# SVDM: An Integrated Combination of SA and VDM

Peter Gorm Larsen *
Jan van Katwijk
Nico Plat
Kees Pronk
Hans Toetenel

Delft University of Technology
Faculty of Technical Mathematics and Informatics
P.O. Box 356, NL-2600 AJ Delft, The Netherlands
E-mail: peter@ifad.dk, {jan, nico, kees, toet}@dutiaa.tudelft.nl

**abstract**

In this paper we present *Structured VDM (SVDM)*, a combination of Structured Analysis (SA) and the Vienna Development Method (VDM). Using this method, the designer is able to move back and forth between an SA graphical specification and a VDM textual specification. In this way the most appropriate notation for explaining the specification to people with a different background can be selected. An overview of the methodological aspects of the approach is given.

## 1 Introduction

The use of formal methods in software development can provide better complexity control, and may therefore reduce the cost of software while resulting in higher quality products. Nevertheless, software development in professional communities is currently at best supported by informal, so-called structured methods such as OOD, SA/SD and JSD. A major issue for educational and research institutions, therefore, is to develop paradigms for software development with increased formality, and to teach the professional community their proper use. One of the problems with such paradigms seems to be the lack of a clear view on the transition from structured to formal methods and back. Although we agree with Hall who states that one does not have to be a mathematician in order to be able to use formal

*Peter Gorm Larsen is with The Institute of Applied Computer Science (IFAD), Forskerparken 10, DK-5230 Odense M, Denmark. This work was funded with the financial support of the Commission of the European Communities under the COMETT program (90/5199-Bc), IFAD and the Danish COWI foundation.

methods [Hall90], most professional software developers are scared by the apparent amount of mathematics needed to read or write a formal specification. It is very hard to convince those professionals that using formal methods will lead to better specifications and programs, and that investments in these methods are indeed cost-effective.

Based on class room experiences, and experiences with software engineers in the field, we are convinced that a simple transition from the use of structured methods to the use of formal methods is too much to ask. A method in which aspects of formal methods and structured methods are somehow combined seems more appropriate.

*Structured Analysis/Structured Design (SA/SD)* [Constantine&79] is one of the most widely used software development methods. When trying to achieve an increase in the use of formal methods, it seems worthwhile to investigate possibilities to either annotate the analysis and design results (the most important ones are *data flow diagrams (DFDs)*) or to exchange either SA or SD with a formal method, in our case the *Vienna Development Method (VDM)* [Bjørner&82, Jones90].

We envisage a combined SA/VDM method as a software development method in which two *views* are maintained: the traditional SA approach (construction of a context diagram followed by refinement in terms of DFDs until an acceptable level of detail has been reached) is followed by a process of transforming DFD constructs into VDM constructs, in this way generating a formal specification of the software system under construction. This process gives the designer two views of the system: a *graphical view* provided by the SA/SD products and a *textual view* provided by the formal specification. The advantages are:

- the designer is given structural guidance for the construction of a formal specification of the system. The formal specification should then be used as a starting point for the further development of the system;
- the formal specification can be regarded as a formal semantics of the DFD, so:
  - the meaning of the DFD (which only used to have an intuitive meaning) has now become precise and unambiguous;
  - it possible to check the DFD for any syntactic and semantic inconsistencies.

In this paper we describe a method in which aspects of SA and VDM are combined. The method is called *Structured VDM (SVDM)*. In Section 2 we briefly discuss the notions of 'structured' and 'formal' methods, and we give a short introduction to both SA and VDM, in order to fix the terminology used in this paper. In Section 3 the rationale for SVDM is discussed. Section 4 contains a detailed description of SVDM. The section starts with a short overview of the method, followed by a more detailed discussion of the various steps, where a small example is used to illustrate each step. In Section 5 we give an overview of the design products of the method. Finally, in Section 6, some remarks are made on the current status of our work and on our future plans.

# 2 Structured and formal methods

The terms 'structured' and 'formal' methods are both ill-defined. The term 'structured method' is used to indicate members of a category of classical software development methods, essentially developed in the mid seventies, of which SA/SD can be regarded as an important representative. These methods are 'structured' in the sense that they provide the designer with a design philosophy and with a set of detailed rules which the designer should follow step by step, at the end of which the complete system has been built. In addition to this, notations are provided in which the designer can express his design products. Usually these notations are graphical, and they lack a formal meaning. Therefore, the design products are potentially ambiguous.

In [Wing90] formal methods are described as "mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop and verify systems in a systematic, rather than ad hoc manner." The word 'formal' in formal method is thus used in the sense that such a method has a (sound) mathematical basis. This basis provides the means to precisely define concepts like consistency, correctness, specification and implementation. This basis also gives the means to inspect the realizability of specifications, to prove the correctness of a system design, and to prove system properties without the need to have an executable representation of the system.

Central to any formal method is a formal specification language, with a formal semantics allowing an unambiguous denotation of the meaning of specifications expressed in that language. Furthermore, these languages contain constructions for expressing abstraction. Abstraction makes designing complex systems easier, by characterizing the essential properties of the problem and emphasizing *what* is required rather than *how* it is achieved. The interested reader is referred to e.g. [Plat&91a] for a more in-depth discussion on formal methods.

## 2.1 Structured Analysis/Structured Design

*Structured Analysis/Structured Design (SA/SD)* is one of the most widely used 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, together with access to data stores. Such a network, which is the most important product of SA, is called a *data flow diagram (DFD)*. A DFD is a directed graph consisting of elementary building blocks. Each building block can be represented graphically (Figure 1). Through the years

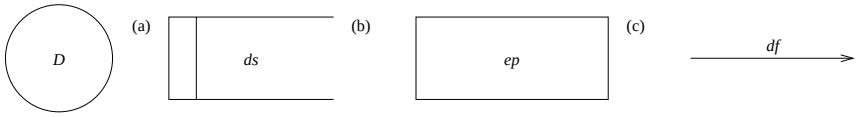


Figure 1: (a) Data transformer; (b) Data store; (c) External process; (d) Data flow

a number of different dialects have evolved, and extensions have been defined (e.g. SA/RT [Ward&85] and SSADM [Longworth&86]), but we will limit ourselves to a very elementary form of DFDs with a small number of different building blocks:

(a) *Data transformers.* Data transformers are usually denoted as bubbles. They can have an arbitrary number of inputs, and an arbitrary number of outputs.

(b) *Data stores.* Data stores provide for (temporary) storage of data.

(c) *External processes.* External process are processes that are not part of the system but belong to the outside world. They are used to show where the input to the system is coming from and where the output of the system is going to.

(d) *Data flows.* Data flows are represented as arrows, connecting data transformers to each other or connecting data transformers to external processes. They represent a flow of data between the constructs they connect. The flow of data is unidirectional in the direction of the arrow.

DFDs are used to model the information flow through a system. As such they provide a limited view of the system: in their most rudimentary form they neither show the control flow of the system nor any timing aspects.

The first step in the SA/SD method is the construction of a context diagram which contains the external processes for the system and one data transformer representing the system. The context diagram is then decomposed into a hierarchy of DFDs. Each of the primitive data transformers contains a (textual) mini-specification, which describes 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 and/or in an entity-relationship diagram. The main design paradigm within the method guides the transformation of the DFDs into structure charts, which represent the structure of the final system.

## 2.2 The Vienna Development Method

VDM is a model-oriented formal method based on a denotational semantic setting, intended to support stepwise refinement of abstract models into concrete implementations. The method includes a formal specification language, VDM-SL, which supports various forms of abstraction.

*Representational abstraction* is supported through data modeling facilities. These facilities are based on six mathematical data structuring mechanisms: sets, sequences, maps, composite objects, cartesian products and unions. At a lower level the language provides various numeric types, booleans, tokens and enumeration types. By using the data structuring mechanism and the basic data types, compound data types can be formed, in VDM denoted by the term *domains*. Domains form in general infinite classes of objects. These classes have a specific mathematical structure. Subtyping is supported by attaching *domain invariants* to domain definitions.

*Operational abstraction* is supported by both functional abstraction and relational abstraction, the first by means of (fully referentially transparent) function specification, the second by operation specification. Both functions and operations may be specified *implicitly* using *pre and post conditions*, or *explicitly* using *applicative* constructions to specify functions, and *imperative* constructions to specify operations.

Operations have direct access to a collection of global objects: the *state* of the specification. The state is constructed as a composite object, built from labelled components.

A VDM specification thus typically consists of a state description (possibly augmented with invariant and initialization predicates), a collection of domain definitions (possibly augmented with invariants), a collection of constant definitions, a collection of operations, and a collection of functions.

# 3 The rationale behind the combination

SVDM should be a method with a firm theoretical basis, which can be understood and used by practitioners from industry. SVDM provides the graphical notations from SA/SD, the methodological guidelines from the SA phase of SA/SD, and the formal aspects of VDM, in this way emphasizing the strong points of SA/SD and VDM. At various stages of development, specifications of the system can be viewed in different ways.

In addition to these starting points, SVDM is based on the following principles:

- During different phases of development, different aspects of the problem domain should be dealt with. Such 'separation of concerns' is a powerful technique when trying to manage the complexity and size of the problem. In SVDM a distinction is made on a high level of abstraction: during the analysis phase, emphasis is put on the *functional* requirements of the system, whereas during the design phase, emphasis is put on the *non-functional* requirements of the system.

- Software development is not strictly a top-down (or bottom-up) activity. Working both top-down and bottom-up is supported in SVDM by the modular structure of the design products, and by the possibility for (partially) performing automatic transformations. Such an automatic transformation is not formally defined in this paper, however.

In this combination of a structured and a formal method we have chosen to use SA and VDM because both are well-known methods which have been developed in the early seventies. It is therefore safe to assume that both methods are mature by now. However, there is nothing that prevents us from expanding such a combined method by enabling additional views from other methods. If it turns out that this combination of SA and VDM is valuable in real software projects, we foresee that other notations and methods may be incorporated as well.

# 4 An overview of SVDM

The paradigm used by our method is the *decomposition* of DFDs in the *analysis phase* followed by an annotated *composition* of DFDs in the *design phase*. 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. The decomposition is done using SA techniques, while the composition of DFDs is described in VDM-SL. The combined method consists of the following steps:

1. Analyze the problem and develop a context diagram, representing the system boundaries.

2. Decompose the context diagram by splitting the high-level data transformer into several lower level 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 may be supplied either *textually*, by means of VDM domain definitions, or, if they are more complex, *graphically*, by means of entity-relationship diagrams.

   It is now possible to (automatically) derive a first VDM specification. This specification is called the *level 0 VDM specification*[1]. Since we consider this document of little practical value we will refer to it as a *secondary document*.

4. Complete the analysis phase by specifying all primitive data transformers. These specifications are called mini-specifications and they must be described either as function or operation definitions in VDM-SL.

   It is now possible to (automatically) generate a VDM specification where the mini-specifications are taken into account. This specification is called the *level 1 VDM specification*.

5. For each DFD containing two or more data transformers, control information must be provided defining the order in which the data transformers should be combined.

   Again it is possible to (automatically) generate a VDM specification using the control information provided in this step. This specification is called the *level 2 VDM specification*.

   It is also now possible to (automatically) generate *level 1 structure charts* for the designed system.

6. Refine mini-specifications into explicit function and operation definitions.

   It is now possible to (automatically) generate a textual representation of the design in the form of a VDM specification. This specification is called the *level 3 VDM specification*.

   At the same time it is now possible to (automatically) generate a *level 2 structure chart* description.

7. Optimize the formal specification. In this (optional) step the level 3 VDM specification is changed such that it better reflects non-functional requirements of the problem to be solved. Structure charts which also reflect these changes can be constructed as well. The VDM specification that is constructed in this step is called the *level 4 VDM specification*, the structure charts are referred to as *level 3 structure charts*.

A graphical overview of the method is given in Figure 2. Steps 1 and 2 are the

[1]The various 'levels' we distinguish refer to the different stages of development of the design products.

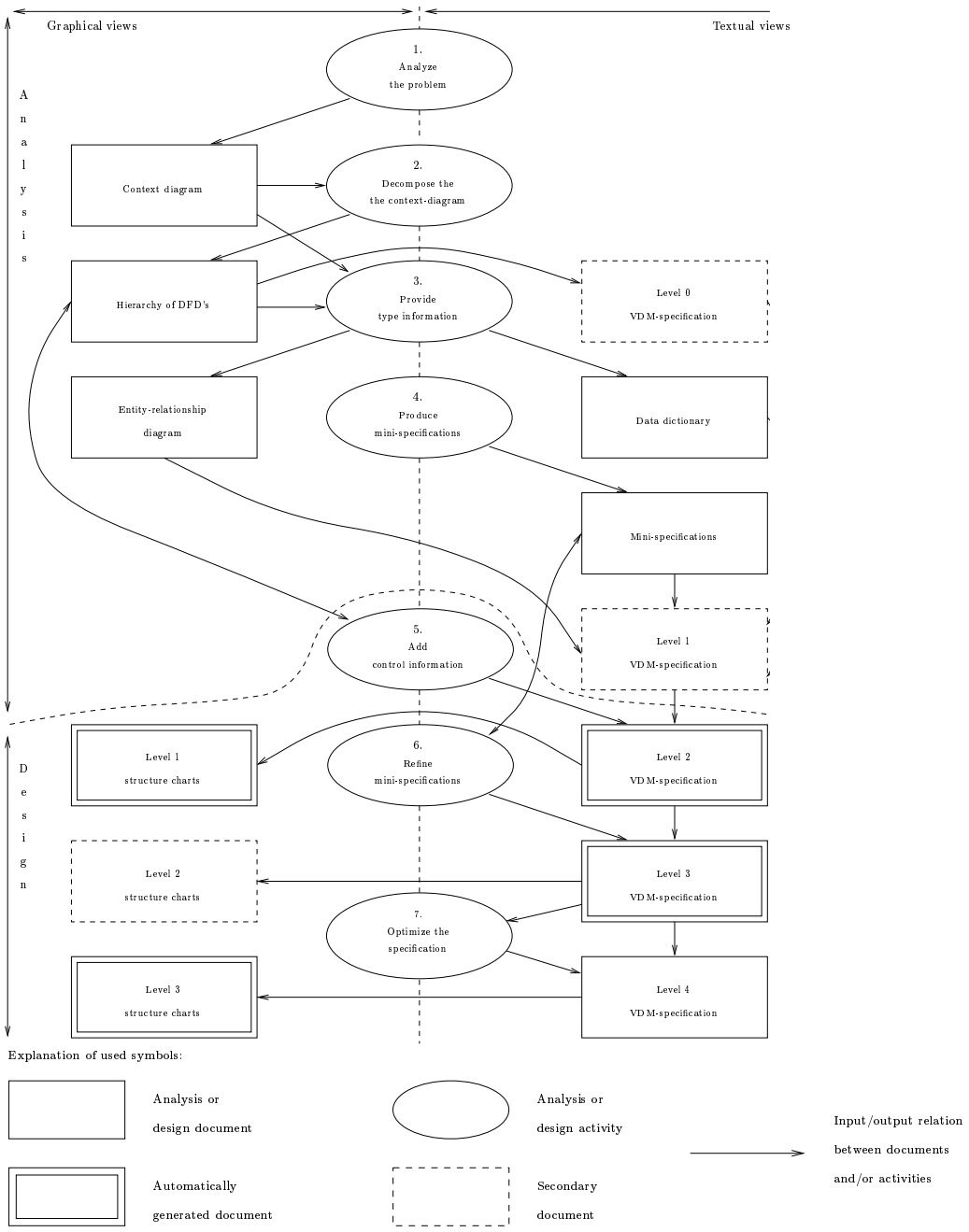


Figure 2: An overview of SVDM

same as their counterparts in SA. Step 3 is the first step towards formalization, but it is carried out using SA terminology. In step 4 the analysis part is finished by specifying the meaning of the primitive data transformers. The mini-specifications must be expressed in VDM-SL (preferably as implicit definitions). When the mini-specifications have been supplied, a more detailed VDM specification can be (automatically) generated. In step 5 the first design decisions on how the different data transformers must be controlled are taken. Using these design decisions a readable VDM specification of the system can be automatically generated. By means of the control information it is also possible to (automatically) generate structure charts. The mini-specifications are then refined (step 6) from implicit to explicit

specifications following the normal VDM refinement rules. It is then possible to automatically generate structure charts and a VDM specification. These documents are meant to serve as a basis for the implementation, and they represent textual and graphical views of the design.

In the following subsections we will discuss each of the steps in more detail. We will illustrate some of the steps by means of a small example: a component of a simple spelling checker.

## 4.1 Analyze the problem

As prescribed by SA, the first step is to analyze the given problem and to develop a high level DFD (also called a context diagram). This context diagram will only contain one data transformer, a number of external processes, and information about the data which flows between the data transformer and its external processes.

## 4.2 Decompose the context diagram

The context diagram is then decomposed into a new DFD where the data transformer has been split into several data transformers and data stores. For each of the data transformers the decomposition process is repeated until an acceptable level of detail has been reached, i.e. each data transformer performs a transformation which does not need further decomposition. Thus, the result of this refinement activity is a decomposition hierarchy of DFDs.

### Example: the spelling checker component

In our example we will show how a small component of a spelling checker can be developed using SVDM. The spelling checker example is used in [Sommerville82] to explain the notions of data flow diagrams and structure charts. The component which we will use to illustrate SVDM, will expect a *file_name*, and checks whether a file having this name is present in a file system *docs*. If the file is present, it is read and its contents is split into a sequence of words *word_list*. Duplicates are removed from *word_list* and the resulting list is sorted. Each word in *word_list* is then checked against a dictionary by another component (not incorporated in our example).

The hierarchy of DFDs for this component after analyzing the problem, constructing and decomposing the context diagram is shown in Figure 3.

## 4.3 Provide type information

All data stores and data flows must be described by type information. In SA type information is usually provided using some textual notation, or by means of entity-relationship diagrams. In our combined method we require formal type definitions, either textually, by means of VDM domain definitions, or, if they are more complex, graphically, by means of entity-relationship diagrams.

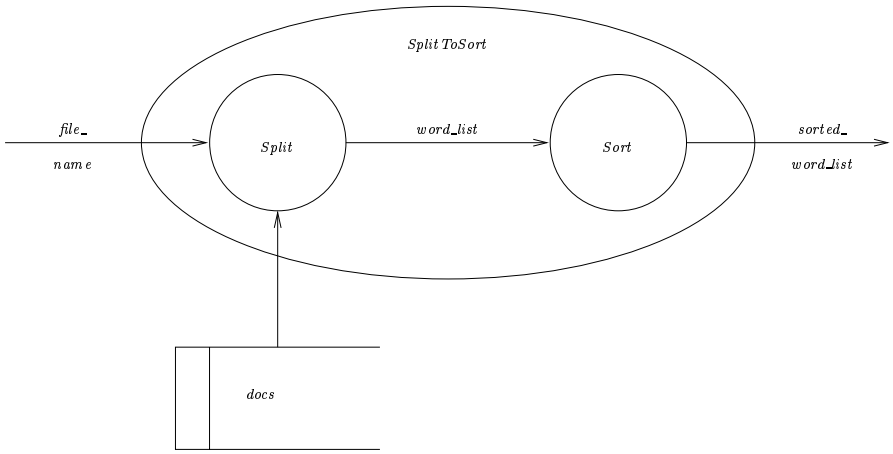


Figure 3: Two-level hierarchy DFD for *SplitToSort*

## Example: the spelling checker component (continued)

Because the spelling checker component is a simple problem, ordinary VDM domain definitions suffice for the required type information:

**types**

$$
\begin{array}{ll}
\mathit{Docs} & = \mathit{FileName} \xrightarrow{m} \mathit{File}; \\
\mathit{FileName} & = \textbf{token}; \\
\mathit{File} & = \textbf{char}^*; \\
\mathit{Word} & = \textbf{char}^+; \\
\mathit{WordList} & = \mathit{Word}^*; \\
\mathit{SortedWordList} & = \mathit{Word}^*
\end{array}
$$

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

**values**

$\mathit{WordSeparator}$ : **char is not yet defined**

**functions**

$$\mathit{LexicallySmaller} : \mathit{Word} \times \mathit{Word} \to \textbf{B}$$

$\mathit{LexicallySmaller}$ (**-**, **-**) $\triangleq$
**is not yet defined**

The invariant for the domain *SortedWordList* ensures that the words in a sequence of words are lexically sorted. Since, at this point, the exact definition of *LexicallySmaller* is not yet relevant, we say that the function is ' **is not yet defined**'. For the same reason we have chosen not to define the constant character value *WordSeparator* yet.

## Generating the level 0 VDM specification

When the type information has been provided for all data flows it is possible to generate a (level 0) VDM specification which captures the information supplied so

far, i.e. the VDM specification ensures that the primitive data transformers transform data of the right type. Essentially, this VDM specification gives a semantics to the hierarchy of DFDs. A possible transformation from DFDs to VDM is described in [Plat&91b][2]. Although it is possible to generate a VDM specification at this stage, the specification will not be very helpful to the designer in itself. Therefore, we consider this generated document as a secondary document. The level 0 VDM specification can be used to perform a consistency check of the system specified so far.

The type information provided as entity-relationship diagrams is transformed into VDM domain definitions similar to the way done in [Dick&91]. The type information for the data stores is modeled as a part of the state for that DFD, similar to the way done in [Plat&91b].

## Example: the spelling checker component (continued)

The level 0 VDM specification for the spelling checker component is given below. We only show the operation definitions for the data transformers.

**state** *SpellingChecker* **of**
  $docs : Docs$
**end**

**operations**

$Split\ (file\_name : FileName)\ word\_list : WordList$
**ext rd** $docs : Docs$
**post true**;

$Sort\ (word\_list : WordList)\ sorted\_word\_list : SortedWordList$
**post true**;

$SplitToSort\ (fn : FileName)\ sorted\_word\_list : SortedWordList$
**ext rd** $docs : Docs$
**pre** $pre\text{-}Split(file\_name, docs) \wedge$
  $\exists file\_name' : FileName, docs' : Docs, word\_list : WordList \cdot$
    $pre\text{-}Split(file\_name', docs') \wedge$
    $post\text{-}Split(file\_name', word\_list, docs') \wedge$
    $pre\text{-}Sort(word\_list)$
**post** $\exists word\_list : WordList \cdot$
    $pre\text{-}Split(file\_name, docs) \wedge post\text{-}Split(file\_name, word\_list, docs) \wedge$
  $\exists file\_name' : FileName, docs' : Docs, word\_list : WordList \cdot$
    $pre\text{-}Split(file\_name', docs') \wedge$
    $post\text{-}Split(file\_name', word\_list, docs') \wedge$
    $pre\text{-}Sort(word\_list) \wedge$
    $post\text{-}Sort(word\_list, sorted\_word\_list)$

[2]The principle for the composition of data transformers is similar to what is called the Parallel ACcess (PAC) approach in [Plat&91b].

At this stage, the post conditions of *Split* and *Sort* always yield the value '**true**'. Because no mini-specifications have been provided yet, from a semantical point of view *any* implementation with the right signature will satisfy *Split* and *Sort*.

The higher-level data transformer in Figure 3 is in the VDM specification represented by the operation *SplitToSort*. The VDM operation *SplitToSort*, which defines the relation between *Split* and *Sort*, must express the looseness present in the DFD notation. This is achieved by using quantified expressions 'introducing' values which will satisfy the pre conditions and post conditions of *Split* and *Sort*. For a more detailed discussion on this subject, the reader is referred to [Plat&91b].

## 4.4 Producing mini-specifications

When type information has been provided, the designer develops mini-specifications (directly expressed in VDM-SL) for each of the primitive data transformers. For each primitive data transformer with more than one input data flow or more than one output data flow, it is necessary to relate these data flows to the parameters for the function/operation definition that is used as the mini-specification for the data transformer.

In order to avoid overspecification we recommend to use implicit definitions; in this way it can be expressed *what* should be done, not *how* it should be done. Of course, there may be cases where it is more 'natural' to express the transformation of data explicitly, and therefore we also allow the explicit style. Operations are used if the data transformer is connected to at least one data store, and functions are used when no data stores are involved.

## Generating the level 1 VDM specification

The level 1 VDM specification, which can be generated at this point, will be more detailed than the level 0 VDM specification, but it will still be of little practical value and it is therefore also considered to be a secondary document. The level 1 VDM specification will be a refinement of the level 0 VDM specification: while the primitive data transformers at level 0 simply restrict the functions/operations to be of the right type, the mini-specifications restrict the input/output relations.

## Example: the spelling checker component (continued)

The designer provides mini-specifications for the operations *Split* and *Sort*.

**operations**

*Split* (*file_name* : *FileName*) *word_list* : *WordList*
**ext rd** *docs* : *Docs*
**post if** *file_name* $\notin$ **dom** *docs*
  **then** *word_list* = [ ]
  **else let** *file* = *docs*(*file_name*) **in**
    *word_list* = *ChangeIntoWords*(*file*);

*Sort* (*word_list* : *WordList*) *sorted_word_list* : *SortedWordList*
**post elems** *word_list* = **elems** *sorted_word_list*;

$SplitToSort\ (fn : FileName)\ sorted\_word\_list : SortedWordList$
**ext rd** $docs : Docs$
**post** $\exists\, word\_list : WordList \cdot$
$post\text{-}Split(file\_name, word\_list, docs) \wedge$
$\exists\, file\_name' : FileName, docs' : Docs, word\_list : WordList \cdot$
$post\text{-}Split(file\_name', word\_list, docs') \wedge$
$post\text{-}Sort(word\_list, sorted\_word\_list)$

**functions**

$ChangeIntoWords : File \to WordList$

$ChangeIntoWords\ (file) \triangleq$
$[[file(j) \mid j \in \{i, \ldots, \textbf{len}\ file\} \cdot \neg \exists\, k \in \{i, \ldots, j\} \cdot$
$file(k) = WordSeparator]$
$\mid i \in \textbf{inds}\ file \cdot file(i) \neq WordSeparator \wedge$
$(i = 1 \vee file(i-1) = WordSeparator)]$

These mini-specifications for *Split* and *Sort* replace the original post conditions. Since neither *Split* nor *Sort* has a pre condition, the calls to *pre-Split* and *pre-Sort* can be removed from *SplitToSort*.

## 4.5 Adding control information to the DFDs

For each DFD containing more than two data transformers, information must be provided on the order in which the data transformers should be combined. This control information resembles the strategy used in *Structured Design (SD)*. In SD the approach is to find the input part, the central processing part and the output part. In our approach it is appropriate to find the same parts and then first combine the input part with the central processing part, and then combine the result of this with the output part.

There are other kinds of control information that can be envisaged. We are investigating whether any of these other kinds of control information should be incorporated in our approach.

## Generating the level 2 VDM specification

At this point it is possible to (automatically) generate a complete, abstract (level 2) VDM specification. This specification can be considered as a textual view of all the information present in both the hierarchy of DFDs, the type information in the entity-relationship diagram and/or the data dictionary, the definition of the mini-specifications, and the control information. Thus, at this step the designer can present the specification in a readable way using two different views.

## Example: the spelling checker component (continued)

In our spelling checker example we are basically modeling a sequential system. Therefore, it is safe to assume that *Split* will be called before *Sort*. The operation *SplitToSort* can therefore be changed as follows:

$SplitToSort\,(file\_name : FileName)\ sorted\_word\_list : SortedWordList$
**ext rd** $docs : Docs$
**post** $\exists\, word\_list : WordList \cdot$
$post\text{-}Split(file\_name, word\_list, docs) \wedge$
$post\text{-}Sort(word\_list, sorted\_word\_list)$

By assuring that the same *word_list* which is the output of *Split* is used as the input to *Sort*, we, in effect, specify a sequential combination of *Split* and *Sort*. A complete and formal transformation from a hierarchy of DFDs to a collection of modules of VDM specifications is given in [Larsen&91a].

### Generating the level 1 structure charts

On basis of the decomposition hierarchy of DFDs, and the control information which has been added, it is possible to automatically generate structure charts which can be used as a graphical overview of the system. However, at this level implicitly defined mini-specifications are considered as primitives, not to be further decomposed.

### Example: the spelling checker component (continued)

The level 1 structure charts for the spelling checker component are shown in Figure 4.

## 4.6 Refine the mini-specifications

The mini-specifications must now be transformed into explicit function and operation definitions (if they are not defined explicitly already). This design step is similar to the reification step in VDM. Having transformed the mini-specifications into explicit function and operation definitions, and assuring an executable subset of VDM-SL has been used (see e.g. [Larsen&91b]), the designer can now execute the specification. This enables the designer to animate the entire specification which may be an advantage if the design is to be shown to an end-user. In this way errors can be corrected before the actual coding of the system is started.

### Generating the level 3 VDM specification

It is now possible to automatically generate a textual representation of the design in the form of a level 3 VDM specification. The level 3 VDM specification will be a refinement of the level 2 VDM specification, provided that the explicitly defined mini-specifications are refinements from the earlier defined implicit ones.

### Example: the spelling checker component (continued)

The following level 3 VDM specification can be defined for the spelling checker component:

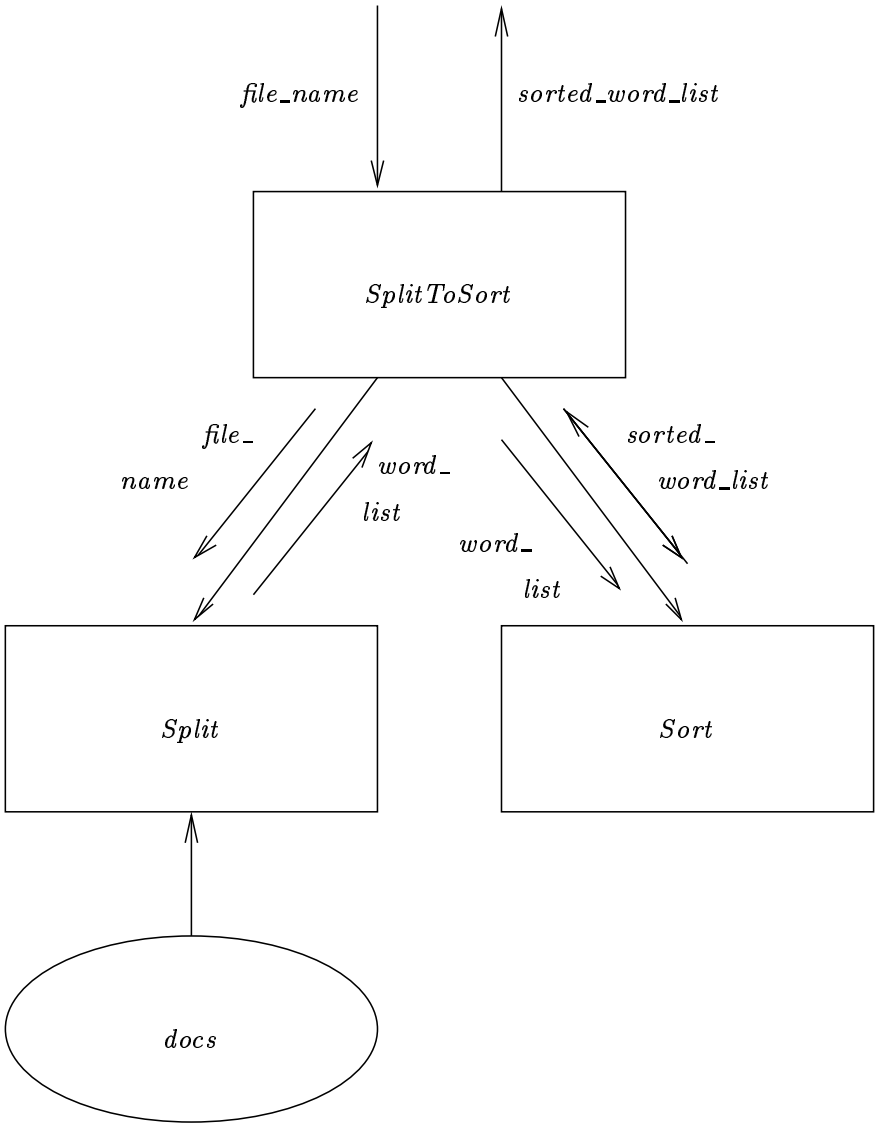


Figure 4: Level 1 structure charts for *SplitToSort*

**operations**

$Split : FileName \xrightarrow{o} WordList$

$Split\,(file\_name) \triangleq$
  **if** $file\_name \notin$ **dom** $docs$
  **then return** $[\,]$
  **else let** $file = docs(file\_name)$ **in**
    **return** $ChangeIntoWords(file)$ ;

$Sort : WordList \xrightarrow{o} SortedWordList$

$Sort\,(word\_list) \triangleq$
  (**dcl** $sorted\_word\_list : SortedWordList := [\,]$,
    $word\_list' : WordList := word\_list$;
  **while** $word\_list' \neq [\,]$

```
    do let word : Word be st ∀ i ∈ inds word_list' ·
                                    LexicallySmaller(word, word_list'(i)) in
        (sorted_word_list := sorted_word_list ⌢ [word];
         word_list' := [word_list'(i) |
                        i ∈ inds word_list' · word_list'(i) ≠ word]);
    return sorted_word_list );

SplitToSort : FileName →o SortedWordList

SplitToSort (file_name) ≜
  (dcl word_list : WordList := Split(file_name) ,
       sorted_word_list : SortedWordList := Sort(word_list) ;
   return sorted_word_list )
```

## Generating the level 2 structure charts

Based on the explicitly defined mini-specifications, the hierarchy of DFDs, and the control information which has been added, it is possible to automatically generate structure charts, which can be used to get an overview of what needs to be implemented, and how the system should be structured in the programming language used. The difference between the two levels of structure charts is that the level 2 structure charts show how the mini-specifications are decomposed, whereas these mini-specifications are considered primitive in the level 1 structure charts. However, it is possible that some designers prefer the level 1 structure charts, because they are less detailed, and so we consider level 2 structure charts as secondary documents.

## Example: the spelling checker component (continued)

The level 2 structure charts for the spelling checker component are shown in Figure 5.

## 4.7 Optimize the design

The system specification so far is largely based on the model of the system that was constructed in the analysis phase. The implementation of the system will be dependent on the characteristics of the programming language used, the target machine and many other factors. During this (optional) step the level 3 VDM specification is adapted to such requirements. Structure charts which also reflect these changes can be constructed as well. The VDM specification that is constructed in this step is called the *level 4 VDM specification*, the structure charts are referred to as *level 3 structure charts*.

## Generating the level 4 VDM specification and the level 3 structure charts

Since level 4 VDM specifications are based on 'informal' knowledge of the problem domain, they cannot be automatically generated. The level 3 structure charts can be derived from the level 4 VDM specification.

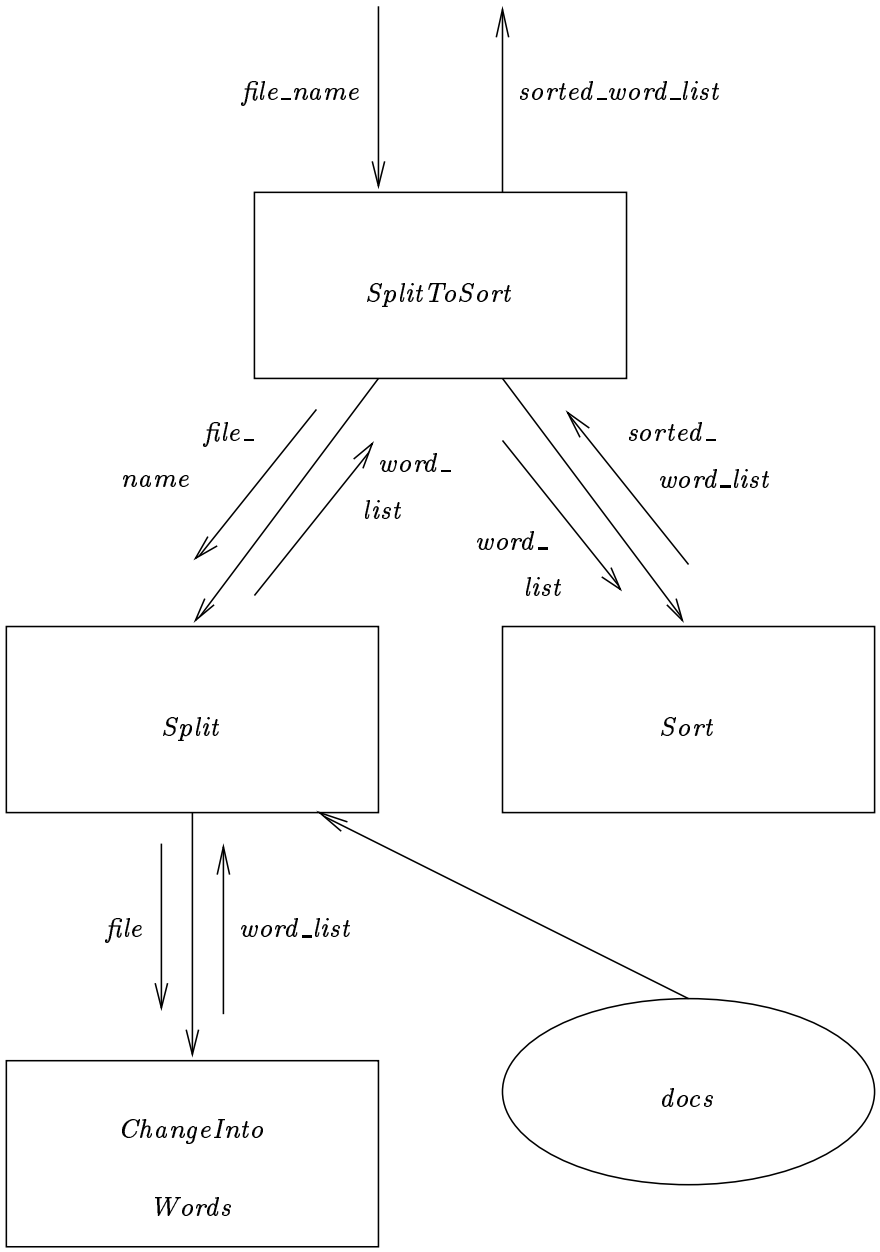


Figure 5: Level 2 structure charts for *SplitToSort*

## Example: the spelling checker component (continued)

An implementation of the operations *Split* and *Sort* in an imperative programming language (and assuming that we model operations as 'procedures' in that programming languages) might cause efficiency and/or space problems, due to the sequences of words which are used as parameters. Therefore, we decide to 'optimize' our design by introducing iteration. To achieve this, we have to change the signatures of *Split* and *Sort* so that no word lists, but just words are passed as parameters, or global variables are used. The bodies of *Split* and *Sort* must be adjusted accordingly.

**types**
*Buffer* = *WordList*;
... = ...

**state** *SpellingChecker* **of**
  *buffer* : *Buffer*
  *docs* : *Docs*
**end**

**operations**

*Split* : *FileName* $\xrightarrow{o}$ ()

*Split* (*file_name*) $\triangleq$
  **if** *file_name* $\notin$ **dom** *docs*
  **then** *buffer* := [ ]
  **else def** *file* = *docs*(*file_name*);
       *buffer* := *ChangeIntoWords*(*file*);

*Sort* : () $\xrightarrow{o}$ *Word*

*Sort* () $\triangleq$
  (**dcl** *word* : *Word*;
   **let** *word'* : *Word* **be st**
        $\forall\, i \in$ **inds** *buffer* $\cdot$ *LexicallySmaller*(*word'*, *buffer*(*i*)) **in**
   *word* := *word'*
   *buffer* := [*buffer*(*i*) | *i* $\in$ **inds** *buffer* $\cdot$ *buffer*(*i*) $\neq$ *word*];
   **return** *word* );

*SplitToSort* : *FileName* $\xrightarrow{o}$ *SortedWordList*

*SplitToSort* (*file_name*) $\triangleq$
  (**dcl** *word* : *Word*,
       *sorted_word_list* : *SortedWordList*;
   *Split*(*file_name*) ;
   **while** *buffer* $\neq$ [ ]
   **do** (*word* := *Sort*() ;
       *sorted_word_list* := *sorted_word_list* $\frown$ [*word*]);
   **return** *sorted_word_list* )

The level 3 structure charts for the spelling checker component are shown in Figure 6.

# 5 Design products

Following SVDM, several design products are constructed. As mentioned earlier, however, not all products are equally important to the designer, and people with a different background may prefer some design products over others. In the end, however, a formal specification at a low level of abstraction is produced. Although SVDM does not address how this latter specification should be transformed into a program, we think that sufficient design information is present to make that transformation quite easy. More practical experience with SVDM on larger case studies is needed to elaborate on this topic.

In Figure 7 we give a short summary of the design products.

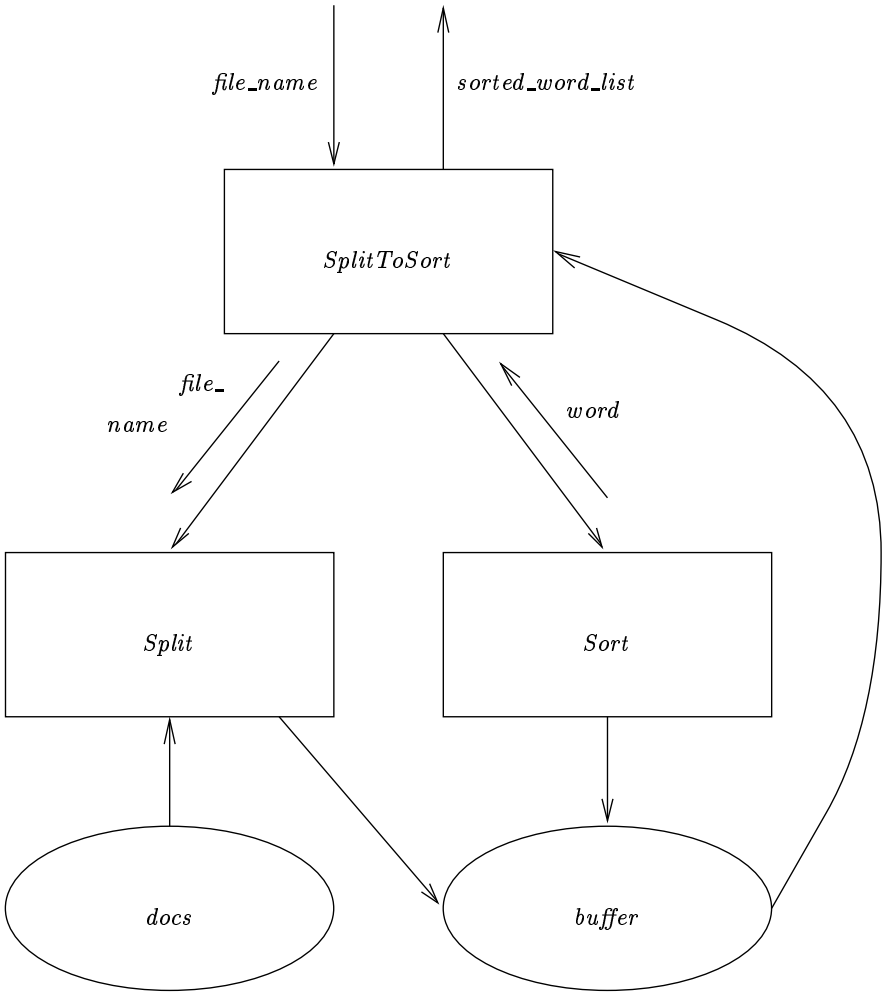


Figure 6: Level 3 structure charts for *SplitToSort*

# 6 Conclusions

In this paper we have presented a method based on a combination of SA and VDM. The ability to analyze a specification from different view points will provide a greater insight in different aspects of the system. In order to make the combined method successful it is necessary to have tool support for changing between the graphical and textual views. Therefore, it is essential to make a further analysis of how such tools can be developed, and what the problems in this area are.

We have defined a precise transformation, expressed in VDM-SL itself, from SA to VDM [Larsen&91a]. This definition can serve as a basis for the implementation of a prototype tool, illustrating the ideas of the combined method.

When tool support has been developed for this combined approach, we would like to have some industrial companies try it out on a few real-life applications, and compare the method to their current practise.

As a further extension of SVDM, it can be interesting to take the inclusion of other notations into account as well.

| *Design product* | *Produced* | *Using* | *Purpose* |
|---|---|---|---|
| DFD hierarchy | by designer | Functional requirements | Model the information flow through the system |
| Level 0 VDM | automatically | DFD hierarchy | Give semantics to DFDs |
| Level 1 VDM | automatically | Mini-specifications | Give semantics to DFDs |
| Level 2 VDM | automatically | Control information | Abstract specification of the system model |
| Level 1 SC | automatically | Level 2 VDM | Show the structure of the system model |
| Level 2 SC | automatically | Level 2 VDM | Show the detailed structure of the system model |
| Level 3 VDM | by designer/ automatically | Explicit mini-specifications | Concrete specification of the system model |
| Level 4 VDM | by designer | Non-functional requirements | Basis for direct implementation |
| Level 3 SC | automatically | Level 4 VDM | Show the structure of the system implementation |

Figure 7: An overview of the design products of SVDM

# 7 References


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

[Constantine&79] L.L. Constantine and E. Yourdon. *Structured Design.* Prentice Hall, 1979.

[Dick&91] Jeremy Dick and Jerome Loubersac. *A Visual Approach to VDM: Entity-Structure Diagrams.* Technical Report DE/DRPA/91001, Bull, 68, Route de Versailles, 78430 Louveciennes (France), January 1991.

[Hall90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.

[Hayes&89] Ian J. Hayes and Cliff B. Jones. *Specifications are not (necessarily) executable.* Technical Report UMCS-89-12-1, Department of Computer Science, University of Manchester, December 1989. will be published in "Software Engineering Journal".

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

[Larsen&91a] Peter Gorm Larsen, Nico Plat, and Hans Toetenel. *A Complete Formal Semantics of Data Flow Diagrams.* Technical Report, Delft University of Technology, July 1991.

[Larsen&91b] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *Submitted to the VDM'91 Symposium*, VDM Europe, Springer-Verlag, March 1991.

[Longworth&86] G. Longworth and D. Nicholls. *SSADM Manual.* NCC, December 1986.

[Myers75] G.J. Myers. *Reliable Software through Composite Design.* Van Nostrand, 1975.

[Plat&91a] Nico Plat, Jan van Katwijk, and Hans Toetenel. *Applications and Benefits of Formal Methods in Software Development.* Technical Report 91-33, Delft University of Technology, Faculty of Technical Mathematics and Informatics, April 1991.

[Plat&91b] Nico Plat, Jan van Katwijk, and Kees Pronk. A Case for Structured Analysis/Formal Design. In *Submitted to the VDM'91 Symposium*, VDM Europe, Springer-Verlag, October 1991.

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

[Ward&85] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems.* Volume 1-3, Yourdon Press, New York, 1985-1986.

[Watt&87] David A. Watt, Brian A. Wichmann and William Findlay. *ADA Language and Methodology.* Prentice-Hall International, 1987.

[Wing90] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Software*, 23(9):8–24, September 1990.

[Yourdon&75] E. Yourdon. *Techniques of Program Structure and Design.* Prentice Hall, 1975.