Formal Transformations: Using SA and VDM as Different Views in Software Development

Nico Plat
Peter Gorm Larsen*
Hans Toetenel

Delft University of Technology
Faculty of Technical Mathematics and Informatics
P.O.Box 356, NL-2600 AJ Delft, The Netherlands

November 8, 1993

Abstract

Structured methods and formal methods for software development can supplement each other by eliminating each other's weak points. It is even better if an efficiently high level of integration can be achieved such that the products of each method can automatically be transformed into a product of the other method. In this paper a small case study is presented in which this principle is illustrated. It is shown how data flow diagrams (the main product of Structured Analysis) can automatically be translated into VDM specifications, in this way creating two different views for the analyst/designer on the system being developed: a graphical view (the DFD) and a textual view (the VDM specification).

Index-terms: Formal methods, VDM, Structured Analysis, systems design, technology transfer

1 Introduction

The proper use of *formal methods* in software development can provide better complexity control and may therefore make the production of software cheaper while resulting in higher quality products. Nevertheless, software development in professional communities is often completely ad hoc, or at best supported by structured methods such as OOD, SA/SD and JSD. Therefore, a challenging task for research institutions is to develop paradigms for software construction with increased formality.

We think that such a paradigm can best be based upon a *combination* of an (existing) structured method and an (existing) formal method. Such a combination has the following advantages:

*Peter Gorm Larsen is with The Institute of Applied Computer Science (IFAD), Munkebjergvænget 17, DK-5230 Odense M, Denmark. This work was funded with the financial support of the Commission of the European Communities under the COMETT programme (90/5199-Bc), IFAD and the Danish COWI-foundation.

- the level of formality in the design process is increased. Therefore, the amount of ambiguities in the design products can be reduced.
- instead of having to write a formal specification from scratch, this formal specification can now be *automatically* constructed, starting from a framework the analyst/designer is familiar with: the notations and guidelines provided by the structured method;
- the analyst/designer is offered a multiple 'view' on the system: a *graphical* view (usually focusing on the structure of the system) by the structured method and a *textual* view (focusing on the functionality of the system) by the formal method.

Our work in this area so far has concentrated on trying to combine *Structured Analysis (SA)* [DeMarco78, Gane&77, Yourdon&75] with *The Vienna Development Method (VDM)* [Bjørner&82, Jones90, BSIVDM91]. We think that a well-integrated combination of notations can be achieved by using *Data Flow Diagrams (DFDs)* as a graphical view of the system and VDM as a textual view. These different views emphasise different aspects of the specified system. In general, it can be said that the graphical view focuses on an overview of the *structure* of the system, whereas the textual view focuses on the detailed *functionality* of the system. Thus, we would like the analyst/designer to be able to view his specification in different ways, depending upon e.g. to whom he wants to show the specification, and which specific aspects of the system he wants to emphasise.

The methodological aspects of our approach have been explained in [Larsen&91a]. At the base of a combined structured/formal method lies a formal connection between the structured method (which therefore has to be 'formalized') and the formal method. An analysis of the possibilities for mapping DFD constructs to corresponding VDM constructs is made in [Toetenel&90] and [Plat&91]. A formal semantics in terms of a transformation from an abstract syntax for DFDs to an abstract syntax for VDM-SL is presented in [Larsen&91b]. This paper is intended to illustrate the latter formal transformation by means of a small case study. Instead of the abstract syntaxes used in the formal transformation we will use 'normal' representations for DFDs (graphical notation) and VDM-SL (textual notation).

In the next section a short overview of SA, VDM and our combined method - called *Structured VDM (SVDM)* - is given. The main purpose of this overview is to introduce some terminology and to provide some major references to relevant material, but we will assume that the reader is able to understand the notations and conventions used in this paper. In section 3 the case study - the design of a simple spelling checker - is illustrated: the DFDs are presented (including some decomposition steps), the formal transformation to VDM is explained and the resulting VDM specification is given. Section 4 contains an analysis of the results and gives an overview of the status of our work.

Relation to other work

When DFDs were introduced originally, they were presented as a graphical notation. The intended semantics of this notation was defined verbally. Since then, some work has been done on formalizing DFDs, with the intention either of disambiguating their meaning, or of using the formal semantics as a base for an integrated structured/formal method.

In [Randell91] a translation back and forth between DFDs and Z-specifications is mentioned. However, this translation simply corresponds to generating an abstract syntax representation of the DFDs, and it does not provide any semantics to composite DFDs. [Alabiso88] contains an explanation of how one can manually transform DFDs into an object-oriented design. The paper touches upon some of the problematic issues arising in a transformation from DFDs, in particular the notion of 'I/O uncohesiveness' which we also encountered in our formalization of DFDs. In [Semmens&91] a small example of how a DFD can be transformed in Z is presented. No formal semantics of the DFDs is presented and it is not clear to what extent the transformation can be automated. In [Bruza&89] some guidelines for how semantics can be attached to DFDs are given. It is sketched how DFDs can be transformed to a Petri Net variant combined with path expressions. However, only an initial description of a transformation is presented. In [Adler88] a semantic base for guiding the decomposition process in the construction of a hierarchy of DFDs is presented. This work is based on graph theory in an algebraic setting. [Eisenback&89] presents a very loose sketch of how DFDs can be transformed into programs in a functional programming language. However, even the composite data transformers must be supplied directly, so nothing is generated, and only very simple DFDs (even without data stores) are considered. In [Fraser&91] a rule-based approach for transforming SA products into VDM specifications is presented. Their VDM specifications are, however, very explicit and hard to read, mainly because of the way decision tables have been taken into account.

Our work differs from the work mentioned above in that our approach provides a complete formal semantics of DFDs (assuming a sequential implementation is being pursued). The looseness property of VDM-SL is used extensively in order to correctly formalize the connections between the data transformers in the DFD. This provides the user with a semantics for DFDs that can be extended when more information about the system is available.

2 SVDM: A combined SA/VDM method

Structured VDM (SVDM), illustrated in this paper, is a prototype method for the development of sequential systems based on a combination of Structured Analysis (SA) and the Vienna Development Method (VDM).

2.1 Structured Analysis

Structured Analysis/Structured Design (SA/SD) [DeMarco78, Gane&77, Yourdon&75] is one of the most widely used structured software development methods. It is based on a function-oriented approach, using data flow abstraction to describe the flow of data through a network of transforming processes (called data transformers) having access to data stores. The first step in the method is the construction of a *context diagram* which contains the external processes for the system and one data transformer representing the system as a whole. The context diagram is then decomposed into a hierarchy of data flow diagrams (DFDs). Each of the primitive data transformers, i.e. each data transformer which the analyst/designer considers to be sufficiently low-level to implement right away instead of further decomposing it, contains a (textual) *mini-specification (MS)*, describing the relation between the inputs and the outputs of the data transformer. The type of the data inside data stores and the type of the data which flows between

the data transformers is specified in a *data dictionary (DD)* and/or in an *entity-relationship diagram (ERD)*. The main design paradigm within the method consists of the transformation of the DFDs into *structure charts (SCs)*, which represent the static structure of the final system.

2.2 The Vienna Development Method

The *Vienna Development Method* (VDM) [Bjørner&82, Jones90] is a model-oriented formal software development method for the description and development of software systems. The method uses a specification language called VDM-SL[1]. A system design is produced through a series of specifications where each specification is more concrete and closer to the implementation than the previous one. Each of these development steps introduces a number of proof obligations, which, if discharged, ensure the correctness of the implemented system. This process is called refinement.

In VDM-SL one can define types, values, functions, operations, and a state. The types are defined by means of domain expressions which provide an abstract description of the data used in the specification. The functions and the operations can be defined either implicitly (describing *what* they do by means of a post-condition) or explicitly (describing *how* they do it in an procedural way). The operations have access to global objects which are defined in a state definition, whereas the functions are purely applicative.

2.3 An overview of SVDM

The paradigm used for SVDM is the *decomposition* of DFDs followed by a *composition* of data transformers. The composition of data transformers is used to create higher-level data transformers until only one data transformer remains; this data transformer corresponds to the context diagram and describes the functionality of the system as a whole. SVDM is described in more detail in [Larsen&91a]. The relevant steps of the method for our case study are:

1. Analyse the problem and develop a high level DFD (context diagram).
2. Decompose the context diagram into a new DFD where the high-level data transformer has been split into several data transformers. Each of these data transformers is subsequently decomposed until an acceptable low-level hierarchy of DFDs has been reached.
3. Provide type information for all data stores and data flows. This type information must be supplied *textually*, i.e. by means of VDM domain definitions.

   It is now possible to (automatically) generate a first VDM specification from the DFD. This transformation has been formally defined in [Larsen&91b]
4. Complete the analysis phase by specifying all primitive data transformers. These specifications are called mini-specifications and they must be described as operation definitions in VDM-SL.

   It is now possible to automatically generate a VDM specification of the entire system where the mini-specifications are taken into account.

[1] In this paper we will use the term VDM-SL for the standard currently being defined under the auspices of BSI and ISO [BSIVDM91].

These steps constitute the first 4 relevant steps of SVDM. Steps 1 and 2 are the same as their counterparts in SA. Step 3 is the first step towards formalization, but it is carried out using SA-terminology. In principle, a VDM specification can be (automatically) generated from the information supplied so far. Step 4 finishes the analysis part by specifying the meaning of the primitive data transformers. In SA this is not done in any formal manner. However, it is normally recommended to use structured English, decision tables, decision trees or similar notations. In our approach the mini-specifications must be expressed in VDM-SL. When the mini-specifications have been supplied, a more detailed VDM specification can be automatically generated.

3 A case study: the spelling checker

The problem of our case study, the design of a simple spelling checker, is fairly well-known. The problem is used in [Sommerville82] to explain the notions of data flow diagrams and structure charts. The spelling checker accepts a file name from a user, checks whether a file with that name is present (if the file is not present the user is notified), reads words from the file and looks up each word in a dictionary. If the word is present in the dictionary, it is considered to be correctly spelled, otherwise it is displayed on the user's terminal. The user may then decide whether the word is misspelled or correctly spelled. If misspelled, the word is held in a file of misspelled words. If correctly spelled, it is to be added to the dictionary. We assume that the file system can be modeled as a mapping from file names to files, and that a file is just a sequence of characters. These requirements may be ambiguous and incomplete, but in order to focus attention to relevant matters only we will silently introduce suitable interpretations and solutions when necessary.

3.1 A DFD for the spelling checker

Our DFD for the spelling checker is a simplified version of the one given in [Sommerville82], although it certainly is not the only valid DFD for the description of a software system with the required functionality. The context diagram for the spelling checker is shown in figure 1.

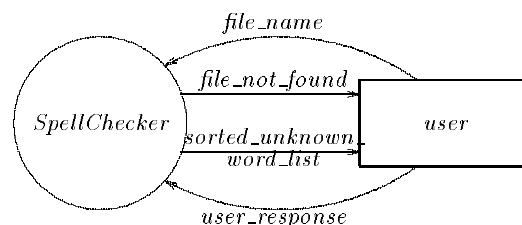

Figure 1: Context diagram for the spelling checker

3.2 The first decomposition step

In the first decomposition step the context diagram can be decomposed into 3 separate data transformers: *FindFile*, *SortWords* and *LookUpAndAdjust* (figure 2), each of which models part

of the functionality of the spelling checker. A number of data stores are introduced in the DFD as well, modeling a file system with files (*docs*), a dictionary (*dic*) and a data base of misspelled words (*misws*), respectively.

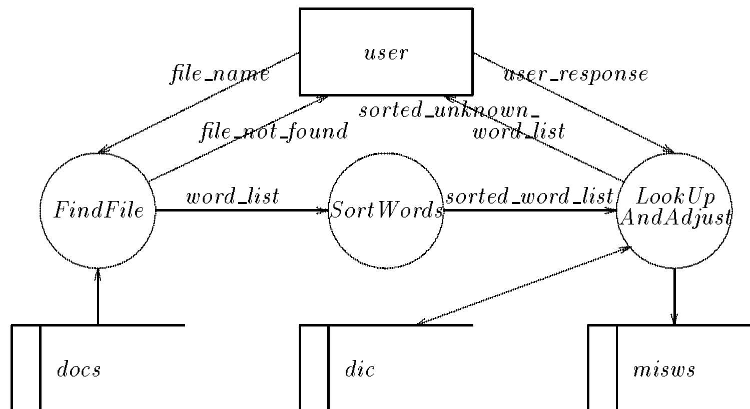

Figure 2: The highest level DFD

One of the first steps in transforming a DFD into a VDM specification is to identify the *possible execution orders* in the DFD. By a possible execution order we mean an order in which all data transformers can be executed such that those which are producing output to be used by other data transformers are executed first. We call a data flow between two data transformers an *internal* data flow, and a data flow between a data transformer and an external process an *external* data flow. In this situation we only have one possible execution order: First *FindFile*, then *SortWords* and finally *LookUpAndAdjust*. The possible execution orders play an important role in the formal transformation of a DFD into a formal specification.

Modeling the DFD

The entire DFD is described by a *VDM module*, in which the connection between the data transformers is described by a *VDM operation*. In the example, the combined effect of *FindFile*, *SortWords* and *LookUpAndAdjust* is described by the operation *SpellChecker*. The former three data transformers will be decomposed themselves (they form a new DFD at a lower level), so their definition is given in another module. To be able to use them in the definition of the operation *SpellChecker*, they must be imported from the module in which they are defined. In this way an entire hierarchy of DFDs can be mapped upon a module structure (figure 3).

The signature of the operation *SpellChecker* is determined by examining all the input and output data flows of the DFD (disregarding those to or from external processes). In general, input data flows are modeled as input parameters to the operation and output data flows are modeled as return values. In figure 2 we can see that the DFD does not have any input or output data flows, and therefore the signature of *SpellChecker* is empty. Data flows from and to external processes are modeled as *state components*. This means that these data flows will appear in an external-clause of the VDM operation[2].

[2]Inside an operation the use of the different state components is specified after the `ext` keyword.

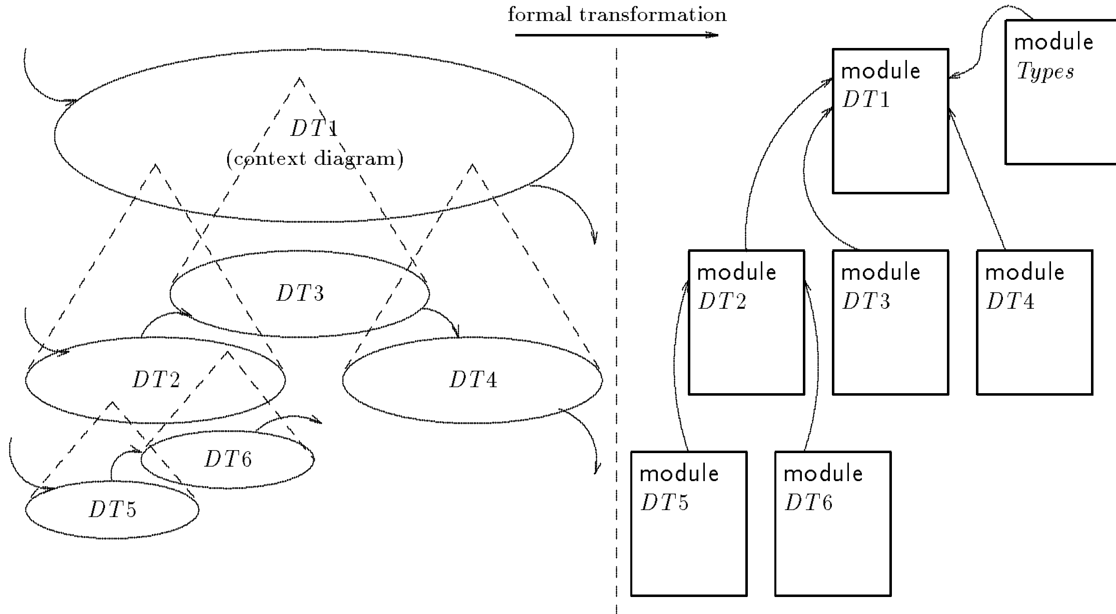

Figure 3: Transformation of a hierarchy of DFDs into a VDM module structure

Combining data transformers

Describing the effect of a combination of data transformers in general can be quite difficult. It depends very much on what the DFD is used for and which specific model for the DFD the user has in mind. This problem has been the subject of our earlier work and is described in [Plat&91]. In SVDM we have made a few assumptions that make the interpretation easier:

- The DFD must model a *sequential system*. This means that we can assume that a data transformer, using the output of another data transformer as its input, can be invoked as soon as that output is available.
- Each of the data transformers in the DFD must produce exactly one output value on each of the outputs of that data transformer for each input value. As shown in [Plat&91], this is not a serious limitation because the formal specification language can be used to model situations in which this is not acceptable, e.g. by introducing union types for the output values in which one of the member types is a quote literal denoting the absence of an output value.
- The data transformers in each possible execution order in the DFD must form an *acyclic* graph. This allows us to find a (unique) dependency order in which the individual data transformers should be executed.

Constructing the pre-condition

In order to construct the pre-condition for the operation *SpellChecker* we can assume that, since the execution of *SpellChecker* starts with the execution of *FindFile*, the pre-condition of *FindFile* must be satisfied as well. When *FindFile* has been executed, *SortWords* can only be executed

if its pre-condition has been fulfilled. For the pre-condition of *SpellChecker* this means that it must be *possible* to fulfill the pre-condition of *SortWords* using a suitable value for *word_list* (the output from *FindFile*). In VDM-SL this can be expressed by using existential quantification: the predicate-part of that existential quantification can be used to construct 'suitable' input values for the pre-condition of *SortWords* by quoting the post-condition of *FindFile*, in this way ensuring that the input for *SortWords* is indeed the output of *FindFile*. The connection between *SortWords* and *LookUpAndAdjust* can be taken into account in the same way.

The pre-condition of *SpellChecker* can now be constructed by taking the conjunction of the above described predicates. In general, the pre-condition of such an operation is constructed by repeating the process described above for each possible execution order in the DFD, and then taking the disjunction of each of them.

Constructing the post-condition

The construction of the post-condition is done quite similarly: the post-condition of *SpellChecker* can be satisfied if values for the data flows between the lower-level data transformers can be constructed which satisfy the post-conditions of those data transformers.

For *SpellChecker* this means that there must exist a *word_list* which satisfies the post-condition of *FindFile* and, when used as input value for *SortWords*, satisfies the post-condition of *SortWords*. The post-condition of the latter should be satisfied using a value for *sorted_word_list* which will also satisfy the post-condition of *LookUpAndAdjust*. The resulting predicate describes the conditions that an implementer must fulfill in order to correctly implement *SpellChecker*.

In general, the post-condition of an operation defining the connection between the data transformers in a DFD is constructed by repeating the process described above for each independent partition in the DFD, and then taking the conjunction of all of them.

The resulting VDM specification is:

module *SpellCheckerModule*

imports

from *TypeModule*

types *Documents*, *Dictionary*, *MisspelledWords*, *FileName*, *FileNotFound*, *WordList*, *SortedWordList*, *UserResponse*,

from *FindFileModule*

operations $FindFile : () \xrightarrow{o} WordList$ using *docs*, *fn*, *fnf*,

from *SortWordsModule*

operations $SortWords : WordList \xrightarrow{o} WordList$,

from *LookUpAndAdjustModule*

operations $LookUpAndAdjust : SortedWordList \xrightarrow{o} ()$ using *dic*, *misws*, *ur*, *suwl*

exports

operations $SpellChecker : () \xrightarrow{o} ()$ using *docs*, *dic*, *misws*, *fn*, *ur*, *suwl*, *fnf*

definitions

state $SpellCheckerState$ of
$docs$: $Documents$
dic : $Dictionary$
$misws$: $MisspelledWords$
fn : $FileName$
ur : $UserResponse$
$suwl$: $SortedWordList$
fnf : $FileNotFound$
end

operations

$SpellChecker\,()$
ext rd $docs$: $Documents$
wr dic : $Dictionary$
wr $misws$: $MisspelledWords$
rd fn : $FileName$
rd ur : $UserResponse$
wr $suwl$: $SortedWordList$
wr fnf : $FileNotFound$
pre $pre\text{-}FindFile(docs, fn, fnf) \wedge$
$\exists\, wl : WordList, fnf' : FileNotFound \cdot$
$post\text{-}FindFile(docs, fn, fnf, wl, fnf') \wedge$
$pre\text{-}SortWords(wl) \wedge$
$\exists\, swl : SortedWordList \cdot$
$post\text{-}SortWords(wl, swl) \wedge$
$pre\text{-}LookUpAndAdjust(swl, dic, misws, ur, suwl)$
post $\exists\, wl : Wordlist \cdot$
$post\text{-}FindFile(docs, fn, \overleftarrow{fnf}, wl, fnf) \wedge$
$\exists\, swl : SortedWordList \cdot$
$post\text{-}SortWords(wl, swl) \wedge$
$post\text{-}LookUpAndAdjust\,(swl, \overleftarrow{dic}, \overleftarrow{misws}, ur, \overleftarrow{suwl},$
$dic, misws, suwl)$

end $SpellCheckerModule$

The decomposition of *FindFile* and *LookUpAndAdjust* is further explained in subsections 3.3 and 3.4. The decomposition of *SortWords* has been omitted for reasons of space.

Data dictionary

For the formal transformation it has been assumed that the user has provided a *data dictionary* in the form of VDM domain equations, collected in a module with the name *TypeModule*. When constructing a module for a DFD, the relevant types are imported from *TypeModule* (see e.g.

the import clause of *SpellCheckerModule*). The data dictionary for our case study is specified as:

module *TypeModule*

exports

types *Documents*, *Dictionary*, *MisspelledWords*, *FileName*, *File*, *FileNotFound*, *WordList*, *SortedWordList*, *UserResponse*

definitions

types

$Documents = FileName \xrightarrow{m} File;$

$FileName = [\mathsf{token}];$

$File = \mathsf{char}^*;$

$FileNotFound = \mathbb{B};$

$Dictionary = Word\text{-}\mathsf{set};$

$Word = \mathsf{char}^+;$

$MisspelledWords = Word\text{-}\mathsf{set};$

$WordList = Word^*;$

$SortedWordList = WordList$

$\mathsf{inv}\ swl \triangleq \forall\, i, j \in \mathsf{inds}\ swl \cdot i < j \;\Rightarrow\; LexicallySmaller(swl(i), swl(j))\ ;$

$UserResponse = (\text{YES} \mid \text{NO})^*$

functions

$LexicallySmaller : \mathsf{char} \times \mathsf{char} \to \mathbb{B}$

$LexicallySmaller\,(c_1, c_2) \triangleq$

is not yet defined

end *TypeModule*

At this point we are not really interested in the exact specification of the auxiliary function *LexicallySmaller*, but we are happy with the intended meaning indicated by its name. We can specify in VDM-SL that a function will be specified at a later time by the clause is not yet defined.

3.3 The decomposition of FindFile

The data transformer *FindFile* can be decomposed into two other data transformers: *GetFile* (reading a file from file system *docs* and returning a *file*) and *Split* (decomposing a file into a list of individual words: *word_list*). The DFD for *FindFile* is shown in figure 4.

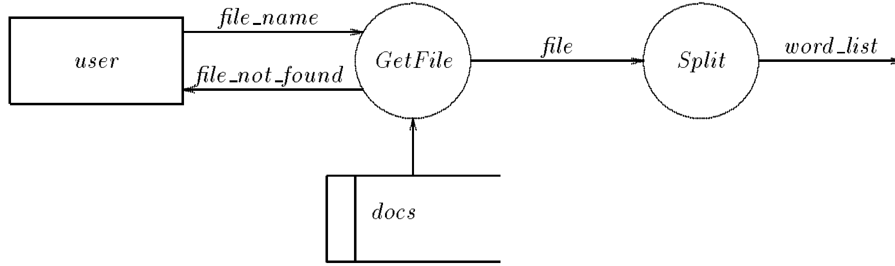

Figure 4: DFD for the decomposition of *FindFile*

In principle, the module for *FindFile* is constructed in much the same way as the one for *SpellChecker*. The difference is that (although we can see this only in the abstract syntax for DFDs, *not* in the graphical notation as used in figure 4) in this case the data transformers *GetFile* and *Split* are primitive, i.e. they are not decomposed into lower-level DFDs. Since at this point the user has not (yet) provided any mini-specifications, our knowledge of these data transformers is restricted to their types. The post-conditions of these operations are very simple: a predicate with the value 'true', i.e. any implementation which satisfies the type restrictions also satisfies the entire specification for that operation.

The resulting VDM specification is:

module $FindFileModule$

imports

from $TypeModule$

types $Documents, FileName, FileNotFound, File, WordList$

exports

operations $FindFile : () \xrightarrow{o} WordList$ using $docs, fn, fnf$

definitions

state $FindFileState$ of
$docs : Documents$
$fn : FileName$
$fnf : FileNotFound$
end

operations

$FindFile\,()\; wl : WordList$
ext rd $docs : Documents$
rd $fn : FileName$
wr $fnf : FileNotFound$
pre $pre\text{-}GetFile(docs, fn, fnf) \wedge$
$\exists f : File, fnf' : FileNotFound \cdot$
$post\text{-}GetFile(docs, fn, fnf, f, fnf') \wedge pre\text{-}Split(f)$
post $\exists f : File \cdot$
$post\text{-}GetFile(docs, fn, \overleftarrow{fnf}, f, fnf) \wedge post\text{-}Split(f, wl);$

$GetFile\,()\ f : File$
ext rd $docs : Documents$
rd $fn \quad : FileName$
wr $fnf \quad : FileNotFound$
post true;

$Split\,(f : File)\ wl : WordList$
post true

end $FindFileModule$

3.4 The decomposition of LookUpAndAdjust

Data transformer *LookUpAndAdjust* can be decomposed into two other data transformers: *LookUp* (accepting a list of words *sorted_word_list* and comparing each of these words with a collection of known words in a dictionary *dic*) and *Adjust* (adding words to either the dictionary *dic* or a collection of misspelled words *misws*, depending upon a *user_response* from the external process *user*). The DFD for *LookUpAndAdjust* is shown in figure 5.

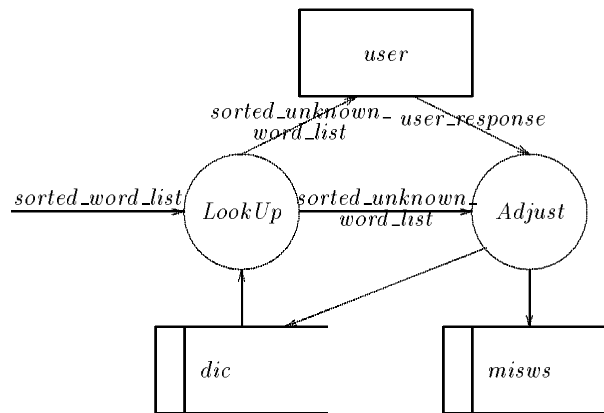

Figure 5: DFD for the decomposition of *LookUpAndAdjust*

Up until now, we have specified the operation that defines the connection between the data transformers in the DFD *implicitly* by means of post-conditions. However, in many cases an *explicit* specification can be more natural, especially as we limit ourselves to sequential systems. The formal transformation is therefore parameterized with a specification *style*: either implicit or explicit specifications can be generated. For the specification of *LookUpAndAdjust* we have given the explicit specification.

Constructing an explicit operation body first follows the same route as the construction of an implicit operation. First, the possible execution orders of the DFD are determined. For each possible execution order a sequence of call statements (calling the operations for the data transformers) is generated in the that order. Names for the data flows between the data transformers are generated by using define statements. Afterwards, the independent partitions are combined in a non-deterministic statement, thus ensuring that all operations for the data transformers are indeed executed for one of the possible execution orders.

The resulting VDM specification is:

module *LookUpAndAdjustModule*

imports

from *TypeModule*

types *Dictionary*, *MisspelledWords*, *SortedWordList*, *UserResponse*

exports

operations $LookUpAndAdjust : SortedWordList \xrightarrow{o} ()$ using *dic*, *misws*, *ur*, *suwl*

definitions

state *LookUpAndAdjustState* of
$dic : Dictionary$
$misws : MisspelledWords$
$ur : UserResponse$
$suwl : SortedWordList$
end

operations

$LookUpAndAdjust : SortedWordList \xrightarrow{o} ()$

$LookUpAndAdjust\,(swl) \triangleq$
def $suwl = LookUp(swl)$;
$Adjust(suwl)$
pre $pre\text{-}LookUp(swl, dic, suwl) \wedge$
$\exists\, suwl_1 : SortedWordList, suwl' : SortedWordList \cdot$
$post\text{-}LookUp(swl, dic, suwl, suwl_1, suwl') \wedge$
$pre\text{-}Adjust(suwl_1, dic, misws, ur)$;

$LookUp\,(swl : SortedWordList)\ suwl_1 : SortedWordList$
ext rd $dic : Dictionary$
wr $suwl : SortedWordList$
post true;

$Adjust\,(suwl_1 : SortedWordList)$
ext wr $dic : Dictionary$
wr $misws : MisspelledWords$
rd $ur : UserResponse$
post true

end *LookUpAndAdjustModule*

Please note that even though that the body of *LookUpAndAdjust* is specified explicitly it is possible to attach a pre-condition to the operation. The data transformers *LookUp* and *Adjust* are primitive and at this point the user has not (yet) provided any mini-specifications. Therefore these data transformers are simply restricted to their types as in section 3.3.

3.5 The specification of primitive data transformers

The last step in SVDM as we have sketched it in this paper is the incorporation of mini-specifications for all the primitive data transformers, supplied by the user. These mini-specifications are passed as a parameter to the formal transformation: if they are not provided, post-conditions with the value true are generated as sketched in subsections 3.3 and 3.4.

Implicit Mini-specifications

Mini-specifications must be provided by the user in the form of VDM operations. They can be provided either in an implicit style or in an explicit style. The implicit mini-specifications for our case study are given below.

Operation *GetFile* expects a file name *fn*, modelled as a state component, from the external process *user*. *GetFile* will return the *File* identified by *fn* (if it exists in the collection of files) *doc* and signal the external process *user* whether or not such a file exists by changing the state component *fnf* accordingly.

$$
\begin{array}{l}
\mathit{GetFile}\,()\ f : \mathit{File} \\
\mathsf{ext\ rd}\ \ \mathit{docs} : \mathit{Documents} \\
\quad\ \ \mathsf{rd}\ \ \mathit{fn} : \mathit{FileName} \\
\quad\ \ \mathsf{wr}\ \mathit{fnf} : \mathit{FileNotFound} \\
\mathsf{post\ if}\ \mathit{fn} \in \mathsf{dom}\ \mathit{docs} \\
\qquad \mathsf{then}\ f = \mathit{docs}(\mathit{fn}) \wedge \mathit{fnf} = \mathsf{false} \\
\qquad \mathsf{else}\ f = \mathsf{nil}\ \wedge \mathit{fnf} = \mathsf{true}
\end{array}
$$

Operation *Split* transforms a file (modelled as a sequence of characters) into a *WordList*.

$$
\begin{array}{l}
\mathit{Split}\,(f : \mathit{File})\ \mathit{wl} : \mathit{WordList} \\
\mathsf{post\ if}\ f = \mathsf{nil} \\
\qquad \mathsf{then}\ \mathit{wl} = [\,] \\
\qquad \mathsf{else\ let}\ w_m = \{ i \mapsto f\,(i, \ldots, j) \mid i, j \in \mathsf{inds}\ f \cdot \\
\qquad\qquad\qquad\qquad\qquad (i < j) \wedge \\
\qquad\qquad\qquad\qquad\qquad (i = 1 \vee l(i) = \text{' '}) \wedge \\
\qquad\qquad\qquad\qquad\qquad (j = \mathsf{len}\ f \vee l(j) = \text{' '}) \wedge \\
\qquad\qquad\qquad\qquad\qquad \forall\, k \in \{i+1, \ldots, j-1\} \cdot l(k) \neq \text{' '}\}\ \mathsf{in} \\
\qquad\qquad \mathit{wl} = [w_m(i) \mid i \in \mathsf{dom}\ w_m]
\end{array}
$$

Operation *LookUp* takes a *WordList* as its argument and looks up each word in that word list in *dic*. If the word does not appear in *dic* it is concatenated to the output word list and to state component *suwl*.

$$
\begin{array}{l}
\mathit{LookUp}\,(\mathit{swl} : \mathit{SortedWordList})\ \mathit{suwl}' : \mathit{SortedWordList} \\
\mathsf{ext\ rd}\ \ \mathit{dic} : \mathit{Dictionary} \\
\quad\ \ \mathsf{wr}\ \mathit{suwl} : \mathit{SortedWordList} \\
\mathsf{post}\ \mathit{suwl} = [\mathit{swl}(i) \mid i \in \mathsf{inds}\ \mathit{swl} \cdot \mathit{swl}(i) \notin \mathit{dic}] \wedge \mathit{suwl}' = \mathit{suwl}
\end{array}
$$

Operation *Adjust* takes a list of (unknown) words as its argument and expects a user response *ur* for each word from external process *user*. If the user response is YES then the word is added to *dic*, otherwise it is saved in a collection of misspelled words *misws*.

$$
\begin{array}{l}
Adjust\,(suwl : SortedWordList) \\
\mathsf{ext\ wr}\ dic : Dictionary \\
\quad \mathsf{wr}\ misws : MisspelledWords \\
\quad \mathsf{rd}\ \ ur : UserResponse \\
\mathsf{pre\ len}\ ur \geq \mathsf{len}\ suwl \\
\mathsf{post}\ dic = \overleftarrow{dic} \cup \{suwl(i) \mid i \in \{1, \ldots, \mathsf{len}\ suwl\} \cdot ur(i) = \textsc{yes}\} \wedge \\
\qquad misws = \overleftarrow{misws} \cup \{suwl(i) \mid i \in \{1, \ldots, \mathsf{len}\ suwl\} \cdot ur(i) = \textsc{no}\}
\end{array}
$$

Explicit Mini-specifications

The explicit style of specifying operations in VDM provides the user with a wide variety of statements, as also found in many modern imperative programming languages. Consequently, such specifications have a strong operational flavor.

```
GetFile : () -o-> File
GetFile (fn) △
  if fn ∈ dom docs
  then (fnf := false;
        return docs(fn) )
  else (fnf := true;
        return nil  )


Split : File -o-> WordList
Split (f) △
  if f = nil
  then return [ ]
  else (dcl w : Word := [ ],
            wl : WordList := [ ];
        for e in f
        do if e = ' '
           then if w ≠ [ ]
                then (wl := wl ⁀ [w];
                      w := [ ])
           else w := w ⁀ [f(i)];
        if w ≠ [ ]
        then (wl := wl ⁀ [w];
              w := [ ]);
        return wl )
```

$LookUp : SortedWordList \xrightarrow{o} SortedWordList$

$LookUp\,(swl) \triangleq$
  $(suwl := [\,];$
  for all $i \in$ inds swl
  do if $swl(i) \notin dic$
    then $suwl := suwl \frown [swl(i)];$
  return $suwl$)

$Adjust : SortedWordList \xrightarrow{o} ()$

$Adjust\,(swl) \triangleq$
  for all $i \in$ inds swl
  do if $ur(i) = \text{YES}$
    then $dic := dic \cup \{swl(i)\}$
    else $misws := misws \cup \{swl(i)\}$

Explicit specifications are in general regarded as 'less abstract' than implicit specifications. In the standard VDM development paradigm [Jones90] the explicit mini-specifications are considered to be implementations of their implicit counterparts.

4 Conclusions

In this paper we have illustrated the automatic generation of formal specifications in VDM-SL from data flow diagrams, by means of a small case study.

One of the main advantages of the approach is that the analyst/designer is given a large amount of freedom: in the initial phases of analysis/design he can concentrate on the structure of the system by using a graphical notation while at the same time he can change to a textual view. The latter view allows him to focus on the functions the system should perform (with the additional advantage that he can now see exactly what his DFD means). Within the textual view he can, also depending upon the stage of development, choose different levels of abstraction: VDM-SL allows for both implicit and explicit definitions, so mini-specifications can be provided both in an implicit and explicit style, and the same holds for the higher-level data transformers.

The specifications produced in this way are rather large, compared to the size of the problem. A large overhead is created by the structuring mechanism of VDM-SL, and we think that this overhead will decrease when the problems grow in size: DFDs for larger problems have a larger number of data transformers but the number of levels in the hierarchy of DFDs usually does not increase with the same rate as the number of data transformers does. Therefore, a relatively larger amount of these data transformers will be primitive (for which no separate modules are created).

The reason why we end up with specifications that look the way they do is that we imposed a number of restrictions on these DFDs. However, we believe that if these restrictions are not acceptable to a user, then it is still possible to generate VDM specifications (albeit not entirely automatically), possible extended with concurrency, that satisfactorily describe the intended behavior of the system.

In order to realistically illustrate the validity of our approach, it should be tried on a large, real-world example. To do this, tool support must be available which can automatically transform DFDs into VDM. The implementation of such tool support will be one of our next steps in this project.

Acknowledgments

We would like to thank John Dawes for his suggestions to improve our use of English in this paper.

5 References

[Adler88] Mike Adler. An Algebra for Data Flow Diagram Process Decomposition. *IEEE Transactions on Software Engeneering*, 14(2):169–183, February 1988.

[Alabiso88] Bruno Alabiso. Transformation of Data Flow Analysis Models to Object Oriented Design. In *OOPSLA'88 Proceedings*, pages 335–353, ACM, November 1988.

[Bjørner&82] Dines Bjørner and Cliff B. Jones. *Formal Specification & Software Development. Series in Computer Science*, Prentice-Hall International, 1982.

[Bruza&89] P.D. Bruza and Th.P. van der Weide. *The Semantics of Data Flow Diagrams.* Technical Report 89-16, University of Nijmegen, Department of Informatics, October 1989.

[BSIVDM91] *VDM Specification Language – Proto-Standard.* Technical Report, British Standards Institution, March 1991. BSI IST/5/50.

[DeMarco78] Tom DeMarco. *Structured Analysis and System Specification.* Yourdon Press, 1978.

[Eisenback&89] Susan Eisenback, Lee McLoughlin, and Chris Sadler. Data-Flow Design as a Visual Programming Language. In *Fifth International Workshop on Software Specification and Design*, pages 281–283, IEEE Computer Society, IEEE, May 1989.

[Fraser&91] Martin D. Fraser, Kuldeep, and Vijay K. Vaishnavi. Informal and Formal Requirements Specification Languages: Bridging the Gap. *IEEE Transactions on Software Engineering*, 17(5):454–466, May 1991.

[Gane&77] Chris Gane and Trish Sarson. *Structured Systems Analysis: Tools & Techniques.* Prentice-Hall, Englewood Cliff, New Jersey 07632, 1977.

[Jones90] Cliff B. Jones. *Systematic Software Development Using VDM (second edition).* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Larsen&91a] Peter Gorm Larsen, Jan van Katwijk, Nico Plat, Kees Pronk, and Hans Toetenel. SVDM: An Integrated Combination of SA and VDM. In *Methods Integration Conference*, September 1991.

[Larsen&91b] Peter Gorm Larsen, Nico Plat, and Hans Toetenel. A Formal Semantics of Data Flow Diagrams. *Submitted to Formal Aspects of Computing*, July 1991. .

[Plat&91] Nico Plat, Jan van Katwijk, and Kees Pronk. A Case for Structured Analysis/Formal Design. In *VDM '91 – Formal Software Development Methods*, VDM Europe, Springer-Verlag, October 1991.

[Randell91] Gill Randell. The Integration of Structured and Formal Methods. In *Workshop Structured Analysis and Formal Methods*, pages 35–36, York, June 1991.

[Semmens&91] Lesley Semmens and Pat Allen. *Using Yourdon and Z: an Approach to Formal Specification.* Technical Report, Faculty of Information and Engineering Systems, Leeds Polytechnic, January 1991. Handed out at a workshop called "Structured Analysis and Formal Methods" June 1991 in York.

[Sommerville82] I. Sommerville. *Software Engineering.* Addison-Wesley, London, 1982.

[Toetenel&90] Hans Toetenel, Jan van Katwijk, Nico Plat. Structured Analysis – Formal Design, using Stream & Object oriented Formal Specification. In *Proc. of the ACM SIGSOFT International Workshop on Formal Methods in Software Development. Software Engineering Notes* 15(4): 118-127, ACM Press, Napa, California, USA, 9-11 May 1990.

[Yourdon&75] E. Yourdon and L.L. Constantine. *Structured Design.* Yourdon Press, 1975.