpretations and their terse textual nature which do not provide a good medium of communication among the software practitioners. Also, the common methodologies prevailing in the industry, especially in the early stages of the system conception, make use of graphical notations and informal descriptions. The formal methods to be used at early stages should make use of such informal descriptions as well.

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. The informal means of description has the advantage of being easy to understand for the user, facilitating the requirements elicitation process while the formal specification provides conciseness, precision and a mathematical basis for automated program synthesis. The introduction of formal methods should not in any way be targeted to replace the prevailing informal methods. 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 may lead to error if attempted by one not well versed in formalism. This has spurred a growing interest, in the recent years, in automated supports to assist the derivation of the formal specification of software from its informal descriptions. Tools that assist during the generation and subsequent stages of formal specification are essential to promote the wide acceptance of formal methods.

In the recent past, several attempts have been made in bridging the gap between the informal description and the formal specification. As remarked in [1], the side effects of being informal are: ambiguity, poor ordering of requirements, contradiction, incompleteness or partial specification, and inaccuracy. An informal specification, on the other hand, comes naturally to the user and the methods used are relatively easy to learn. We view formality and informality, expressed through a natural language, as two opposite ends of a spectrum, between which there are several semi-formal methods of specifying the requirements.

In this paper, we describe a semi-formal medium to express requirements which we believe is easy to learn. A user develops the specification interactively with our system. As soon as the required information for any formal "module" is interactively obtained from the user, an automatic transformation will be triggered. Thus, the formalization is a part by part process. As in [8], we distinguish two stages of formalization: data modeling and specification of operations. For each stage, our system provides a distinct tool. It consists of a graphical component called MER diagram and a textual component called KFD.

The MER or Modified Entity-Relationship diagram is based on the well known Entity Relationship approach to data modeling [2]. The modification introduced is essentially for the purposes of "abstraction".
The KFD or Keyword based Formatted Description is a set of textual descriptions given according to a fixed format using keywords. The keywords could be selected suitably for each domain. The KFDs together describe the various operations performed in the problem domain and the MER diagrams describe the entities, attributes, and relationships at various levels of abstraction. The MER diagrams and KFDs, introduced in section 2, form an input to the transformation described in this paper. The transformation process, presented in section 3 produces the corresponding formal specification in VDM (Vienna Development Method [7]). We expect that a user will develop the MER and KFD inputs interactively using appropriate software tools. The transformation process will be capable of disambiguating or providing the missing links, based on certain stored knowledge or by soliciting additional user inputs. Although not discussed in this paper, our ultimate goal is to execute the transformed VDM formal descriptions, locate any errors or shortcomings, and link the discovered shortcomings back to the MER and KFDs which are the actual user inputs.

2 Semi-formal Input

The MER and KFD will be presented through the mailing system example abstracted from [3]. The system introduced in [3] being too large to present here, we restrict ourselves to a subset of that system. The informal statement of the requirements are as follows:

Each user is provided with a personal terminal which serves as an electronic desk where documents can be filed and transmitted to other desks. Additionally, a user has a list of aliases associating names to users in the system. Hence, the system can be regarded as a network of desks, where each desk can send and receive documents to and from every other desk, and as a directory of users' aliases. As part of an electronic desk associated with a user is set of trays. The out-tray contains documents waiting to be sent. The in-tray contains incoming documents transmitted by other users. The pending tray contains copies of incoming documents, for which replies are required. For each mail item, a header will be generated when passing the item from the pad to the out-tray. The user's pad is considered as the input/output device where the user reads/writes a mail. 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 [3]). 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.

From this informal description, we need to identify the system entities and the relationships among entities in order to produce the MER model. The MER model derived from this description is given in Fig. 1.

2.1 Entities and Entity Sets

An entity set (or entity type) is a group of similar entities characterized by a set of properties which are attributes and/or relationships among other entity types. An entity is an instance of an entity type. In the MER diagram, a rectangle represents an entity set. The properties of an entity set (i.e. what defines it) are determined by all its outgoing arrows. An arrow leaving the entity set points to either an attribute (ellipse), a relationship (diamond) or the is-a relationship (triangle). The basic distinction between an entity in the E-R and MER data model is in their level of abstraction. E-R entities are basic units while a higher order MER entity is described by means of other entities and relationships, and hence abstracts the concept of entity association in their definition.

In the MER diagram of the mailing system shown in Fig. 1, we identify seven user-defined entity types and one predefined type called \textit{Unique-entity}. Predefined entities are concepts known to the system which can be used in the MER model to define new entities. Each predefined entity is a piece of domain or common knowledge stored in a knowledge-base and made available to the user. Since predefined entity types are known to the system as domain knowledge,
reasoning can be done about entities of a predefined type or about entities of any user-defined type inheriting from a predefined entity type through the is-a relationship.

2.2 Relationships

In the real world, a relationship is an abstract concept. If we want to model and to refer to a relationship between entities, we need an abstract data or object that records the relationship. For that reason, a relationship is part of the definition of a higher order abstract entity. Consider the relationship Traylist in Fig. 1. Connected to the relationship diamond are entity types User, Trays and Deskring. Moreover, the direction of the arrows indicates that the association between entity types User and Trays is part of the definition of entity type Deskring. Each entity from Deskring is defined by a relationship set Traylist. Deskring in this case is considered a higher order abstract entity type. For any relationship set R, R must be part of the definition of a higher order abstract entity or is part of the association of another relationship. Distinctly, we can differentiate three types of relationships:

**Primitive** A primitive relationship \( R \) defined in a higher order abstract entity \( E_1 \) to an entity type \( E_2 \) is described as a set of entities of type \( E_2 \) associated to each entity of type \( E_1 \). An instance of \( R \) would be a set \( \{e_{21}, e_{22}, \ldots, e_{2n}\} \) where an entity \( e_{2i} \) is of type \( E_2 \). In Fig. 1, In, Out, Pend, Refs, To, Cc are primitive relationships.

**Second-order** A second-order relationship \( R \) defined between two entity types \( E_1 \) and \( E_2 \) is: 1) a set of 2-tuples \( (e_{1i}, e_{2j}) \) where \( e_{1i} \) is an entity of type \( E_1 \) and \( e_{2j} \) is an entity of type \( E_2 \) [one-to-one association between \( E_1 \) and \( E_2 \)], 2) a set of 2-tuples \( (e_{1i}, s_i) \) where \( e_{1i} \) is an entity of type \( E_1 \) and \( s_i \) is a set of entities of type \( E_2 \) [one-to-many association between \( E_1 \) and \( E_2 \)]. Alias and Traylist are second-order relationships, in Fig. 1.

**Higher-order** A higher-order relationship \( R \) defined between an entity type \( E_1 \) and another relationship \( R_1 \) is a set of 2-tuples \( (e_{1i}, r_{1j}) \) where \( e_{1i} \) is an entity of type \( E_1 \) and \( r_{1j} \) is a relationship set which is an instance of \( R_1 \) [one-to-many association between \( E_1 \) and the entities in \( R_1 \)]. \( R_1 \) must be in turn a second-order or a higher-order relationship. In Fig. 1, Dirlist is the only higher-order relationship.

The arrows leaving a relationship diamond can be labeled 1 and 2 to indicate the order of the entities in a 2-tuple. Labels "M" and "MO" indicate many and many-ordered entities, respectively.

We notice that the is-a relationship exists as in the E-R data model. The is-a relationship (triangle) between two entities indicates one form of inheritance. If \( A \) is-a \( B \), then \( A \) inherits the properties of \( B \) i.e. all the attributes and relationships defined in \( B \) will also exist in \( A \) in addition to \( A \)'s own properties. The is-a relationship is particularly useful with predefined entity types. For example, User is a Unique-entity and therefore inherits a unique id attribute. The interactive part of our system will examine each attribute present in each entity type of the model and will request the user to specify a data type for the attribute value. For example, a data type will be requested for the six attributes of the entity Mailitem. A valid data type can be any of the user-defined types, predefined entity types or built-in types (Natural, Integer, Text, Boolean). Our system will then ask the user to provide the system global entities. These will be instances of the MER entity types. For our example, the following attribute value types must be provided: Refno: Ref, Body: Text, Whensent: Date, Reply-req: Boolean, Subject: Text, From: Name, Name: Text. Furthermore, we need to define the following global entities: direct: Directory and deskr: Deskring.
2.3 Keyword Based Formatted Description

Having provided the entities, the next step is to specify the tasks to be performed by the intended software system. The method we employ is based on a set of keywords. The fixed format of an operation is as follows:

Operation name: OPNAME
Operands: type1, type2, ..., typen [ → retunrtype ]
Syntax: par1, ..., parn, OPNAME, parj, ..., parn
Constraints: A list of constraints
Semantics: A list of actions
Description: Natural language description

Fig. 2 includes the KFD specifications of the required operations for the mailing system. In the Operands section, we provide a list of entity types that match respectively with each of the parameters in the Syntax section which in turn are instances of those entity types. For example, in the POST operation of Fig. 2, the parameters USERX and MAILX are instances of entity types User and Mailitem respectively. The parameters are the only known entities in the scope of the operation apart from global entities. The Syntax part specifies how the operation must be invoked. Then follows a list of constraints connected by logical operators (and/or). Each constraint ensures that the properties of certain entities have the required characteristics. Using a very small set of keywords, we can check on attribute values or on the content of relationship sets. We refer to an attribute or a relationship set of an entity with the keyword of. Hence, To of MAILX is not empty ensures that relationship To of entity MAILX is not empty. The keyword is-in tests for membership of an entity in a relationship set. The user is not required to bother about the type compatibility between the entity and those in the relationship set. Hence, some degree of incompleteness and ambiguity is accepted in the user specification. This incompleteness will be resolved through the use of a knowledge-base as discussed later or, in the worst case, through a clarification dialogue with the user.

The Semantics part of the KFD describes the meaning or the goal of the operation as a list of actions to be performed on the entities. A set of action keywords exists to manipulate entity values. 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 to one or more entities as the goal of the operation requires. This is possible through the use of the control construct for-all. Additionally, conditional control keywords exist such as if-then-else and such-that. These constructs help the user express their requirements closer to their natural level of communication. Action keywords have different meanings depending on the type of entities they are applied to. This permits us to have a very small set of keywords to facilitate the user’s task.

3 Transformation into Formal Specifications

An automated generation of VDM can be attempted due to the structure of both the MER and the KFD. Our transformation system is primarily rule-based in nature and we attempt to minimize the user interaction in the transformation process. We distinguish three sources of knowledge in our system:

1. Knowledge about the MER structure (properties of entity types, inheritance, sets, ordered sets) and semantics of keywords.
2. Common and domain knowledge through predefined entity types.
3. Knowledge about the syntax and semantics of VDM.

Domain knowledge is represented as conceptual structures [14] where concept types and relationships form a semantic network. This makes domain knowledge to be independent of the transformation rules. The other types of knowledge are rules embedded in the transformation process. The transformation process is divided in two parts: generating the VDM state from the MER diagram and creating VDM operations using KFDs. All three types of knowledge are used in both stages. However, the information captured in the MER diagram is also required in transforming the KFDs into VDM operations.

3.1 The VDM State

The user-defined entity types as well as the predefined entity types of the current domain are the data types of the VDM state. Each entity type is transformed into a VDM record type where the attribute names and the relationship names become fields of the VDM record. Primitive relationships are VDM sets or lists depending on whether the relationship is labeled "M" or "MO" respectively. Second-order relationships
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 deskx
Semantics: add mailx to Out of TrayList(userx) of deskx
Description: ...

Operation name: READ
Operands: User, Ref — Mailitem
Syntax: userx read mailid
Constraints: mailid is in Pend of TrayList(userx) of deskx
Semantics: return mailx such that mailx is in Pend of TrayList(userx) of deskx
and Refno of mailx is mailid
Description: ...

Operation name: COLLECT
Operands: User, Ref — Mailitem
Syntax: userx collect mailid
Constraints: mailid is in In of TrayList(userx) of deskx
Semantics: return mailx such that mailx is in In of TrayList(userx) of deskx
Description: ...

Operation name: ADDALIAS
Operands: User, Name, User
Syntax: userx addalias namey, usery
Constraints: namey is not in DirList(userx) of deskx and
usery is in DirList(userx)
Semantics: add (namey, usery) to DirList(userx) of deskx
Description: ...

Operation name: DELALIAS
Operands: User, Name, User
Syntax: userx delalias namey, usery
Constraints: namey is in DirList(userx) of deskx
Semantics: remove namey from DirList(userx) of deskx
Description: ...

Operation name: CLEAR
Operands: User
clear userx
Constraints: Out of TrayList(userx) of deskx is not empty
Semantics: for all mailx in Out of TrayList(userx) of deskx do
create newmail with To of mailx, Cc of mailx, From of mailx,
Subject of maily, Reply, req of mailx, Refs of mailx,
now(), rewref(), Body of maily
for all userx in TrayList of deskx do
if userx is in DirList(userx) of deskx then
add newmail in In of TrayList(userx) of deskx
if Reply, req of mailx and userx is in DirList(userx) of deskx
then add newmail in Pend of TrayList(userx) of deskx
end-for
end-for
for all m in Pend of TrayList(userx) of deskx do
if Refno of m is in Refs of outs
then create newmail with To of mailx,
Subject of mailx, Reply, req of mailx, Refs of mailx,
now(), rewref(), Body of mailx
add newmail in In of TrayList(userx) of deskx
remove m from Pend of TrayList(userx) of deskx
end-for
set Out of TrayList(userx) of deskx to empty
Description: ...

Figure 2: KFDs for the mailing system operations
associate an entity type to another entity type or to a set/ordered set of entities of another type. They are translated to VDM as mapping types where the first entity type in the 2-tuple is mapped into the second entity type or into a set/list of entities of that type. Higher-order relationships are also translated into mapping types. A higher-order relationship associating entity of type \( E \) to a relationship \( R \) is defined as a mapping type from \( E \) to \( R \) where \( R \) is another mapping type in VDM.

For every entity type \( A \) inheriting an entity type \( B \), the VDM record of \( A \) will include the fields of the VDM record of \( B \) if \( B \) is not a predefined entity type. If \( B \) is a predefined entity type then the knowledge encapsulated in \( B \) must be used in the transformation process. In other words, \( B \) might have its own transformation rules since \( B \) models a piece of common or domain knowledge known to the transformation system. The generated VDM state for the mailing system is shown in Fig. 3. This state is obtained through the transformation rules and refined through a process which makes use of the knowledge about VDM syntax.

Any entity type \( Ent \) inheriting from the \textit{Unique-entity} type will possess an attribute \textit{Ent-id} of type \textit{Ent-idT} in the VDM record, where \textit{Ent-idT} is a simple VDM type defined as a set of unique \textit{Ent} identification numbers. This amount of information is deduced from the knowledge associated with the \textit{Unique-entity} type. The VDM global variables are created from the user specified global entities.

### 3.2 The VDM Operations

There exist a one-to-one mapping between the KFDs and the VDM operations. The instance variables in the \textit{Syntax} part of the KFD are the parameters of the VDM operation. Their corresponding types are respectively listed in the \textit{Operands} section of the KFD. Hence, the header of a KFD as shown in section 2.3 will generate the VDM component:

\[
\text{OPNAME} (\text{PAR}_1 : \text{type}_1, \text{PAR}_2 : \text{type}_2, \ldots, \text{PAR}_n : \text{type}_n) [R : \text{returntype}].
\]

If the operation returns a value, a \textit{returntype} is specified which is the entity type of the value returned. In VDM, a variable \( R \) of \textit{returntype} will be returned where \( R \) will be determined in the postcondition of the VDM operation.

The VDM operation includes a clause which indicates which global state variables the operation needs to access. This is specified by the keyword \textit{ext}. For every global entity \( \text{ENT} \) of type \textit{EntT} used in a KFD where \( \text{ENT} \) is affected by the keywords \textit{add, remove} or \textit{set} stated in the \textit{Semantics} part of the KFD, we will generate:

\[
\text{ext.} \ \text{ENT} : \text{wr} \ \text{EntT}.
\]

If \( \text{ENT} \) is only referred to but never affected by those action keywords, then we will generate:

\[
\text{ext.} \ \text{ENT} : \text{rd} \ \text{EntT}.
\]

The constraints are conditions that must hold in order to be able to execute the operation. A constraint in the KFD will create one or more conditions in the precondition clause of a VDM operation. The KFDs are not complete descriptions, hence additional information might be necessary to eliminate inconsistencies or incompleteness. The list of constraints connected by the keywords \textit{and/or} creates a sequence of VDM preconditions connected by the operators \textit{A} or \textit{V}. During the initial stages of constrained specification, a user may not know the type of objects and consequently the KFD keywords will have to be interpreted in a strict fashion as more information becomes available during the transformation stages. For example, testing for membership in VDM is done differently depending on the type of the group set while only one keyword is used for that purpose within KFDs. As another example, the action \textit{add} might be the addition of an entity to a set, or the appending of an entity to an array. The VDM global variables are created from the user specified global entities.

\[
\text{State:}
\]

\begin{tabular}{|c|c|}
\hline
\textbf{State} & \textbf{Definition} \\
\hline
DESKR & Deskring \\
DIRECT & Directory \\
Deskring & User \rightarrow Trays \\
Directory & User \rightarrow Alias \\
Alias & Name \rightarrow User \\
Trays & IN \rightarrow Material-set \\
PEND & Material-set \\
Mailitem & TO \rightarrow Name-list \\
CC & Name-list \\
FROM & Name \\
SUBJECT & Text \\
REPLREQ & Bool \\
REPS & Ref-set \\
WHENSENT & Date \\
REFNO & Ref \\
BODY & Text \\
\hline
\textbf{Ref} & Nat */ a set of distinct reference numbers */ \\
\textbf{Date} & */ A system generated date */ \\
\textbf{User} & Nat */ a set of distinct user id numbers */ \\
\textbf{Name} & Text \\
\hline
\end{tabular}

Figure 3: Generated VDM state for the mailing system
entity DESKR. However, we are only given the identification number of the mail item, i.e. MAILID. Thus, the constraint MAILID is-in PEND of TRAYLIST(USERX) of DESKR is given an entity of type Ref and a set of entities of type Mailitem. Using the knowledge about the structure of the mail item and the fact that each mail item is uniquely identified by the attribute REFNO of type Ref which in turn inherits from Unique-entity, the transformation system deduces that the previous constraint really means that there must exist a mail item in PEND of TRAYLIST(USERX) of DESKR having REFNO equal to MAILID. Furthermore, this mail should be unique in the relationship set PEND. TRAYLIST(USERX) refers to the entity Trays associated to USERX in the relationship TRAYLIST. Using the knowledge about VDM mapping types, the transformation system must additionally generate a VDM precondition verifying that USERX is part of the domain of TRAYLIST first. This level of details is not required from the user and is taken care by the system.

The rules to generate VDM specifications for the is-in keyword are presented below:

For each constraint of the form obj is-in relset do
If relset is a primitive relationship of "M" entities ent then
If obj is an entity of type ent then
   obj is-in relset \implies obj \in relset
If obj's type is equal to the type of a unique id attribute attrid of entity ent then
   obj is-in relset \implies (\exists e \in \text{dom relset}) (\text{ATTRID}(e) = obj)
If obj's type does not correspond to any attribute type of ent then
   Request clarification from the user.
If relset is a primitive relationship of "MO" entities ent then
If obj is an entity of type ent then
   obj is-in relset \implies obj \in \text{elems relset}
If obj's type is equal to the type of a unique id attribute attrid of entity ent then
   obj is-in relset \implies (\exists e \in \text{elems relset}) (\text{ATTRID}(e) = obj)
If obj's type does not correspond to any attribute type of ent then
   Request clarification from the user.
If relset is a second order or high order relationship from ent to some other entity or some relationship then
If obj is an entity of type ent then
   obj is-in relset \implies obj \in \text{dom relset}

4 Related Work

In [6] 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 necessitating only 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
Figure 4: Generated VDM operations for the mailing system
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 being carried out with software practitioners.

In [11] 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 [11] 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 [5], the authors describe a methodology similar to ours in certain aspects such as data modelling 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 OSD approach [5]. Our KFDs are more natural to a user’s conceptual thinking than the forced upon state-based model as proposed by them.

In [16] 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 [10], 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. [13] 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 [13] and also in [8]. 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.

5 Conclusion

Our main interest in this paper is to introduce a simple informal description medium namely the MER diagrams and KFDs and explore a knowledge based approach for formalizing it. We have also presented a subset of the transformation rules which transform the descriptions into VDM specifications. Due to space constraints, we did not provide the complete set of transformation rules. An interactive software system for aiding the transformation process reported in this paper is being developed. Details about dialog generation, triggers for initiating a dialog with the user, and more details about the transformation process can be found in [4].

References


