Experiments with Formal Methods in Software Engineering

door

Nico Plat
1. Gezien de onvermijdelijke, belangrijke rol van formele methoden in de nabije toekomst mag van onderzoeksinstellingen, software producenten en subsidiërende instellingen worden verwacht dat zij structureel in formele methoden investeren. Als tegenprestatie mag van de formele methoden gemeenschap worden verwacht dat zij meer aansprekende resultaten laat zien.

2. Het gebruik van formele methoden vereist een andere manier van denken over software ontwikkeling. Ter ondersteuning van dit proces dienen de volgende maatregelen te worden getroffen:
   - het ontwikkelen van technieken waarin formele methoden en conventionele technieken voor software ontwikkeling zijn gecombineerd;
   - de definitie van toegankelijke industriële standaarden voor formele methoden;
   - het ontwikkelen van professionele software tools ter ondersteuning van het werken met formele methoden;
   - het vervangen van de naam 'formele methoden' door bijvoorbeeld 'exacte modellerings technieken'.

3. Het is een intrigerende gedachte dat voor het ontwikkelen van applicatie software model-georiënteerde formele methoden de voorkeur verdienen boven algebraïsche formele methoden, terwijl de laatste de voorkeur verdienen boven model-georiënteerde methoden als het gaat om de definitie van de semantiek van een taal.

4. De ISO standaard (in wording) voor de specificatietaal VDM-SL is voor de huidige generatie software engineers door het gebruik van complexe formele notatie slecht bruikbaar. De waarde van dit document ligt dan ook vooral in de kennis en ervaring die met het compleet formeel definiëren van een taal is opgedaan.


6. Om de onderzoeksresultaten van een wetenschappelijke groep als geheel te maximaliseren moeten de belangen van de individuele onderzoekers worden geminimaliseerd.

7. Vrijwilligersinstellingen die een zgn. ‘goed doel’ nastreven zijn vaak heilzamer voor de betrokken vrijwilligers dan voor het onderhavige goede doel.

8. De reclamecampagne van uitvaartvereniging AVVL met als slogan “Is er koffie na de dood?” is een stap in de richting van het doorbreken van het taboe op de dood in de westere samenleving.

9. Uit de hernieuwde aandacht voor de ‘autoloze zondag’ binnen de Nederlandse politiek zou de indruk kunnen ontstaan dat alle andere maatschappelijke problemen in Nederland reeds zijn opgelost.

10. Stellingen in proefschriften geven vaak eerder blijk van een grote creativiteit van de promovendus dan dat zij serieus moeten worden genomen. Er is dan ook geen wetenschappelijke reden meer om stellingen te handhaven.

Experiments with Formal Methods
in Software Engineering
Experiments with Formal Methods in Software Engineering

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft op gezag van de Rector Magnificus, Prof.ir. K.F. Wakker, in het openbaar te verdedigen ten overstaan van een commissie aangewezen door het College van Dekanen op dinsdag 21 september 1993 te 16.00 uur

door

Nico Plat

informatica ingenieur geboren te Rotterdam
Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. J. van Katwijk

De overige leden van de promotiecommissie zijn:
Prof.dr. H. Koppelaar Technische Universiteit Delft
Prof.dr.ir. W.L. van der Poel Technische Universiteit Delft
Dr.ir. W.J. Toetenel Technische Universiteit Delft
Prof.dr.ir. K. de Vlaminck Katholieke Universiteit Leuven
Prof.dr. S.C. van Westrhenen Technische Universiteit Delft
Prof.dr. J.M. Wing Carnegie Mellon University
“We do not use formalisms for the sake of being just formal. We use formalisms (1) because they appear to help structure more finely the development, (2) because they are the primary means we know of to help guarantee correctness of software, (3) because developments that have used formalisms have been far more productive, by a 3-5 fold order of magnitude, than developments not using formalisms, and (4) because it is fun, including intellectually satisfying, to use formalism.”

DINES BJØRNER
Published and produced by:
Nico Plat
P.O. Box 3277
NL-2601 DG Delft
The Netherlands
E-mail: Nico.Plat@ACM.org

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Plat, Nico
Experiments with Formal Methods in Software Engineering /
Nico Plat. - [S.l.: s.n.]
Proefschrift Technische Universiteit Delft. - Met index,
lit. opg. - Met samenvatting in het Nederlands.
ISBN 90-9006392-7
Trefw.: software.

©1993, Nico Plat
All rights reserved. No part of this thesis may be reproduced in any form or by any means without prior written permission of the author.
## Contents

**Preface**  
Preface

1 **Introduction**  
1.1 Formal methods in software development  
1.2 Problems with software development  
1.3 The industrialization of formal methods  
1.4 Aim and outline of the research  
1.5 Overview of the thesis

2 **Context**  
2.1 Software specifications  
2.2 Formal methods  
2.3 Classifying formal methods  
2.4 The Vienna Development Method  
2.5 Motivation for the use of VDM

3 **Standardization and support**  
3.1 Overview of the VDM-SL standard  
3.2 Static semantics for VDM-SL  
3.3 The syntax mapping  
3.4 Conformance, compliance and tool support  
3.5 Conclusions
4 The application of VDM in DoD-STD-2167A  
4.1 Overview of DoD-STD-2167A .......................... 77  
4.2 Applying VDM in the context of DoD-STD-2167A .......................... 81  
4.3 Modes of VDM application .......................... 92  
4.4 Conclusions ....................................... 95  

5 The application of VDM in combination with SA  
5.1 Overview of Structured Analysis .................. 101  
5.2 The transformation of data flow diagrams .................. 102  
5.3 Methodological aspects of combining SA and VDM .................. 127  
5.4 Conclusions ....................................... 144  

6 Case study: a lift system  
6.1 Introduction of the case study .................. 148  
6.2 The analysis phase: functional aspects .................. 150  
6.3 The analysis phase: concurrent aspects .................. 161  
6.4 The design phase ................................ 164  
6.5 Evaluation ........................................ 171  
6.6 Conclusions and future research .................. 173  

7 Evaluation  
7.1 Standardization and support for VDM .................. 177  
7.2 The application of VDM in DoD-STD-2167A .................. 179  
7.3 A combination of SA and VDM .................. 179  
7.4 Overall evaluation ................................ 180  

References ........................................ 183  

Index ........................................ 201  

Summary ........................................ 209  

Samenvatting .................................... 213  

Curriculum Vitae ................................... 217
Preface

In the past four decades computers and computer software have become an important factor in our society. Many devices used in our daily lives depend for their functioning on software, and technological advances could be made through the use of software. Software technology has also become crucial to business success. Organizations depend on software for their profitability and their long-term competitive survival. At the same time, the computer field itself is rapidly changing. We have seen exponential growth over the past four decades. These changes and growth have not only changed the world we live in, but have raised our expectations, leading to increasingly ambitious systems.

The motivation for the development of the first computers was the need to solve scientific calculations. The designers of these machines discovered the benefits of storing programs in the same way data was stored. This idea made it possible for a programmer to turn a general-purpose computer into a system which would behave like a special-purpose machine especially engineered to solve a given problem. The magnitude of the programming task required by a large application caused users to devote more of their time to this topic than to the hardware needed. Lumping all their problems together, they began to refer to ‘software’ when they were talking about a system aspect that was not hardware or management. The term ‘software’ has appeal: it
offers an appropriate contrast to hardware. Software is less tangible and easier to modify than hardware. Software has no mass, no volume, no color, no odor – no physical properties, and is therefore not subject to the laws of gravity or the laws of electro-dynamics. Code is merely a static image of a computer program, and while the effects produced by a program are often observable, the program itself is not. Software does not degrade with time as hardware does. Software failures are caused by design and implementation errors, not by degradation. However, these very attributes have led to severe problems. By accepting the hardware as fixed and immutable, users have forced the software to absorb all the changes needed to make the hardware fit the problem.

The costs of software development are immense. In The Netherlands alone, currently $0.8 billion each year is spent on software development (source: COSSO). Although precise, up-to-date figures are very difficult to establish, it has been suggested [Boehm 87] that in 1985 worldwide software costs were over $140 billion. Even small improvements in software productivity can therefore result in a significant reduction of absolute costs. The annual growth of world wide spending on software is currently estimated at 12%. Figure 0.1 shows the increase in the spending on software world wide and in the USA (source: [Boehm 87]). While these figures are impressive, they do not fully convey the growing

![Figure 0.1: Software cost trends](image)

importance of the software industry in modern society. As software systems become more numerous and more critical to post-industrial
societies, the demand for high-quality software increases at a steady rate.

The recognition that systematically producing large-scale software systems is an engineering discipline took place at a NATO conference held in Garmisch (Germany) in 1968 [NATO 69]. Initial experience in building large software systems showed that existing methods for software development were inadequate. On this occasion the term 'Software Engineering' was first introduced, thus stressing the consensus that an approach to producing software was needed which uses engineering principles in the process of development, made up of both technical and non-technical aspects.

Software Engineering and traditional engineering disciplines share the pragmatic approach to the development and maintenance of technological artifacts. Software Engineering is not just about producing software, but also means producing software in a cost-effective way. Given unlimited resources, most software problems can probably be solved, but the challenge for the software engineer is to produce high-quality software with a finite amount of resources and to a predicted schedule.

Since the 1968 NATO conference, many developments have taken place in the field of software engineering: the introduction of high-level programming languages, the application of structured analysis and design methods, the use of computerized support for software development, etc. Each of these developments has brought progress – sometimes little, and sometimes very significant – in the direction of being able to control and predict the process of producing software, and has extended the degree of complexity of software systems that we are able to build today. None of these developments, however, has produced the ultimate answer – or, to speak in more popular terms: none of these has proven to be a 'silver bullet' – in developing extremely reliable software at extremely low cost. Probably such a silver bullet does not exist.

There is, however, a more fundamental question to be answered here: what is the basis of the process of producing software? Evidently, at the basis of producing something (in this case software) that has its
roots in mathematics, there is also a process which has its roots in mathematics. This link to mathematics can, in software development methods, be found in so-called ‘formal methods’: methods with a sound (formal) basis in mathematics that may be used for the specification and verified development (i.e. provably correct) development of software systems. It is very likely that the use of these formal methods will play a vital – if not fundamental – role in the further development of software engineering in the next 10 years.

Interestingly enough, the development of formal methods and the discipline of software engineering have been rather isolated from one another: whereas software engineering has been embraced in both academic as well as in industrial circles, formal methods have so far mainly drawn academic attention, with much emphasis on their underlying theories, but little emphasis on pragmatic aspects of formal methods. Consequently, the industrial world has so far been rather hesitant in taking up formal methods. Making a contribution to bridging the gap between these two worlds is not only an interesting, but also a very challenging task. Roughly spoken, the aim of the research project reported on in this thesis is to make a contribution in that direction, by exploring the issues that might prevent formal methods from being used in the much wider context of industrial software development.

Acknowledgments
The research for, and the writing of this thesis could not have been completed without the support of many people. First and foremost I would like to thank my supervisor: Jan van Katwijk. He has supported and stimulated me from the first moment I met him, first as a M.Sc. student, and later as a Ph.D. student. His direct approach to computer science and related subjects has significantly influenced my way of thinking. Second on the list is Hans Toetenel, whose enthusiasm and creativity have kept me going at the moments it was really necessary. I hope that our friendship will further develop, not in the least by cultivating our joint interest in good food! Thanks also go to Kees Pronk, who has proved to be a reliable and consistent advisor; his eye for detail has improved the final result of my work. The SEPC group led by Jan van Katwijk as a whole owes my gratitude for providing a pleasant
work environment, essential for a 4 year continuous period of work. Special thanks are due to my Danish colleague and friend Peter Gorm Larsen. Our mutual interests have resulted in several joint research papers – in fact, parts of the material in chapters 3 and 5 of this thesis is joint work. I have found working together with him very stimulating and I hope this will continue in the future. I thank the members of BSI Panel IST/5/-/19 (chaired by Derek Andrews) for their part in the instructive experience of working on the VDM-SL standard. I thank the CEC (DG XIII) for funding a short study visit to Denmark in August 1989, and the NWO (Nederlandse organisatie voor Wetenschappelijk Onderzoek) for partially funding my attendance of the Methods Integration conference in Leeds (UK) in September 1991. Ton Biegstraaten was always available to give me expert advise during the combat with \LaTeX\ (used for typesetting this thesis), for which I owe him thanks. I thank Hub Engelen and Onno Roep for doing their best to keep our computer systems up and running; dealing with a moody person like myself when things went wrong must not always have been easy. My gratitude also goes to our secretarial staff, Trudie Stoute, Toos Brussee and Coby Bouwer, for their cheerful presence and their help with some of the clerical work in the later stages of preparing this thesis. I thank Koos Schonewille for his aid in designing and preparing the cover of this thesis. I thank Eugène Dürr and Steef Geenen for allowing me to finalize this thesis while already working for CAP Gemini Innovation. The following persons have contributed, either by discussion or by commenting on earlier published papers, to the work presented in this thesis: Michael Andersen, John Dawes, René Elmström, Ronald Huijsman, Andy Lovering, Gertjan van Oosten, Hans Tonino, and Marcel Verhoef. Thanks! Last but not least I want to thank my parents, Jan and Nees Plat, who have constantly supported and encouraged me, and without whom I wouldn't be what I am today.

Nico Plat,
Delft, August 1993.
Introduction

The problems involved with the systematic production of large-scale software are numerous and hard to tackle. Many improvements have been proposed over the years, some of which have proved their value, whereas others have been silently discarded again. One technique that is steadily growing in importance is the application of methods with a sound basis in mathematics, so-called formal methods. The link with mathematics makes that formal methods have valuable properties which make the verified development of software systems possible. The global aim of this thesis is to increase the understanding of the role and limitations of formal methods in software development, in particular their relationship to other (non-formal) methods, techniques and models for software development.

This chapter starts with a brief introduction to formal specifications and formal methods, for a more comprehensive introduction we refer to [TSE SE-16(9)] and [Software 7(5)]; companion issues of two journals on this topic recently published by the IEEE. Then, in section 1.2, an overview is given of what can be considered as major problems in current software development practice, and it is suggested how the use
of formal methods can contribute to solving these problems or limiting
their effects. In section 1.3 we discuss the obstacles in the industrial
acceptance of formal methods. In section 1.4 the research topic of this
research project is introduced. Section 1.5 presents an overview of the
contents of this thesis.

1.1 Formal methods in software development

1.1.1 Formal specifications

Mathematical techniques have contributed significantly to the develop-
ment of computer science. An example is the introduction of high-level
programming languages in the late 1950s. Initially, these languages
were defined using natural language, but such definitions can lead
to different interpretations of meaning, and thus also to different im-
plications of compilers for these programming languages. More
precise means of defining programming languages were pursued and
found when Backus introduced a scheme for the syntactic definition of
programming languages [Backus 59, Naur 63] known as BNF (Backus
Naur Form).

The use of mathematics to describe the semantics (i.e. meaning) of pro-
gramming languages has its roots in the early work of Floyd [Floyd 67]
and Hoare [Hoare 69]. Their work on axiomatic semantics – a tech-
nique for describing the semantics of a programming language by us-
ing mathematical statements to describe the required effect of each
construct provided by the language – led to the introduction of formal
specifications to describe the effect of programs in mathematical terms.
Formal specifications are texts expressed in a mathematically based
language, which can, therefore, be used to provide a precise and unam-
biguous description of the expected behaviour of a program, in this way
introducing possibilities for proving properties of such a program. The
use of formal specifications in software development is usually called
the use of a ‘formal method’.
1.1.2 Formal methods

Formal methods can be described in different ways. De Roever defines formal methods as "that branch of research in the foundation of computer science which deals with modeling and reasoning about (properties of) sequential and distributed systems" [de Roever 91]. De Roever emphasizes in his definition the possibilities for reasoning about systems, but does not elaborate on any methodological aspects. Landwehr et al. describe a formal method as having "an effective procedure for determining whether [...] the method has [...] been correctly applied" [Landwehr 91]. In this definition, emphasis is placed on checking the application of a method. No mention is made, however, of whether the effective procedure should be based on mathematics or not. Wing describes formal methods 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." [Wing 90]. Formal methods thus essentially are formal systems; they provide frameworks to inspect the satisfiability of specifications, to prove the correctness of an implementation of a system, and to prove properties of systems without the need to have an executable representation of the system.

The word 'method' implies that a formal method includes techniques to apply its mathematical basis, and guidelines to do so systematically, along with a set of general principles. In [Bjørner 89], Bjørner describes the notion of method as "a procedure with selection criteria for choosing among and using a number of techniques and tools in order to efficiently achieve the construction of a certain efficient artifact". The guidelines and principles together provide a design philosophy according to which specifications can be developed into implementations. Such a design philosophy will normally include principles which are also found in non-formal methods, like decomposition, refinement, etc.

Thus, formal methods are methods for the specification and verified development of software systems using formal specifications. They potentially enable larger software systems to be developed than can be done nowadays: they provide for complexity control by offering abstraction facilities and they make unambiguous and consistent specifi-
cations possible at lower cost, and enable the verified design of software components. This approach also provides a framework for organizing the mental process of design that can be taught and transferred. Systematic approaches are much easier to teach than those that rely on common sense without much explicit guidance. Mathematics provides a set of general concepts that can be used as building blocks in solving many different kinds of problems. Recognizing that a particular problem in software development is a special case of a familiar general structure can help to impose order and simplicity on a maze of details. This is important because conceptual complexity imposes the ultimate limits on the software systems that can be built.

1.2 Problems with software development

To get some appreciation for why formal methods may be useful in industrial software development, we shall discuss a number of problems with the controlled production of large-scale software and their suggested underlying causes. The enumeration is certainly not complete, but the most important problems are covered.

A first observation is that there frequently is a difference between the planned and actual progress of a software project. Several surveys and investigations have been carried out in this field [Jenkins 84, Phan 88, Phan 90, Genuchten 91a, Genuchten 91b], which show that software is indeed often delivered late, and that project budgets are often overrun. The survey of Phan (encompassing 191 organizations) investigated the frequency with which cost overruns and late deliveries occurred (figure 1.1). The average cost overrun was 33%. The results of the other surveys confirm these figures.

The surveys also showed that the reasons for delay in delivery are not primarily technical in nature, but are far more related to organizational, managerial and human aspects; over-optimistic planning was most often mentioned as the main reason for the overrun in lead time and in budget.

A second problem is that Software costs, both in financial and in hu-
Figure 1.1: Prevalence of cost overruns (a) and late deliveries (b)

In man terms, are increasing rapidly as the computerization of society spreads. Software in large systems may approach $4$ billion in initial costs alone [Charette 86]. Software costs are the dominant costs of software-based systems today. The largest part of the costs involved in the production of software are the costs of manpower. The size of most software systems built today is such that the effort necessary to build these systems is gigantic as well. The development of the IBM 360 operating system required over 5000 programmers for five years to create the million object instructions necessary for it to run [Brooks 75]. The unfortunate trend, however, is that although in the past decades programmer productivity has yearly gone up by 4% [Dolotta 76], and the number of programmers has also gone up yearly by 4%, the demand for software yearly rises by 12%.

A second major reason adding to the expense of software are the errors introduced in the course of the software development process. According to Charette [Charette 86], 64% of all errors are introduced during requirements specification and design, and 36% of the errors are introduced during the implementation phase. Figure 1.2 shows the (cumulative) costs to fix errors during the various phases of the life cycle (source: [Boehm 81]).

Given these figures, it seems reasonable to assume that the cost of software will go down, when (a) the productivity of the people involved
in the development of software is increased, (b) the introduction of errors can be prevented as much as possible, and (c) errors can be detected as early as possible.

By far the most worrying problem associated with the production of software is the fact that software often does not meet the requirements originally established by the customer. Two types of requirements can be identified:

1. **Explicit requirements.** These are the requirements that are explicitly negotiated over between a customer and a software producer, written down in a software requirements document (SRD). An SRD can be regarded as a contract between the customer and the software producer.

2. **Implicit requirements.** These are the qualities that the software product is assumed to possess, without being explicitly expressed. A typical example is that we assume a software product to be correct in some sense. When the situation or the application domain demands that particular emphasis is put on such a requirement, it is often changed into an explicit requirement.

Due to the complexity of the software systems built nowadays, it is very difficult to perform validation and verification in a cost-effective way. Errors can remain undetected because current validation and
verification techniques can only in simple cases give a 100% guarantee that the delivered product conforms to both the implicit and the explicit requirements. Testing is, nowadays, the most often used verification technique. Unfortunately, when the complexity of the application increases, testing soon becomes infeasible because the number of test cases becomes too large. Thus, there are both economical and theoretical limits to using testing as a technique for ensuring 100% conformance to the requirements.

Explicit requirements have the additional problem that a specification of a problem must correctly and adequately represent the customer's requirements. This is difficult to ensure, because a developer must rely on the customer's description of these requirements which may be inaccurate, incomplete, and subject to change. Detailed requirements analysis can help to identify the customer's requirements, but at present there is no way of ensuring that a specification actually fulfills these requirements.

With respect to the problems with current large-scale software development, formal methods potentially provide:

- a means to reduce the cost of software\(^1\), by reducing the introduction of errors (formal verification) and by moving the stage at which errors are most likely introduced to the beginning of the software process (formal modeling);
- a means to ensure that the software system under development will satisfy its requirements.

To our knowledge, there are no convincing arguments that formal methods have a positive influence on the on-time delivery of software systems.

\(^1\) Although it has never conclusively been shown that formal methods reduce development costs, most people believe that this is the case, as e.g. indicated in a survey by Austin et al. [Austin 93].
1.3 The industrialization of formal methods

The recognition of the importance of formal methods is also growing outside academic circles. In the UK for example, the use of formal methods has now become obligatory for the development of safety-critical systems for the MoD (Ministry of Defence). The requirements for the development of such systems are defined in interim defense standards 00-55 [Defence Standard 00-55] and 00-56 [Defence Standard 00-56]. Pilot projects in which formal methods are used have been carried out or are carried out in industry by a few companies. Some isolated cases of larger efforts are known (e.g. the formal specification in Z of IBM's transaction processing system CICS [Hayes 85, Houston 91]), but the use of formal methods to date in commercial situations has been limited. The effective transfer of a new technology – like formal methods – into industry is a complex affair which must be handled with care. New technologies are not only required to be effective, but also have to solve a perceived problem and have to be acceptable to their intended users.

Hall presents in [Hall 90] a comprehensive overview of several problems - or to be more precise: what appear to be problems - with the wider industrial acceptance of formal methods. Coleman and Cheng reflect on their personal experience with the introduction of formal methods in industrial organizations in [Coleman 90, Cheng 92]. Norris and Stoc- km a n report on a survey that was conducted among software developers working for British Telecom [Norris 89]. Craigen et al. [Craigen 93a] provide a survey on use and problems of formal methods in 12 applications; a summary of their findings can be found in [Craigen 93b]. A more extensive survey is presented in [Austin 93], comprising both a literature survey as well as an analysis of questionnaires returned by 126 organizations, mainly in the UK.

In the following subsections some of the problems as discussed in these experience reports and surveys will be addressed in detail.
1.3.1 The role of formal methods in the software development process

The application of formal methods in software development potentially increases the effectiveness of the software process and the quality of the developed software systems. Software processes, however, tend to be quite complex: they consist of different phases, during which a wide variety of activities are performed. Because the systems that are developed are usually large, many people are involved. The hierarchy in the project organization that is thus necessarily created involves different tasks, and also different views of abstraction on the software process. Management, for example, has interest in monitoring the progress of a project, and needs a view on the software process which provides them with check points at which they can make decisions on how the team should proceed. Standards for software development often fulfill this need. Individual project members (whether they are programmers, system analysts, or have any other function does not really matter) need a view on the software process that provides them with methods, techniques, and procedures for performing their tasks.

Although theoretic issues with respect to formal methods as such are being further developed at a steady pace, questions remain on the stage in the software development process during which formal methods should be applied, and on the benefits of applying formal methods for the total development process. An insufficient understanding of the relationship between the software process and formal methods hinders the actual application of formal methods for the development of ‘real’ systems to a large extent, as argued in [Norris 89], [Wood 90] and in [Coleman 90]. It is, therefore, necessary to investigate how formal methods fit in the ‘software life cycle’ and to analyze how formal methods can be used together with techniques and methods that are currently used in industry [Craigen 93a, Austin 93].

1.3.2 Acceptability to non-experts

When formal specifications are presented to non-experts (this could e.g. be a customer for whom a software system is being developed) problems may arise because the non-expert cannot (fully) understand and
appreciate the formal specifications that are produced. It has been suggested that the acceptability of using formal methods for non-experts can be increased by:

- using natural language in combination with formal specifications. The specification of the system in natural language can then be regarded as annotations to the formal specification;
- using executable formal specifications. Such specifications can be used as prototypes to animate the behaviour of the system. Of course, such an approach puts constraints on the degree of abstraction that can be employed, and it is not clear whether the advantage of increased acceptability to customers outweighs the disadvantage of loss of abstraction; debates on this issue are ongoing [Hayes 89, Larsen 91].

Although no empirical evidence has been published that support these assumptions, it seems likely that both techniques will improve the acceptability of formal methods to people without the necessary mathematical background.

1.3.3 Education

The application of formal methods requires some knowledge of discrete mathematics and logic, and the majority of the current generation of software developers - and their management - lack this knowledge [Coleman 90]. Many other engineering disciplines have a mathematical basis on which they rely for the construction of products, and consequently only designers familiar with this mathematical basis are qualified to construct these products. Unfortunately, most designers working in the software industry are more familiar with classical (informal) techniques than with the underlying mathematics of computing. It is anticipated that the education problem will become less important when a new generation of computer science graduates with the appropriate background start working in industry. At Delft University, first-year computer science students are now being taught the basic principles of formal methods as part of their introductory course on programming.
1.3.4 Automated support

It is unlikely that large formal specifications without any errors - necessary to be able to reason about them - can be constructed without the use of automated support (tools), e.g. syntax checkers and and type checkers. The same holds for reasoning about such formal specifications. Non-trivial proofs are too complicated to be carried out depending on human intelligence alone; proof assistants can be used to verify that the logic rules have been applied correctly.

Although it seems easier to construct computer-based tools for something 'formal' than for something 'informal', and although e.g. an investigation by the author [Plat 89] showed the existence of at least 16 different tools and tool sets to support the application of the formal method VDM (see section 2.4) alone, the current availability of tools (both in terms of quantity and quality) is still limited [Craigen 93a]. Until recently, most support tools for formal methods were the experimental products of academia, which tended to be useful for small applications only. Currently, a small market for formal method tools seems to be developing and product-quality tools are becoming available; see e.g. the overviews as given in [VDM'91] and [FME'93].

At Delft University of Technology, a contribution to this process has been made by means of the development of a syntax checker and static semantic analyzer for VDM [Plat 88, Plat 91a].

1.3.5 Standardization

Standardization is an important issue in all branches of industry. Currently, no standards for formal methods exist, but there are a few examples of standards for formal specification languages. A standard for a formal specification language stimulates the wider use and acceptance of that language, because:

- is an indication of the maturity of the language and of the (international) support that exists for the language. This maturity reduces the risk for a company to invest in using the language.
- serves as a reference document for the language. As such, it will lead to a decrease in the wild growth of 'dialects' of the language,
which improves the efficiency with which the language can be used by different parties.

- will give an impulse to the development of automated support for using the language, because the market of potential users is increased by the very fact that there is a standard.

In the telecommunication industry the need for standardized specification languages is nowadays recognized. Within ISO (the International Standards Organisation) two standards have been developed for such specification languages (LOTOS [ISO8807], and ESTELLE [ISO9074]). The CCITT (Comité Consultatif International Téléphonique et Télégraphique) has standardized the formal specification language SDL (CCITT 88), which is used to describe distributed systems. Besides these examples, standards are currently also under construction for the formal specification languages of Z (based on [Spivey 92]), and VDM (see section 2.4). Apart from these examples, standardization of formal specification languages is – so far – very uncommon.

1.4  Aim and outline of the research

Bringing the worlds of ‘conventional’ software engineering and formal methods together is not only a very challenging area, but also validates the very existence of formal methods; it cannot be expected (and is for obvious reasons not desirable) that formal methods will completely replace the more conventional techniques for software development, but they can be regarded as important improvements of existing techniques. The gradual introduction of formal methods in current software practice is, therefore, an important research topic.

The global aim of the research project reported on in this thesis is to increase the understanding in the relationships between formal methods and more conventional software engineering by studying the problems described in the previous section. We have chosen to approach these problems by performing a number of experiments (each involving the use of the formal method VDM (Vienna Development

2. The problems with respect to education will not be addresses, however; these lie beyond the scope of this research.
1.4. Aim and outline of the research

Method) [Bjørner 78, Jones 90a]; a motivation for this choice is presented in section 2.5), and to analyze the outcome of the experiments with respect to their relationship with conventional software development, in terms of more generalized results. In this way, we hope to make a step in the direction of a model for software development in which both formal methods as well as current software development techniques have an appropriate place.

The first ‘experiment’ that we choose was to work on a standard for a formal specification language, in this way making a contribution to solving the problems in the area of standardization and tool support for formal methods. The standardization we got involved in was that of VDM-SL, the formal specification language of VDM. We also developed some basic support for using VDM: a type checker\(^3\). The analysis of the results of this contribution addresses to what extent the VDM-SL standard can be regarded as a useful standard for industrial companies.

The second and third experiment address the relationship between formal methods and conventional software engineering. Current software engineering practices have proven their value in real-life software development. It would be unwise and unrealistic to change that practice from one day to the other drastically. Nevertheless, an increased degree of formalization may provide better control over the software process, and therefore it is important to find out how this can be achieved gradually.

With respect to this issue we choose to do two experiments. The first addresses the embedding of formal methods in a conventional software development model. The experiment consists of an analysis of the usefulness and problems that occur with the application of VDM in a software development model that is widely used and for some organizations even is obligatory: DoD-STD-2167A [DoD-STD-2167A]. The experiment aims to examine at which places VDM can effectively be applied, what VDM can best be applied for, and what improvements can be envisaged for a more effective application of formal methods in models such as DoD-STD-2167A.

---

3. The details of the type checker are not discussed in this thesis, however.
The second of these two experiments addresses the more technical issues of combining formal methods with (non-formal) 'structured' methods for the analysis and design of software. Structured methods are widely used in practical software development, and the current generation of software engineers has much experience with such methods. Consequently it is worthwhile to examine if suitable combinations of formal and structured methods can be constructed, which may provide an acceptable way of introducing formal methods into development processes that are otherwise mostly informal, and which may provide the benefits of both kinds of methods. Such an approach may also increase the acceptability of formal methods to non-experts. This experiment is implemented as the construction of such a 'combined' method, based on Structured Analysis [Yourdon 75, DeMarco 79, Gane 79] and VDM. The method is illustrated through a case study, in which a simple lift system is developed.

1.5 Overview of the thesis

The remainder of this thesis is organized as follows.

Chapter 2 describes the context in which the research reported on in this thesis was carried out. This context includes a more elaborate description of the concept of 'formal method' than was done in this chapter. Furthermore, a classification of formal methods is presented which, in contrast to the traditional classification based on the semantic basis of formal methods, is application induced. The chapter concludes with an overview of VDM, and a rationale for why VDM was chosen as the formal method used for the research experiments.

In chapter 3 the first research experiment is described, addressing the standardization and support for formal methods. Several issues with respect to the standardization of VDM are touched upon, in particular those related to the static semantics of VDM's specification language and to language conformance and compliance.

In chapter 4 the second research experiment is described, addressing the use and problems associated with the application of VDM in a
widely used standard for software development: DoD-STD-2167A. In this way, a better understanding can be achieved in the exact role of formal methods at a more global level of the software process.

Chapter 5 discusses the third experiment: the combination of formal methods with structured methods. In this third experiment, the relationships between data flow diagrams and VDM specifications are analyzed, and a promising combination of VDM and Structured Analysis is examined. Chapter 6 contains a small case study, illustrating the advantages and disadvantages of the combined method as defined in chapter 5.

Finally, in chapter 7 the research project is evaluated and some directions for future research in this area are outlined.
In chapter 1 one the central theme of this thesis – formal methods – was introduced. In this chapter formal methods will be further discussed; a definition is given and requirements that can be put on them for a useful application are analyzed. From these requirements a classification scheme for formal methods can be derived, which – in contrast to the more traditional classification scheme based on the difference in semantic foundation between formal methods – is based on the type of support formal methods provide for what can be regarded as ‘elementary transformation steps’ in the software process. In combination these topics form the context of the research described in this thesis.

2.1 Software specifications

When discussing the process of developing software, it is convenient to have a model for this process that provides a framework against which defects in the process and solutions for these defects can be identified. The model used in this thesis is based on the following assumptions:
• The process starts with the recognition of a need for a software solution and ends with the implementation of a software system that satisfies that need.

• The process is initiated by a customer (needing a software system), and is carried out by a software developer. Between these two parties a 'contract' must exists, which defines the requirements that the software system must fulfill. Usually these requirements include functional requirements, constraints, etc.

• The size of the software system is large; although the quantification 'large' is somewhat subjective, the development of 'small' software systems is not considered very interesting in this thesis.

These assumptions form the motivation for the presence of the following components in the software process model:

• A problem. This is something in the mind of the customer, possibly vague and incomplete, for which he seeks a software solution.

• A problem representation. There must be a mutual understanding between the customer and the software developer of the requirements on the software system to be developed. These requirements are a reflection (usually written down in a so-called software requirements document (SRD)) of the problem in the mind of the customer. The problem representation thus serves as the contract between the customer and the software developer.

• A solution representation, i.e. the implementation of the software system. The construction of such a solution representation is the primary aim of the software process. The solution representation must satisfy the problem representation.

• A solution. The solution is the use of the software system that has been developed on a computer system. The solution must solve the problem of the customer.

---

4. This assumption is less trivial than it seems: whereas in the past the development of a software system used to be an isolated effort, software projects are now better characterized as continuous efforts of organizations; instead of building a system from scratch, existing software is enhanced to reflect changed or added requirements. However, to make such an approach possible, it is important that during the original effort in which the system is created, sufficient attention is paid to the specifications that are produced. Enhancement can then be based on the appropriate specification.
2.1. Software specifications

- A transformation process, i.e. the process during which the problem representation is transformed into the solution representation by the software developer.

- A control process. Since the software system under development is large, it cannot be developed by one person alone. Instead, teams of developers – possibly organized in some hierarchy – are needed, who must be coordinated. Usually, this process of control is called ‘management’.

The model is shown in figure 2.1.

Figure 2.1: A transformational model for the production of software

2.1.1 Informal software specifications

Because the transformation from a problem representation to a solution is – in general – complex, the transformation process is usually divided into a series of smaller transformations, where each individual transformation addresses a specific part of the problem and the changes are such that the transformation itself can be controlled and it can be verified that the transformation that was made is actually
correct. Such smaller transformations will be called elementar y trans-
formation steps (figure 2.2), because it is assumed that the transfor-
mation cannot be decomposed into even smaller transformations, or
because it is not considered useful to decompose the transformation
into smaller ones.

![Diagram showing transformation process]

Figure 2.2: An elementary transformation step in the software process

Each elementary transformation step has the following elements:

- a source specification. A source specification describes the start-
ing point of an elementary transformation step.
- a target specification. A target specification is a description of the
  result of an elementary transformation step.
- a transformation. A transformation is the activity of transform-
ing a source specification into a target specification.
- a one-dimensional verification. A one-dimensional verification is
  the activity of verifying that a target specification is internally
  consistent.
- a two-dimensional verification. A two-dimensional verification is
  the activity of verifying that a transformation from a source speci-
fication to a target specification has been correct. What 'correct'
  means in this context depends on the kind of transformation that
  has been made. In general, a verification activity will consist of
testing and/or proving correctness.

A source specification is a specification of the intended characteristics
of the target specification. This means that the source specification is
an abstraction of the target specification, given at a level of detail which
is suitable for the specific stage of development. The target specification
is the result of the transformation; during the transformation activity,
details are added to the source specification (usually under guidance of some ‘design method’) which make that source specification more concrete. The target specification is, therefore, an implementation of the source specification, and will simultaneously serve as a source specification for the next transformation. The chain of development steps that is created in this way provides a path from an abstract specification of the system (in general: the system’s requirements) to a concrete implementation (in general: the executable code of the target software system), during which a number of specifications are created that can either be seen as specifications for their successors, or as implementations of their predecessors (figure 2.3).

Thus, software development is in this thesis regarded as a process of constructing an abstract specification of a customer’s problem, and subsequently transforming this specification in a number of steps into a concrete specification of the solution. Suitable specifications have a representation which is suitable for this transformation process and which have the appropriate abstraction mechanisms to deal with the complexity inherent in large software systems.

### 2.1.2 Requirements for notations for specifications

The way specifications are represented is important because specifications are communicated between different parties (human and non-human). Therefore, a first requirement of a specification notation is that it allows unambiguous communication, in this way making the communication more effective; it is desirable that such notations not only have a defined syntax (i.e. a well-formed reflection on paper, computer screens, etc.) but are also interpreted by everybody in the same way. To ensure the latter, they must have a well-defined meaning (se-
mantics); although specifications are not reality itself, they have a meaning which associates them with reality.

A second requirement of a specification notation is that it must provide facilities for expressing abstraction. Source specifications can be regarded as abstractions of their respective target specifications. They focus on specific points of attention which are relevant for the problem under consideration, and do not touch upon details which are not relevant. Abstraction can be regarded as one of the most important technical tools to control the complexity degree of the problem.

Several different kinds of abstraction can be envisaged:

- **Separation of concerns.** Specifications sometimes describe different, orthogonal aspects, e.g. functional aspects, architectural aspects, control aspects or the architectural aspects of the system. These aspects can be specified (and sometimes be developed) in isolation, so that at the appropriate stage, attention can be focussed on those aspect of the system that are important at that stage. Separating these different orthogonal aspects from one another is called separation of concerns. In the class of formal specification languages called ‘programming languages’, such aspects are usually implicitly combined, e.g. the flow of control is intertwined with the functional aspects of the language.

- **Decomposition.** When it is possible to express required aspects of a system in terms of subcomponents, i.e. the subcomponents together with the relationships between them are adequate descriptions of the total system, then we call this decomposition. Decomposition is a form of abstraction because the details of the subcomponents are not (necessarily) given.

- **Representational abstraction.** The execution of a computer program can be regarded as a series of manipulations of data. The format of this data is described in terms of data structures. Originally, these data structures and the way they are accessed were very much based on the underlying machine model. As such they matched the reality of computers, but not the reality of the 'real world'. Representational abstraction is a form of abstraction in which an attempt is made to use data structures which are as close to the real world as possible.
2.1. Software specifications

- **Operational abstraction.** This kind of abstraction enables the removal of irrelevant detail from the description of the effect of actions on certain entities. Two important kinds can be distinguished:
  1. The use of **pre- and post-conditions.** Pre- and post-conditions describe desired effects in terms of input/output relations, i.e. they describe what should be done without describing how the effect should be achieved. Pre- and post-conditions tend to be concise descriptions of required behaviour, and have no specific bias towards various kinds of possible implementations.
  2. **Looseness.** Looseness is a description technique that allows the expression of a range of legal results. In the transformation from a source specification to a target specification this means that more than one valid transformation exists.

These two requirements, unambiguity and facilities for expressing abstraction, are considered to be the main features of specification languages in this thesis.

2.1.3 Formal specifications

A category of specifications that satisfies the requirement of precise and unambiguous descriptions of intended characteristics is that of **formal specifications.** Preciseness by formal specifications is achieved through the use of a mathematically defined notation for the representation of a specification, and absence of ambiguity through the fact that the meaning of a formal specification is defined in terms of an underlying mathematical model, i.e. the interpretation of such a specification is unambiguous with respect to the underlying mathematical model.

This observation introduces two domains: a **specification domain** and a **model domain.**

In this thesis we furthermore require that formal specification notations provide facilities to express abstraction. As such, the concept of abstraction is not fundamental for formal specifications, but in prac-

---

5. The underlying mathematical model is then associated with 'reality'; this association is, however, always intuitive.
tise the use of abstraction techniques turns out to be almost always possible. With respect to the model domain, this means that instead of only one model that can be regarded as the meaning of a formal specification, a collection (set) of models can be associated with a formal specification as its meaning; the fact that a specification is abstract (with respect to a specification further along the line in the software process) means that some of the elements that the specification aims to describe are still undetermined, and that the development of that specification, when used as a source specification, could result in different target specifications, depending on yet unknown factors. Therefore, a number of models can be regarded as the meaning of that specification, and each of these models satisfies (i.e. is an interpretation of) that specification$^6$ (figure 2.4). To ensure that the models in the set are actually different from one another with respect to the relevant aspects as expressed in the specification, a canonical notation must be used to represent a model.

The following definition of a formal specification language can now be given.

**Definition 2-1**

A formal specification language is a language with a well-defined syntax and semantics; the latter can be regarded as a set of models each of which is a valid interpretation of a specification expressed in the language.

The syntax of a formal specification language is usually defined in an by a combination of a context-free grammar and a static semantics. The latter removes all those sentences from the language generated by the grammar which are not well-formed. The symbols of the language can either be textual or graphical, but usually they are textual.

---

6. Note that the fact that a number of models can be associated with a specification instead of just one does not mean that the specification is ambiguous. In the case of a formal specification there should be no confusion about which models are in the set of models. In the case of an ambiguous specification one cannot be sure whether a model is in the set or not.
2.2 Formal methods

In chapter 1 several descriptions of what a formal method is were given. In the context of this research, however, it is important to have a definition in which formal methods are linked to the transformational model as introduced at the beginning of this chapter.

The word ‘method’ in formal method implies that a formal method includes techniques to apply its mathematical basis, and guidelines to do so systematically, along with a set of general principles. With respect to the transformational model for software development this means that the formal method should address one or more elementary transformation steps of the total transformation process. The rigor with which this is done may differ, but in order to reason about the correctness of a single elementary transformation step it is necessary that at least the relationship that exists between two consecutive specifications is defined; usually this relationship is called a ‘refinement’ relationship. It is not necessary that a formal method uses a single specification language in which specifications are expressed; in fact several specifi-
cation languages may be used for specifications at different stages of development.

We can now define a formal method as follows.

**Definition 2.2**

A *formal method* is a (collection of) formal specification language(s), and defines formally the relationships that exist between formal source and target specifications in one or more elementary transformation steps.

---

**2.3 Classifying formal methods**

To be able to select a formal method from the formal methods that currently exist, it is useful to have a classification of formal methods, in this way making a better motivated choice possible. Formal methods are usually arranged in what in this thesis is called the semantic foundation-based classification.

**2.3.1 The semantic foundation-based classification**

The most commonly accepted classification of formal methods is based on the semantic foundation of the formal specification languages that they use. The two main semantic approaches that can be distinguished are the *model-oriented approach* and the *property-oriented approach*; each of them has served as a basis for a broad class of formal methods. A comparison of these two approaches can be found in [Duce 87].

**Model-oriented formal methods**

In the model-oriented approach, the specification language provides constructs which enable the user to specify a *model* of the system's behaviour in terms of a mathematical model using abstract data structures such as sets, sequences, cartesian products, maps and functions. The intended behaviour of the system can be understood by inferring
behaviour from the model. The main advantage of this approach is that such models are natural for humans to understand.

Model-oriented formal methods can be further categorized as *process algebra methods* (e.g. *CCS* [Milner 80, Walker 87, Milner 89], *CSP* [Hoare 85]), *state machine methods* (e.g. *ASLAN* [Auernheimer 86], *PAISLey* [Zave 81, Zave 82], *Petri nets* [Peterson 77], and *Statecharts* [Harel 84, Harel 87, Harel 92]), or *set theoretic methods* (e.g. *COLD*, *RAISE* [RAISE 92, George 92], *VDM* (see section 2.4), and *Z* [Spivey 88]).

**Property-oriented formal methods**

Property-oriented formal methods use formal specification languages which enable the expression of the necessary minimal constraints of the system's behaviour, without prescribing an internal structure or a model of the system. This means that no specific architecture of the system is implied by the formal specification (no bias towards a particular implementation), giving the designer some freedom in the further development of the system. A disadvantage of property-oriented formal methods is that they seem to be difficult to understand for many humans.

Property-oriented formal methods fall into two different subcategories: *axiomatic methods* and *algebraic methods*. Axiomatic methods are based on first-order predicate logic, which is used to state pre- and post-conditions of operations over abstract data types. Examples of such methods are *Larch* [Guttag 90], *ANNA* [Luckham 85, Luckham 90], and *EHDM* [Rushby 91].

Algebraic methods are based on multi-sorted algebras, where properties of the behaviour of the specified system are related to equations over the entities of the algebra. Methods in this group based on equational logic are for example *Clear* [Burstall 81a, Burstall 81b], *ASF* [Bergstra 87] and *ACT ONE* [Ehrig 83]. Property-oriented methods based on temporal logic include *Temporal Logic* [Pnueli 86], and *ProCos* [Björner 91]. Property-oriented methods based on higher-order logic include *EVES* [Craigen 91, Saaltink 91], *HOL* [Gordon 87], and *OBJ3* [Goguen 88]. *LOTOS* [Bolognesi 87, ISO8807] is a well-known...
hybrid method, which is based on a combination of different semantic concepts.

2.3.2 The application-based classification

The classification presented in section 2.3.1 is based on the differences in semantic foundations between different formal methods. In reality, however, the motivation of choosing a particular formal method is often not based on its semantic foundation, but on how and for what it can be used, or even which aspects of that formal method are useful in a particular situation. Therefore, a classification of formal methods is needed which is not based on their semantic foundation, but on their practical use. This classification will in chapter 4 be used to analyze the integration of different categories of formal methods in a systematic way.

To determine the usability of a (formal) method in the software process, the demands that software processes put upon the methods applied in that process must be identified. These are:

- the method must provide support for at least one of the activities (i.e. either transformation, one-dimensional verification, or two-dimensional verification) in an elementary transformation step;
- the product(s) of the application of the method for an activity (the target specification) must be usable as a starting point (the source specification) for an activity directly following that activity.
- the product(s) of the application of the method for an activity must be understandable for those who must use the (se) product(s), for example if a formal specification is delivered as a software requirements document for a customer not familiar with formal specifications, then this is clearly not acceptable.
- the method must be supported by appropriate tool support. The complexity of software development is such that it cannot be done in an efficient way without the aid of automated support.

The application-based classification of formal methods distinguishes between formal methods by considering:
1. their support for the elementary transformation steps of the software process as discussed in section 2.1.1;
2. differentiating on the application style of a formal method, making a distinction between methods that require complete formality and methods that allow a 'rigorous' approach, i.e. methods containing both formal and informal elements.

Six different categories of formal methods can be distinguished in this way.

**Formal specification methods**

*Formal specification methods* (FSMs) are used to describe a source or target specification so that it is possible to reason formally about (aspects of) that source or target specification.

**Rigorous specification methods**

A rigorous specification method (RSM) provides a mathematical notation for annotating (parts of) a source or target specification. Such annotations consist of fragments of formal specifications, describing isolated (or incomplete) aspects of a system, so that no formal verification of these aspects is possible.

**Formal verification methods**

*Formal verification methods* (FVMs) are used for carrying out the verification activity in a formal way. Given a source specification, and a target specification derived from this source specification, it must be verified that the target specification is a correct transformation of the source specification. A formal verification method provides proof obligations. These proof obligations make explicit what has to be verified as part of the relationship between the source and target specification when a development step has been performed.

**Rigorous verification methods**

A rigorous verification method (RVM) provides the possibility to formally verify the result of a transformation step, but does not require
that every aspect of the system is formally verified. Instead, such a method allows that it is (not formally) argued or tested that a performed transition is correct. Rigorous verification can be regarded as a compromise between total non-formality (i.e. testing) and total formality (i.e. carrying out proofs), where only crucial or very complex issues are completely formally proven, and the more trivial aspects are rigorously dealt with.

**Formal development methods**

A *formal development method* (FDM) provides formal support directly for a transformation step. It provides a calculus which can be used to transform a source specification into a target specification. Theoretically such a calculus could be automated, and then the verification activity would be unnecessary.

Currently, formal development methods only exist in a very limited form, offering specialized support for aspects of the wide range of development activities. Examples are the Bird-Meertens formalism [Meertens 86], CIP [Bauer 89], CCS [Walker 87, Milner 89] and TK [Henson 88]. These formal development methods are better known as *constructive* or *transformational* methods. The calculi associated with these methods can be used to transform one formal specification into another. An overview of automated support for constructive methods is given in [Partsch 83].

**Rigorous development methods**

Whereas a formal development method provides formal support for performing the transformation activity, a *rigorous development method* (RDM) provides *non-formal* support for transforming a source specification into a target specification by giving guidelines for how such a transformation should be done. Usually such guidelines are very 'loose', i.e. the developer has a certain degree of freedom in applying them. Since the development has become non-formal again, formal verification is an essential ingredient of rigorous development methods.

The described categories of formal methods partially overlap; the rela-
tionships between them are shown in figure 2.5. The figure shows that a formal method can be a member of more than one category. Therefore, it is for example meaningful to speak of the use of a formal method as a rigorous verification method, when simultaneously that formal method can be used as a formal specification method. It is useful to make these distinctions, because the modes of application of formal methods can be different from situation to situation. In many cases it is sufficient if a formal method is used as a formal specification method, e.g. when the major difficulties lie in comprehending the exact functionality of a system. In other cases, e.g. when safety aspects play a dominant role, it may be necessary to choose a representative from the category of formal verification methods.

Figure 2.5: A classification of formal methods

2.4 The Vienna Development Method

Throughout this thesis one specific formal method is used: the Vienna Development Method. In this section we will give a summary of its characteristics, for an extensive overview of the use of VDM and its formal specification language VDM-SL we refer to text books such as [Bjørner 82], [Jones 90a], [Jones 90b], [Andrews 91] and [Dawes 91].

2.4.1 History

‘VDM’ is a generic term because several VDM ‘dialects’ exist. The development of VDM started back in 1970, when in the IBM laboratory in
Vienna a group formed by Heinz Zemanek worked on formal language definition and compiler design. They built on ideas of Elgot, Landin and McCarthy to create an operational semantics approach capable of defining the whole of PL/I, including the parallel features of the language. For this purpose they used a meta-language which was called *Vienna Definition Language (VDL)*. The approach taken was successful, but it also showed that operational semantics could complicate formal reasoning in an unnecessary way. A new approach was taken, called *denotational semantics*, in late 1972. A PL/I compiler was designed using a meta-language called *Meta-IV*. Due to external reasons the compiler was never finished, but the formal definition of PL/I in a denotational style is generally seen as the birth of VDM.

The diversion of the IBM group to handle more practical problems led to its dissolution. From then on, further development mainly took place at two locations: in Lyngby, Denmark (Prof. Dines Bjørner) and in Manchester, UK (Prof. Cliff B. Jones). The Danish VDM research has concentrated on systems software specifications, which has led inter alia to a complete formal definition of the Ada language, whereas the English research has mainly concentrated on algorithm and data structure refinement. VDM's proof obligations are one of the major results of the latter research. Thus, the main reason for the existence of different VDM dialects is the different areas of application VDM can be used for. Unfortunately, such a diversion does not stimulate industrial acceptance of VDM. Therefore, in 1986 work started on the establishment of a standard version of VDM's formal specification language: VDM-SL.

The standardization effort was initiated by the *British Standards Institution (BSI)*, establishing a Panel (BSI IST/5/-50 until 1990, since then BSI IST/5/-19) whose membership was also open to members from foreign organizations. In 1990 the need for a VDM-SL standard was also recognized by *ISO/IEC JTC1* by accepting a new work item. A Working Group (WG19) was formed by SC22. The author of this thesis is an active member of SC22/WG19, relevant publications are [Plat 92a], [Plat 92b] and [Larsen 92].

2.4.2 VDM-SL

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 types, 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 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* constructs (expressions) to specify functions, and *imperative* constructs (statements) to specify operations. Operations have direct access to a collection of global objects: the *state* of the specification. The state is specified as a composite type consisting of labeled components.

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

The VDM development paradigm (figure 2.6) matches the transformational model for software development as introduced in section 2.1. An

Figure 2.6: VDM development paradigm

initial VDM specification should be as abstract as possible. Two techniques are available for the development of an initial specification: data reification, which addresses lowering the level of abstraction of data types, and operation modeling, which addresses the development of VDM functions and operations.

Data reification involves the transition from abstract to concrete data types and the justification of the transition. Choosing a more concrete data model implies a redefinition of all operations and functions on the original model in terms of the new model, a process called operation modeling. Central to data reification is the notion of adequacy, expressed through two functions on the abstract and concrete domains: the abstraction function and the retrieve function. The abstraction func-
tion maps abstract values upon concrete values, the retrieve function does the opposite, mapping concrete values onto abstract ones.

The introduction of representation details in the specification forces operations and functions to become more algorithmic (and hence more concrete). The final step within the development is the transition of a low-level specification into the programming language of choice. This process is supported by the operation decomposition rules for VDM-SL constructs. These rules render the possibility of a constructive mapping of VDM-SL constructs into programming language constructs.

2.5 Motivation for the use of VDM

The formal method VDM was chosen as a 'representative' formal method in this thesis for the following reasons:

- VDM is model-oriented. Model-oriented formal methods are easier to understand for human beings than property-oriented formal methods, an opinion shared by e.g. Bloomfield and Froome [Bloomfield 86] and McParland [McParland 89]. Furthermore, although algebraic specification techniques have been widely applied to small examples, there is yet little evidence that they are suitable for specifying and developing large systems.

- VDM-SL is a wide-spectrum language. The language does not only support highly abstract specifications, but very concrete specifications as well. This makes it possible to choose an appropriate level of abstraction for a particular purpose.

- VDM is a representative of the large class of formal verification methods. The class of formal verification methods is the largest class that can currently effectively be used in practical software development. Although formal development methods are potentially more powerful, they are not yet mature enough to be used in industrial applications. No calculus can be associated with VDM (although a first attempt in this direction is described in [Lewington 90]) and, therefore, no use of VDM as a formal development method is as yet possible.
• VDM is an 'established' method with extensive documentation. Having been under development since the early sixties, having been used for many projects, and currently being standardized, it is safe to say that VDM is a mature formal method. Furthermore, the survey by Austin et al. [Austin 93] showed that VDM (together with Z) belongs to the most widely used formal methods.

• VDM can be used for describing different system aspects separately. Although primarily intended for describing functional aspects, VDM can also be used for describing e.g. temporal aspects of systems, by providing explicit models for such aspects [Brink 93].

• VDM is alive and kicking. The four VDM symposia held in the five past years [VDM'87, VDM'88, VDM'90, VDM'91], and the continued effort to stimulate the use of VDM in industrial companies by the EC sponsored society Formal Methods Europe, support this claim.

• A personal preference. The author likes VDM and is used to it. As a member of BSI IST/5/-/19 and ISO/IEC JTC1/SC22/WG19 he has made contributions to the international standard for VDM-SL. Although this observation can hardly be qualified as a scientific reason for choosing VDM, it is an important pragmatic reason, considering the limited time scale within which the research project had to be carried out.

Having justified the choice for VDM, the illusion should be avoided that VDM is the ideal formal method to use in all situations. When deciding to use a formal method, a careful analysis must be made as to what the formal method is going to be used for and how it is going to be used. Based on that analysis, an appropriate formal method should be selected.
In this chapter we will discuss technology transfer aspects of the ISO/VDM-SL standard, with emphasis on what makes the standard in its current shape useful (or not) for industrial organizations. The chapter is organized as follows. First, in section 3.1 an overview of the standard will be presented. The standard consists of a number of components, each of which is introduced. Our attention has been focussed on three of these components:

1. The static semantics of VDM-SL. The static semantics define the well-formedness criteria for a VDM-SL specification. Our original work in this area has not directly been included in the standard, but it turns out that it can be used in an easy manner for the construction of an efficient checker of the well-formedness for a VDM specification.

2. The syntax mapping from the Outer Abstract Syntax (OAS) to the Core Abstract Syntax (CAS). The standard comprises two abstract syntaxes for VDM specifications between which a constructive mapping has been defined.

3. The conformance clause. A conformance clause is a common component of a programming language standard, but its use and
definition must be revised for a specification language.

We report on these three components in more detail in sections 3.2, 3.3, and 3.4, respectively. In section 3.5 we summarize our conclusions with respect to the technology transfer aspects of these three components.

The material presented in this chapter has also been published as references [Plat 92a], [Plat 90], [Plat 92b] and [Larsen 92].

3.1 Overview of the VDM-SL standard

In this section we will present an overview of the standard for VDM-SL. A reference guide for the language following the standard (to be) as closely as possible is reference [Dawes 91]. Another recent text book illustrating the language is reference [Andrews 91]. At the start of the standardization process it was agreed that the standard should as much as possible be based on existing work. Two main VDM books ([Björner 78] and [Björner 82]) were used as baseline documents. The static semantics of the STC/VDM-RL dialect [Monahan 85] was used as the starting point for the semantics of VDM-SL. Another decision that was taken right from the beginning was that the standard itself should as much as possible be formally defined, in order to make it as precise as possible.

The complete definition of the standard for VDM-SL can be divided into a number of major components: a syntax (defined at several levels of abstraction), symbol representations, a static semantics, a dynamic semantics, and a syntax mapping. The relation between these components is shown in figure 3.1.

3.1.1 The syntax

The main component of the ‘user interface’ of VDM-SL is formed by its syntax, which exists in two forms:

1. A concrete form in terms of EBNF rewriting rules. The EBNF (Extended Backus Naur Form) formalism is nowadays the most
accepted form for defining the concrete syntax of a formal language, in particular because of the possibilities for automatically generating parsers from such a definition.

2. An abstract form in terms of VDM-SL type definitions. This form is called the Outer Abstract Syntax (OAS). The OAS was introduced because it can be used in parts of the standard which are defined in terms of VDM-SL functions. This is clearly not possible for the EBNF form, because EBNF is an entirely different formalism. The OAS is used both by the static semantics $SS$ (section 3.1.6) and by the syntax mapping functions $SM$ (section 3.1.4).

As an example, consider the syntax for an if-then-else expression. Using the EBNF formalism, the concrete syntax is

8:

\[
\text{if expression} = \text{'if', expression, 'then', expression, } \\
\{ \text{elseif expression } \},
\]

8. The example we use to illustrate the standard is simple. Still, some details require substantial extra explanation which lies beyond the scope of this section.
'else', expression ;
elseif expression = 'elseif', expression,
    'then', expression ;

In terms of VDM-SL type definitions, the syntax for the if-then-else expression is:

\[
\begin{align*}
    \text{If} & \colon \text{test} : \text{Expr} \\
    & \colon \text{cons} : \text{Expr} \\
    & \colon \text{altns} : \text{ElsifExpr}^{*} \\
    & \colon \text{altn} : \text{Expr} \\
    \text{ElsifExpr} & \colon \text{test} : \text{Expr} \\
    & \colon \text{cons} : \text{Expr}
\end{align*}
\]

The syntax is the starting point which is used for the definition of all other relevant aspects of VDM-SL in the standard.

3.1.2 Symbol representations

The required representation of the symbols used in a specification is also defined in the standard. Currently, two different representations exist:

1. A mathematical representation. The mathematical representation provides elegant symbols, clearly distinguishing between keywords, reserved words, and resembling generally accepted mathematical notation as much as possible.
2. An ASCII representation. An ASCII representation has been defined to make automatic processing of VDM-SL specifications possible.

The relation between the syntax and the symbol representations is defined by a lexis \( \mathcal{L} \).

As an example of the difference in appearance between the two representations, consider an operation deleting a 'maplet' from some state component. Using the mathematical representation, such an operation could look like:

\[
\text{Delete} \ d : \mathbb{N} \\
\text{ext wr collection} : \mathbb{N} \rightarrow \mathbb{B}
\]
pre \ d \in \text{dom} \ collection

post \ collection = \text{collection} \triangleleft \{d\}

The ASCII representation of the same operation is:

Delete \ (d: \text{nat})
\text{ext wr collection: map nat to bool}
pre \ d \in \text{set dom collection}
post \ collection = \text{collection}^\sim \triangleleft\ (-: \{d\})

Whereas the structure of the two representations is the same, the outward appearance clearly is not.

3.1.3 The core abstract syntax

Besides the outer abstract syntax, the standard also contains another abstract syntax representation for VDM specifications which is less complicated than the OAS: the Core Abstract Syntax (CAS). The CAS is used for the definition of the formal semantics of the language (the dynamic semantics $\mathcal{DS}$, section 3.1.5), and it was introduced because the formulae for the formal semantics are less complicated and thus easier to understand when defined over the CAS than when defined over the OAS.

Consider e.g. the equivalent of the OAS if-then-else expression in the CAS:

\[
\text{If \ test : Expr} \\
\text{cons : Expr} \\
\text{alt n : Expr}
\]

The nested structure of elseif expressions is no longer present in the CAS.

3.1.4 The syntax mapping

The gap between the concrete representation of a VDM-SL specification (OAS) and the representation over which the meaning of a specifica-
tion has been defined (CAS) is large, and the transformation between the two is far from trivial. Therefore, the standard contains a formal definition of a syntax mapping $SM$ from a specification in terms of the OAS to a specification in terms of the CAS.

As an example, consider again the two constructs for if-then-else expressions in the OAS and in the CAS. The function describing the syntax mapping between the two constructs is:

$$IfExpr2IfExpr :$$

$$OAS.Expr \times OAS.Expr \times OAS.ElseIfExpr^* \times OAS.Expr \times Env \rightarrow CAS.IfExpr$$

$$IfExpr2IfExpr (test, cons, altns, altn, \rho) \triangleq$$

$$\text{let } test_i = Expr2Expr (test, \rho),$$
$$cons_i = Expr2Expr (cons, \rho),$$
$$altn_i = \text{if } altns = []$$
$$\text{then } Expr2Expr (altn, \rho)$$
$$\text{else let } mk-OAS.ElseIfExpr (atest, acons) = \text{hd } altns \text{ in }$$
$$\text{IfExpr2IfExpr (atest, acons, tl } altns, \text{ altn, } \rho \text{) in }$$
$$mk-CAS.IfExpr (test_i, cons_i, altn_i)$$

The function takes the OAS form of the if-then-else expression and an environment $\rho$, in which identifier transformations are kept, as its arguments. The else-if part which is present in the OAS but not in the CAS is dealt with by constructing a nested CAS if-then-else expression.

Apart from the fact that constructs must be simplified (i.e. transformed from the 'complicated' OAS to the more 'simple' CAS) some extensions are made as well by $SM$. An obvious example is the generation of 'quoted' post-conditions in VDM-SL: when a VDM specification contains e.g. an operation $OP$, then the specifier can automatically assume the presence of two functions $pre-OP$ and $post-OP$, modeling the pre- and post-conditions of $OP$, respectively (see also [Jones 90a]). These functions need not explicitly be specified by the user (and are thus not present in a specification in terms of an OAS), but in the dynamic semantics no assumptions are made about the presence of these functions. They are, therefore, explicitly generated by $SM$. 
3.1. Overview of the VDM-SL standard

$SM$ is discussed in more detail in section 3.3.

3.1.5 The dynamic semantics

All VDM specifications which can be represented in the core abstract syntax are given a formal meaning by the dynamic semantics $DS$. The definition of $DS$ is based on set theory and the mathematical notation which is used has been fixed. A domain universe is built which contains all valid 'values' which can be expressed in VDM-SL. On top of this domain universe a collection of semantic domains is defined. These semantic domains are used in the actual definition of $DS$.

The meaning of a VDM specification can be regarded as a set of models, because specifications can be loose. Loose specification is a technique offered by VDM-SL allowing the user to specify highly abstract components which can be implemented with different functionality. The implementation must simply have a functionality that can be said to implement the loosely specified construct according to the implementation relations which should hold for VDM-SL.

The following function gives the denotation for the if-then-else expression:

$$\text{EvalIfExpr} : \text{If} \rightarrow \text{LEEval}$$

$$\text{EvalIfExpr} (\text{MkTag} (\text{\textquote{If}}, (\text{test}, \text{cons}, \text{alt}))) \triangleq$$

$$\{ \lambda \text{env} : \text{\textquote{if testev} (\text{env}) } \notin \text{BOOL\_VAL}$$

then $\perp$

else if $\text{testev} (\text{env}) = \text{True} ()$

then $\text{consev} (\text{env})$

else $\text{altnev} (\text{env})$

| $\text{testev} \in \text{EvalExpr (test)},$

$\text{consev} \in \text{EvalExpr (cons)},$

$\text{altnev} \in \text{EvalExpr (alt)}$}

The denotation of the if-then-else expression is a function which takes an argument in terms of the CAS and returns a value in the semantic domain, in this case a value from the domain $\text{LEEval}$. The specific structure of the latter domain is not relevant for understanding what
the function does. The function starts by evaluating the test part. If this evaluation yields the boolean value true, then the denotation of the expression corresponds to the denotation of the cons part, otherwise the denotation of the altm part. If the evaluation of the test part yields a non-boolean value, then \( \perp \) (bottom) is returned as the denotation of the if-then-else expression. Thus, the dynamic semantics is total; even ‘meaningless’ specification constructs are given a meaning (meaningless expressions are denoted by \( \perp \) while an entire meaningless specification is denoted by an empty set of models).

The actual definition of \( \mathcal{DS} \) is given in a denotational way, without using the traditional style of explicitly constructing the denotation. Instead, first the set of all possible models is created, and then, by examining the syntactic specification, this set is restricted to those models that can be considered the denotation of the specification. This technique offers a relational style of denotational semantics, also used in [Hoare 85].

### 3.1.6 The static semantics

VDM specifications that are syntactically correct according to the EBNF rules do not necessarily obey the typing and scoping rules of the language. The standard, therefore, provides a formal definition of the well-formedness of a VDM specification: the static semantics \( \mathcal{SS} \) of the language. The static semantics has itself been formulated in VDM-SL. An alternative approach using inference rules has also been investigated [Damm 91].

The VDM specifications which have at least one model in the dynamic semantics can be considered as those which are well-formed. In general, it is not statically decidable whether or not a given VDM specification is well-formed. The static semantics for VDM-SL differs from the static semantics of other languages in the sense that it only rejects specifications which are definitely not well-formed, and only accepts specifications which are definitely well-formed. Thus, the static semantics for VDM-SL attaches a well-formedness grade to a VDM specification. Such a well-formedness grade indicates whether a specification is definitely well-formed, definitely not-well-formed, or possibly
well-formed. This approach creates the possibility to extent the basic rejection and basic acceptance with more sophisticated checks, in this way making the area of 'possibly well-formed' specifications smaller (see figure 3.2).

![Figure 3.2: Rejected and accepted subsets of specifications](image)

With respects to the static semantics, the example of the if-then-else expression requires that the test expression has a boolean type. In addition, it is required that the type of the consequence expression and the type of the alternative expression conforms to the expected type of the entire if-then-else expression in certain ways. To declare the if-then-else expression to be definitely well-formed, both types must be subtypes of the expected type of the if-then-else expression. On the other hand, the if-then-else expression cannot be rejected if either the type of the consequence expression or the type of the alternative expression(s) is a subtype of the entire if-then-else expression. In this way each specification construct is given a well-formedness grade. A detailed description of the static semantics and the rationale behind it can be found in [Bruun 91].

### 3.2 Static semantics for VDM-SL

Having briefly introduced the standard, we will now first turn to the subject of defining a static semantics for VDM-SL; more in particular, we shall focus on the type system which is at the basis of a static semantics. Our approach has been applied for the construction of a type checker for VDM-SL, which can be regarded as an extension of the visibility analyzer constructed by the author of this thesis as part of his Master's Thesis project; the reader is referred to [Plat 88] for a description of this Master's Thesis project and the tool. At the time of this
research the BSI Panel had not yet decided upon the approach to be followed for the construction of the static semantics for VDM-SL, but after consideration the approach as developed at the Technical University of Denmark was selected [Bruun 91], which allows for a more detailed answer to the question whether or not a VDM-SL specification is well-formed than our approach. The drawback of the Danish approach is that it is much more complicated than our approach, which makes their static semantics more difficult to understand and more difficult to implement (our type system can be relatively easy and efficiently be implemented by using an attribute grammar evaluating system, see [Plat 90]) and also that the usefulness of being able to distinguish between different granularity levels for well-formedness checks has not been proven in practice.

3.2.1 The type system

The notion of 'type' in VDM-SL is used to place semantic constraints upon the formation of expression terms. These constraints are formulated in terms of a so-called type system. A type system for a language is a set of rules associating a type with expressions in that language [Cardelli 85, Sethi 89]. For the definition of our type system we will assume that expressions are represented as abstract syntax trees (in terms of the VDM-SL standard: Outer Abstract Syntax trees); we will call such trees expression trees. We will assume that type information for identifiers is available at every node in the expression tree. Our type system then consists of a description of a single bottom-up tree walk of the expression tree during which at each node \( n \) two different actions are carried out:

1. **Type propagation.**
   We define type propagation as the construction of a set of types, based on node \( n \) itself and (in general) sets of types associated with the subtrees of \( n \). We call such a set of types the possible types set of \( n \). The possible types set of \( n \) is itself used for the type propagation performed at the parent-node of \( n \).

2. **Type constraint checking.**
   We define type constraint checking as checking whether the possible types sets of the subtrees satisfy a node \( n \) dependent
relation. We will not consider constraints with regard to type invariants: type invariants are constraints that the writer of the specification is expected to verify. If the type constraints of a node are not met, then an error message is elaborated.

Such a tree walk as a whole constitutes the process that we call type checking. The result of the type propagation at a node at level \( k \) in the tree is used for the type constraint checking at a node at level \( k-1 \) (i.e. one level upwards) in the tree. Therefore, we discuss type propagation first in the next subsection and type constraint checking in the subsection following.

### 3.2.2 Type propagation

We want to determine the type of each node of the expression tree, so that this information can be propagated to the parent-node of that node. Since some language constructs are overloaded, and since the language allows the use of union types, problems occur. Consider e.g. the (very simple) expression:

\[
a
\]

where the identifier \( a \) is of type \( N | B \). We do not know what the type of the expression \( a \) is; all we know is that it is either of type \( N \) or of type \( B \).

A central idea behind our approach is that type constraint checking is useful, even when having incomplete or ambiguous type information at a node in the expression tree, provided that sufficient type information is available to filter out the erroneous cases.

Making type information available at a node in the expression tree is the main purpose of the type propagation activity. Since, as was illustrated, it is not always possible to uniquely determine the type of a node in the expression tree, we do not associate a single type with that node, but a set of types: the so-called possible types set. The

---

9. Other constraints might be present as well, which, however, are not related to the possible types set of a subtree. Such a constraint is e.g. that a sequence to which the \( \text{hd} \) operator is applied, should not be empty. Non-type related constraints are not considered in this section.
possible types set denotes a set of types of which each member is a candidate type for the node under consideration. We will use the term \textit{type propagation} to denote the process of establishing the possible types set of an expression.

Occasionally it may not be possible to establish a possible types set of an expression at all, e.g. the type of an undefined identifier. In this case the value of the possible types set is a set containing the special value \textit{unknown}. The value \textit{unknown} is used to denote the whole set of types in the type universe.

The algorithm for a function that computes the possible types set of an expression can be expressed in a VDM-SL-like notation. This typically involves a recursive function that decomposes the expression in its subexpressions, and then computes the possible types set in terms of the possible types set of these subexpressions. Such a function must be provided for every kind of expression in the language. Suppose we have defined such a function \textit{PTO} (Possible Types Of). Then the signature of this function is:

\[ PTO : \textit{Expr} \rightarrow \textit{Type-set} \]

where \textit{Expr} corresponds to the construct with the same name in the abstract syntax of VDM-SL.

For the definition of \textit{PTO} we need a few auxiliary functions. These functions are used to formalize aspects of type propagation in our type system. The purpose of these definitions will be demonstrated in the examples of the definition of \textit{PTO}. In the first definition the notion of a type \textit{enclosing} another type is formalized, thus defining an ordering over the type universe.

\textbf{Definition 3-1}
Consider the function \textit{Encloses} with the following signature:

\[ \textit{Encloses} : \textit{Type} \times \textit{Type} \rightarrow \mathbb{B} \]

where \textit{Type} corresponds to the construct with the same name in the abstract syntax of VDM-SL. The definition of \textit{Encloses} corresponds to an intuitive notion of an ordering over types, such that type \( t_1 \) being
'larger' than another type \( t_2 \) is a meaningful expression.

\[ \text{Encloses : Type} \times \text{Type} \rightarrow \mathbb{B} \]

\[ \text{Encloses}(t_1, t_2) \triangleq \]

\begin{cases} 
\text{mk-}(t_1, t_2) : \\
\text{mk-}(\text{mk-BasicType}(bt_1), \text{mk-BasicType}(bt_2)) \rightarrow bt_2 \subseteq bt_1, \\
\text{mk-}(\text{mk-QuoteType}(lt_1), \text{mk-QuoteType}(lt_2)) \rightarrow lt_1 = lt_2, \\
\text{mk-}(\text{mk-UnionType}(sett_1), t_2) \rightarrow \\
\quad \text{if is-UnionType}(t_2) \\
\quad \text{then let mk-UnionType}(sett_2) = t_2 \text{ in} \\
\quad \quad \forall x \in \text{sett}_2 : \\
\quad \quad \quad \exists t \in \text{sett}_1 \cdot \text{Encloses}(t, x) \\
\quad \text{else} \\
\quad \quad \exists t \in \text{sett}_1 \cdot \text{Encloses}(t, t_2), \\
\text{mk-}(t_1, \text{mk-UnionType}(sett_2)) \rightarrow \forall t \in \text{sett}_2 \cdot \text{Encloses}(t_1, t), \\
\text{mk-}(\text{mk-ProductType}(seqt_1), \text{mk-ProductType}(seqt_2)) \rightarrow \\
\quad (\forall i \in (\text{inds seqt}_1 \cap \text{inds seqt}_2) : \\
\quad \quad \text{Encloses}(seqt_1(i), seqt_2(i))) \land \\
\quad \text{len seqt}_1 = \text{len seqt}_2, \\
\text{mk-}(\text{mk-OptionalType}(ot_1), t_2) \rightarrow \\
\quad \text{if is-OptionalType}(t_2) \\
\quad \text{then let mk-OptionalType}(ot_2) = t_2 \text{ in} \\
\quad \quad \text{Encloses}(ot_1, ot_2) \text{ else Encloses}(ot_1, t_2), \\
\text{mk-}(\text{mk-SetType}(ct_1), \text{mk-SetType}(ct_2)) \rightarrow \text{Encloses}(ct_1, ct_2), \\
\text{mk-}(\text{mk-SeqType}(ct_1), \text{mk-SeqType}(ct_2)) \rightarrow \text{Encloses}(ct_1, ct_2), \\
\text{mk-}(\text{mk-MapType}(dt_1, rt_1), \text{mk-MapType}(dt_2, rt_2)) \rightarrow \\
\quad \text{Encloses}(dt_1, dt_2) \land \text{Encloses}(rt_1, rt_2), \\
\text{mk-}(\text{mk-FnType}(dseqt_1, rt_1), \text{mk-FnType}(dseqt_2, rt_2)) \rightarrow \\
\quad (\forall i \in (\text{inds dseqt}_1 \cap \text{inds dseqt}_2) : \\
\quad \quad \text{Encloses}(dseqt_1(i), dseqt_2(i))) \land \\
\quad \text{len dseqt}_1 = \text{len dseqt}_2 \land \text{Encloses}(rt_1, rt_2), \\
\text{mk-}(\text{mk-TypeName}(\text{mk-Name}(\text{pre, loc})), t_2) \rightarrow \\
\quad \text{Encloses}((\text{LookUp}(\text{pre, loc}), t_2), \\
\text{mk-}(t_1, \text{mk-TypeName}(\text{mk-Name}(\text{pre, loc})))) \rightarrow \\
\quad \text{Encloses}(t_1, (\text{LookUp}(\text{pre, loc})), \\
\text{mk-}(\text{mk-TypeVar}(id_1), \text{mk-TypeVar}(id_2)) \rightarrow id_1 = id_2, \\
\text{others} \rightarrow t_1 = t_2 \end{cases} \]
Definition 3-2
We treat a union type as if it were a type set of which the elements are the member types of the union type. Therefore, we need a function that can convert a set of types of which the members that are union types, are flattened. We define a function $\text{Flatten}$ as follows:

$$\text{Flatten} : \text{Type-set} \rightarrow \text{Type-set}$$

$$\text{Flatten}(ts) \triangleq \bigcup \{\text{if is-UnionType}(x) \text{ then let } mk\text{-UnionType}(sett) = x \text{ in } Flatten(sett) \text{ else } \{x\} \mid x \in ts\}$$

Definition 3-3
The function $\text{SubTypesOf}$ is defined as follows:

$$\text{SubTypesOf} : \text{Type-set} \rightarrow \text{Type-set}$$

$$\text{SubTypesOf}(ts) \triangleq Flatten(\{x \mid x : \text{Type} \cdot \exists t \in ts \cdot Encloses(t, x)\})$$

Definitions 3-1 up to 3-3 are used to define a function $\text{MET}$ (Minimal Enclosing Types), which is used to calculate the possible types set of an expression.

Definition 3-4
The function $\text{MET}$ is defined as follows:
\[ MET : Type\text{-}set\text{-}set \rightarrow Type\text{-}set \]

\[ MET(tss) \triangleq SubTypesOf(\{ x \mid x : Type \cdot \\
(\forall ts \in tss \cdot \exists t \in Flatten(ts) \cdot Encloses(x, t)) \land \\
(\exists ts \in tss \cdot x \in Flatten(ts))\}) \]

\[ box \]

\[ MET \] can best be illustrated by giving a few examples.

**Example 3-1**

\[ MET(\{\{N\}, \{Z\}, \{N, B\text{-}set\}\}) = SubTypesOf(\{Z\}) = \{N_1, N, Z\} \]

\[ MET(\{\{(N \mid B)\text{-}set\}, \{N_1\text{-}set\}\}) = SubTypesOf(\{(N \mid B)\text{-}set\}) = \\
\{(N_1 \mid B)\text{-}set, (N \mid B)\text{-}set, N_1\text{-}set, N\text{-}set, B\text{-}set\} \]

\[ MET(\{\{N\}, \{B\}\}) = SubTypesOf(\{\}) = \{\} \]

\[ MET(\{\{N_1\}, \{unknown\}\}) = SubTypesOf(\{R\}) = \{N_1, N, Z, Q, R\} \]

\[ MET(\{\{Z \mid B\}\}) = SubTypesOf(\{Z, B\}) = \{N_1, N, Z, B\} \]

\[ box \]

So, \[ MET \] is a function that first reduces a set of type sets so that only types remain which are enclosing types of at least one of the types in every flattened component type set, and then expands this set with all the subtypes of the element types. The notion of minimal enclosing type can be used as a mechanism for eliminating those types from a set of types that can no longer be regarded as significant for the computation of the possible types set for a node \( n \).

A final observation regarding \( PTO \) is that we would like this function never to return an empty set as its result. An empty set as the result of applying \( PTO \) to a node \( n \) indicates that it was not possible to determine a possible types set for that node. This would, however, seriously influence the type propagation and type constraint checking
at the parent-nodes of \( n \). We avoid this by doing type repair in such situations. The general policy is to expand the possible types set as much as possible with valid types for that node \( n \), so that the value of the possible types set of \( n \) will influence the type propagation at the parent-nodes of \( n \) as little as possible.

Now we can give a few examples of the function \( PTO \) as it is defined in our type system.

**Example 3-2**
The possible types set of a binary addition expression is determined by the possible types sets of the operands and by the fact that it can only contain numeric types. A suitable definition of \( PTO \) for this kind of expression is:

\[
PTO : \text{Expr} \rightarrow \text{Type-set}
\]

\[
PTO(\text{mk-BinaryExpr}(\text{lhs}, \text{NUMPLUS}, \text{rhs})) \triangleq
\]

\[
\text{let } \text{PossibleTypes} = \text{MET}(\{PTO(\text{lhs}), PTO(\text{rhs}), \{N_1\}\}) \text{ in}
\]

\[
\text{if } \text{PossibleTypes} = \{\}
\]

\[
\text{then } \{N_1, N, Z, Q, \mathbb{R}\}
\]

\[
\text{else } \text{PossibleTypes}
\]

So the possible types set of the expression \( a + b \) where identifier \( a \) is of type \( \mathbb{Z} \mid \mathbb{B} \) and identifier \( b \) is of type \( N \mid \mathbb{B} \) would be: \( \{N, N_1, Z\} \). \( \Box \)

**Example 3-3**
The possible types set of a map composition\(^{10}\) expression is given by:

\[10. (m_1 \circ m_2)(x) \text{ is defined as } m_1(m_2(x)).\]
3.2. Static semantics for VDM-SL

\[ PTO : \text{Expr} \rightarrow \text{Type-set} \]

\[ PTO (\text{mk-BinaryExpr}(m_1, \text{MAPCOMPOSE}, m_2)) \triangleq \]

let \text{PossibleTypes} =
\begin{align*}
& \{ \text{mk-MapType}(d_1, d_2) \mid d_1, d_2 : \text{Type} \cdot \\
& d_1 \in \{ x.\text{dom} \mid x \in PTO(m_2) \cdot \text{is-MapType}(x) \} \wedge \\
& d_2 \in \{ x.\text{rng} \mid x \in PTO(m_1) \cdot \text{is-MapType}(x) \} \wedge \\
& x.\text{dom} \in \\
& \text{MET}(\{ y.\text{dom} \mid y \in PTO(m_1) \cdot \text{is-MapType}(y) \}, \\
& \{ z.\text{rng} \mid z \in PTO(m_2) \cdot \text{is-MapType}(z) \}) \} \}\text{ in} \\
& \text{if PossibleTypes} = \{ \} \\
& \text{then} \{ \text{mk-MapType}(d, r) \mid d \in \{ \text{unknown} \} \wedge r \in \{ \text{unknown} \} \} \\
& \text{else} \text{PossibleTypes}
\end{align*}

So, the possible types set of the expression \( m_1 \circ m_2 \) where \( m_1 \) is of type \( B \xrightarrow{m} B \) and \( m_2 \) is of type \( Z \xrightarrow{m} N \mid B \) would be: \( \{ N_1 \xrightarrow{m} B, N \xrightarrow{m} B, Z \xrightarrow{m} B \} \).

This example clearly shows that type propagation and type constraint checking are separated. Even if the type constraints for a correct application of a map composition are not met (see example 3-3), then a 'reasonable' possible types set can be determined for this expression.

\[ \square \]

**Example 3-4**
The possible types set of an identifier is determined with the use of the type information kept in a symbol table.

\[ PTO : \text{Expr} \rightarrow \text{Type-set} \]

\[ PTO (\text{mk-Name}(pre, \text{loc})) \triangleq \]

\[ \text{MET}(\{ \text{LookUp}(pre, \text{loc}) \}) \]

\[ \square \]

**Example 3-5**
Determining the possible types set of a literal expression is a bit more complicated. We do not want to use \( \text{MET} \) here, because in order to be as precise as possible, we want to narrow down rather than widening
the possibilities as much as possible. We define the possible types set of a literal expression as follows:

\[ PTO : \text{Expr} \rightarrow \text{Type-set} \]

\[ PTO(\text{mk-Literal}(\text{val})) \triangleq \{ x \mid x \in \text{Type}[\text{val} \in x \land \\
- \exists t \in \text{Type} \cdot t \neq x \land \text{val} \in t \land \text{Encloses}(x, t) \} \]

Thus for the literal 5 the possible types set would be \( \{N_1\} \), and for the literal expression -7 the possible types set would be \( \{Z\} \).

The result of a single bottom-up tree walk with regard to type propagation is an expression tree, of which each node is labeled with a set of possible types for the particular subtree labeled by that node. To prove that this result is correct, we must prove that we are able to make a top-down scan of this tree during which, given a type from the set of possible types at the top node, we can reduce the possible types set at each node in the tree in such a way that each node can be labeled uniquely with exactly one type. We will not give such a proof in this section but refer to similar proofs that have been carried out for type checking Ada (where the same kind of problems occur with overloading), e.g. in [Wallis 80].

In our type system the only use of the possible types sets at the tree nodes is for the type constraint checking, which is discussed in the following subsection. The possible types set at the root of the expression tree is not used at all. This may seem strange, but should become clear when realizing that, so far, we have only been discussing expression trees. If we extend our view to 'specification trees', the possible types set of the root node of an expression tree gets a context restricting the allowable types at that node.

### 3.2.3 Type constraint checking

The second activity that is carried out at each node \( n \) of the expression tree is type constraint checking. We define type constraint checking as checking whether a node dependent type constraint holds. We limit the discussion to those constraints that involve the possible types sets
of the subtrees of the node $n$. For this class of constraint checks, two
functions, $\texttt{StaticCheck}$ and $\texttt{DynamicCheck}$, are defined for each node
$n$. Each function returns a boolean value indicating whether the type
constraints for node $n$ are satisfied or not. Their signatures are:

$$\texttt{StaticCheck : Expr} \rightarrow \mathbb{B}$$
$$\texttt{DynamicCheck : Expr} \rightarrow \mathbb{B}$$

The function $\texttt{StaticCheck}$ applied to node $n$ defines all the constraints
put upon the possible types sets of the subtrees of $n$ that can auto-
matically be verified by a type checker implementing the type system.
When $\texttt{StaticCheck}$ applied to $n$ returns the boolean value false, then
the expression for which that node is the root node is not well-formed.
A typical action for a type checker in such a situation is to elaborate
an error message. In terms of programming languages, application of
$\texttt{StaticCheck}$ can be compared with performing a compile-time check. We
say that an expression tree is well-formed if $\texttt{StaticCheck}(n)$ holds for
all nodes $n$ in the expression tree.

Other constraints might be put upon the types (instead of the possi-
ble types set) of the subtrees of $n$. Since, in general, we do not know
the type of a subtree but only the possible types set of a subtree,
these constraints cannot be verified automatically by an implementa-
tion of the type system. Fortunately, we can identify those cases where
these constraints might not hold. We use the function $\texttt{DynamicCheck}$
for this purpose. If application of the function $\texttt{DynamicCheck}$ returns
the boolean value false, then a verification condition is elaborated. A
verification condition is a proof obligation which must be satisfied by
the writer of the specification. In terms of programming languages,
application of $\texttt{DynamicCheck}$ can be compared with generating code for
performing a run-time check.

The comparison of two possible types sets for mutual compatibility
plays an important rôle in most applications of the functions $\texttt{StaticCheck}$
and $\texttt{DynamicCheck}$. The function that compares two possible types sets
with each other is called $\texttt{IsCompatible}$. The function has the following
signature:

$$\texttt{IsCompatible : Type-set} \times \texttt{Type-set} \rightarrow \mathbb{B}$$
Before we discuss possible definitions of `IsCompatible`, we will give an example of `StaticCheck` first in terms of `IsCompatible`.

**Example 3-6**
Consider a binary addition expression. For such an expression the function `StaticCheck` is defined as follows:

\[
\text{StaticCheck} : \text{Expr} \rightarrow \mathbb{B} \\
\text{StaticCheck} (\text{mk-BinaryExpr}(\text{lhs}, \text{NUMPLUS}, \text{rhs})) \triangleq \\
\text{IsCompatible} (\text{PTO}(\text{lhs}), \{N_1, N, Z, Q, R\}) \land \\
\text{IsCompatible} (\text{PTO}(\text{rhs}), \{N_1, N, Z, Q, R\})
\]

\[\square\]

The definition of `DynamicCheck` depends on the way `IsCompatible` is defined, because `DynamicCheck` is used to find all those type errors that couldn’t be found using `StaticCheck` only.

To establish a definition of the function `IsCompatible`, two different policies can be envisaged. We will call the first policy the `Strong Policy`. The central idea behind this policy is to call a type set `ts_1` compatible to a type set `ts_2`, if `ts_1` is strongly compatible to `ts_2`. We say that `ts_1` is strongly compatible to `ts_2`, if `ts_2` contains at least all elements of `ts_1`. Using this approach for the comparison of a possible types set to another type set, would mean that we consider what might be wrong in a dynamic context, is also wrong in a static context; this is also the reason for naming this policy the `strong` policy. For this policy the function `IsCompatible_s` is defined as in the following definition.

**Definition 3-5**

\[
\text{IsCompatible}_s : \text{Type-set} \times \text{Type-set} \rightarrow \mathbb{B} \\
\text{IsCompatible}_s (ts_1, ts_2) \triangleq \\
ts_1 \subseteq ts_2
\]

Note that this function defines an asymmetric relation. \[\square\]
Example 3.7
Consider again the binary addition expression. Interpreting the compatibility between type sets according to the strong policy, \( \text{StaticCheck} \) reads as follows:

\[
\text{StaticCheck} : \textit{Expr} \to \mathbb{B} \\
\text{StaticCheck} (\textit{mk-BinaryExpr}(\textit{lhs}, \text{NUMPLUS}, \text{rhs})) \triangleq \\
\text{IsCompatible$_{s}$}(\textit{PTO})(\text{lhs}), \{\text{N$_1$, N, Z, Q, R}\}) \land \\
\text{IsCompatible$_{s}$}(\textit{PTO})(\text{rhs}), \{\text{N$_1$, N, Z, Q, R}\})
\]

So, if we interpret \( \text{IsCompatible} \) according to the Strong Policy, then the expression \( a + b \) where \( a \) is of type \( \text{N} \) and \( b \) is of type \( \text{N} \mid \mathbb{B} \) would not be well-formed, because in a dynamic context the expression \( b \) could be of type \( \mathbb{B} \).

The function \( \text{DynamicCheck} \) can then be defined as follows:

\[
\text{DynamicCheck} : \textit{Expr} \to \mathbb{B} \\
\text{DynamicCheck} (\textit{mk-BinaryExpr}(\textit{lhs}, \text{NUMPLUS}, \text{rhs})) \triangleq \\
\text{true}
\]

The definition of \( \text{StaticCheck} \) guarantees that all type constraints that might be wrong in a dynamic context are found, thus nothing remains to be done by \( \text{DynamicCheck} \).

When using the Strong Policy, no further dynamic type constraints need to be put upon the possible types set of a node \( n \) (apart from e.g. checking type invariants, but that is not considered in our type system).

The second policy we discuss for type constraint checking, is the so-called Weak Policy. The central idea is to call a type set \( ts_1 \) compatible to a type set \( ts_2 \), if \( ts_1 \) is weakly compatible to \( ts_2 \). We say that \( ts_1 \) is weakly compatible to \( ts_2 \), if \( ts_1 \) contains at least one type that is also in \( ts_2 \). Using this approach for the comparison of a possible types set to another type set, would mean that we consider what might be right in a dynamic context, is also right in a static context; this is the reason for naming this policy the \textit{weak} policy. For this policy the function \( \text{IsCompatible}_w \) is defined as follows.
Definition 3-6

\[ \text{IsCompatible}_w : \text{Type-set} \times \text{Type-set} \rightarrow \mathbb{B} \]

\[ \text{IsCompatible}_w(t_{s1}, t_{s2}) \triangleq t_{s1} \cap t_{s2} \neq \{\} \]

\(\square\)

A side effect of this approach is that possible types sets are allowed which might contain types that are not valid types for the node \(n\) in a dynamic context. To detect such a situation, we use the function \(\text{IsCompatible}_w\) in most definitions of \(\text{DynamicCheck}\).

Example 3-8

Consider again the binary addition expression from examples 3-6 and 3-7. Interpreting the compatibility between type sets according to the Weak Policy, \(\text{StaticCheck}\) reads as follows:

\[ \text{StaticCheck : Expr} \rightarrow \mathbb{B} \]

\[ \text{StaticCheck}(\text{mk-BinaryExpr}(\text{lhs}, \text{NUMPLUS}, \text{rhs})) \triangleq \]

\[ \text{IsCompatible}_w(\text{PTO(lhs)}, \{\text{N}_1, \text{N}, \text{Z}, \text{Q}, \text{R}\}) \land \]

\[ \text{IsCompatible}_w(\text{PTO(rhs)}, \{\text{N}_1, \text{N}, \text{Z}, \text{Q}, \text{R}\}) \]

The function \(\text{DynamicCheck}\) uses the strong version of \(\text{IsCompatible}\):

\[ \text{DynamicCheck : Expr} \rightarrow \mathbb{B} \]

\[ \text{DynamicCheck}(\text{mk-BinaryExpr}(\text{lhs}, \text{NUMPLUS}, \text{rhs})) \triangleq \]

\[ \text{IsCompatible}_s(\text{PTO(lhs)}, \{\text{N}_1, \text{N}, \text{Z}, \text{Q}, \text{R}\}) \land \]

\[ \text{IsCompatible}_s(\text{PTO(rhs)}, \{\text{N}_1, \text{N}, \text{Z}, \text{Q}, \text{R}\}) \]

According to this interpretation the expression \(a + b\), where \(a\) is of type \(N\) and \(b\) is of type \(N | B\) would be well-formed (application of \(\text{StaticCheck}\)), but since \(\text{DynamicCheck}\) returns false for these parameters, a verification condition would be elaborated to verify that \(b\) does not have a value of type \(B\).  
\(\square\)
The choice between the Strong Policy and the Weak Policy is more a matter of definition than of practical importance. An implementation of a type checker following the former policy can detect the same number of ‘type conflicting’ situations as an implementation of a type checker following the latter policy. However, what is seen as a type error according to the one policy may not be an error according to the other policy; such situations are reported to the user in terms of verification conditions instead. The essential differences thus lie in the area of language definition.

For our type system we have chosen the Weak Policy, because we believe that a language should not restrict a user more than absolutely necessary in what can be expressed. The Weak Policy allows language constructs to be expressed, which in a dynamic context need not necessarily be valid.

As an additional illustration of the use of the Weak Policy, consider the following example.

**Example 3-9**
For map composition a static type constraint is placed between the domain type of the left operand and the range type of the right operand:

\[
\text{StaticCheck} : \text{Expr} \rightarrow \mathbb{B}
\]

\[
\text{StaticCheck}(\text{mk-BinaryExpr}(m_1, \text{MAPCOMPOSE}, m_2)) \triangleq
\]

\[
\text{let } ts_1 = \{ x.\text{rng} \mid x \in \text{PTO}(m_2) \cdot \text{is-MapType}(x) \},
\]

\[
ts_2 = \{ x.\text{dom} \mid x \in \text{PTO}(m_1) \cdot \text{is-MapType}(x) \} \text{ in }
\]

\[
\text{IsCompatible}_w(ts_1, ts_2)
\]

It is easy to see that this constraint holds for the expression in example 3-3:

\[
\{N_1, N, B\} \cap \{B\} = \{B\} \neq \{}
\]

Thus the map composition expression in example 3-3 is well-formed.

The dynamic type constraint (for a map composition) as defined by the function:
DynamicCheck : Expr → B

DynamicCheck(mk-BinaryExpr(m_1, MAPCOMPOSE, m_2)) ∆
let ts_1 = {x.rng | x ∈ PTO(m_2) • is-MapType(x)},
                 ts_2 = {x.dom | x ∈ PTO(m_1) • is-MapType(x)} in
IsCompatible(ts_1, ts_2)

does not hold, since

\{N_1, N, B\} ∩ \{B\} ≠ \{N_1, N, B\}

A verification condition is elaborated to show that (in this case) the
range of m_2 does not contain any values from N_1 or N. □

3.2.4 Conclusions

The problem of type checking in VDM-SL is sufficiently different from
type checking most ‘ordinary’ programming languages that a traditional
approach is not viable. This is due to the richness of the VDM-SL
type constructs, caused by the presence of union types which make
it impossible to identify a restricted set of syntactic constructs as the
only ones which may cause undetectable inconsistencies.

In this section we have presented a type system for VDM-SL, expressed
using VDM-SL itself. The type system is rather simple, using only a few
more complicated functions. As a result, we believe that it can easily
be implemented, as argued in more detail in [Plat 90].

The approach to type checking which is actually used in the VDM-SL
standard provides a more detailed answer to the question whether
or not a VDM-SL specification is well-formed than our approach. The
drawback of that approach is that it is much more complicated than
ours, which makes the static semantics more difficult to understand
and more difficult to implement. Furthermore, the usefulness of be-
ing able to distinguish between different granularity levels for well-
formedness checks has not been proven in practice.
3.3 The syntax mapping

In this section we will discuss our second contribution to the ISO/VDM-SL standard: the syntax mapping, as introduced in section 3.1.4.

The syntax mapping is defined by means of a collection of explicit VDM-SL functions, (recursively) calling one another. This style of defining the syntax mapping was chosen (instead of e.g. term rewriting) because it is very similar to the way both the static semantics and the dynamic semantics have been defined. Explicit VDM-SL functions were used because it was felt that an applicative, explicit style of specification suits problems of which the main purpose is to map one sort of construct to another.

In the following subsections we will elaborate on some technical aspects of the syntax mapping, the mapping itself can be found in [Plat 92b].

3.3.1 Pre-conditions in the VDM-SL definition of the syntax mapping

At some points in the transformation process the syntax of the constructs is changed such that information can get lost if no care is taken. Typical situations occur because in the OAS constructs of the same kind are usually kept in sequences, whereas in the CAS maps are used. E.g. in the OAS it is possible to have function definitions for functions with the same name, whereas in the CAS function definitions are kept as mappings from their name to their definition, in this way assuring that no two functions can have the same name. Pre-conditions are used in the syntax mapping to avoid such situations.

3.3.2 Transformation of a specification

The syntax mapping will reconfigure a specification written in the OAS to a ‘new’ specification compatible with the dynamic semantics. If meaning is to be attached to the CAS specification then the syntax mapping must produce a series of maps which are compatible with the dynamic semantics of VDM-SL. Care must be taken to ensure that each
map is disjoint as information may be lost when the maps are merged, and that all implicit constructs are redefined explicitly.

The syntax mapping sorts a specification made up of type, state, value, function, and operation definitions and reorganizes it into the following format:

\[
\begin{align*}
\text{Definitions} & \quad : \text{ValueId} \xrightarrow{m} \text{TypeDef} \\
\expofnm & \quad : \text{Id} \xrightarrow{m} \text{ExplPolyFnDef} \\
\exmofnm & \quad : \text{Id} \xrightarrow{m} \text{ExplFnDef} \\
impofnm & \quad : \text{Id} \xrightarrow{m} \text{ImplPolyFnDef} \\
immofofn & \quad : \text{Id} \xrightarrow{m} \text{ImplFnDef} \\
\valuem & \quad : \text{Pattern} \xrightarrow{m} \text{ValDef} \\
\explopm & \quad : \text{ValueId} \xrightarrow{m} \text{ExplOpDef} \\
implopm & \quad : \text{ValueId} \xrightarrow{m} \text{ImplOpDef} \\
\text{state} & \quad : [\text{StateDef}] \\
\end{align*}
\]

Information on how each part is transformed now follows.

**The introduction of additional identifiers**

At some places, an identifier is needed which is not already used at any other place in the specification, to avoid name clashes. These identifiers are needed because of the normalized shape of the CAS.

The most obvious example is the transformation of parameter lists. Consider the function \( fn \):

\[
\begin{align*}
\text{fnOAS} : & \quad I_1 \times \ldots \times I_n \rightarrow \ldots \\
\text{fnOAS} (i_1, \ldots, i_n) \triangleq & \\
& \ldots
\end{align*}
\]

The equivalent of this function in the CAS is:

\[
\begin{align*}
\text{fnCAS} : & \quad I_1 \times \ldots \times I_n \rightarrow \ldots \\
\text{fnCAS} (i) \triangleq & \\
\text{let} \, m_{k-}(i_1, \ldots, i_n) = i \text{ in} & \\
& \ldots
\end{align*}
\]

where \( i \) is a newly introduced identifier. A function \( \text{GetUnusedId} \) is used
to produce such identifiers.

The generation of quoted pre- and post-conditions for functions
The syntax mapping generates quoted pre- and post-conditions in the following cases. For all functions $fn$ in the specification, defined as:

$$fn : I \rightarrow O$$

$$fn (i) \triangleq$$

$$\ldots$$

pre $Pre_{fn}(i)$

or

$$fn (i : I) o : O$$

pre $Pre_{fn}(i)$

post $\ldots$

a definition for a function called $pre-fn$ is generated, which is defined as:

$$pre-f : I \rightarrow \mathbb{B}$$

$$pre-f (i) \triangleq$$

$Pre_{fn}(i)$

For all functions $fn$ in the specification, defined as:

$$fn (i : I) o : O$$

pre $\ldots$

post $Post_{fn}(i, o)$

a definition for a function called $post-fn$ is generated, which is defined as:

$$post-f : I \times O \rightarrow \mathbb{B}$$

$$post-f (i, o) \triangleq$$

$Post_{fn}(i, o)$
The generation of quoted pre- and post-conditions for operations
For all operations \( op \) in the specification, defined as:

\[
\begin{align*}
op &: I \rightarrow O \\
op(i) &
\end{align*}
\]

\( \text{pre} \ P_{\text{rec}_{op}}(i, \sigma) \)

and

\[
\begin{align*}
op(i : I) o &: O \\
\text{ext rd } \sigma_1 &: \Sigma_1 \\
\text{wr } \sigma_2 &: \Sigma_2 \\
\text{pre} \ P_{\text{rec}_{op}}(i, \sigma) \\
\text{post} &\ldots
\end{align*}
\]

given a state definition

\[
\begin{align*}
\text{state } \Sigma \text{ of} \\
\sigma_1 &: \Sigma_1 \\
\sigma_2 &: \Sigma_2 \\
\sigma_3 &: \Sigma_3 \\
\text{end}
\end{align*}
\]

a definition for a function called \( \text{pre-} op \) is generated, which is defined as:

\[
\begin{align*}
\text{pre-} op &: I \times \Sigma \rightarrow \mathbb{B} \\
\text{pre-} op(i, \sigma) &: \triangleq P_{\text{rec}_{op}}(i, \sigma)
\end{align*}
\]

For all operations \( op \) in the specification, defined as:

\[
\begin{align*}
op(i : I) o &: O \\
\text{ext rd } \sigma_1 &: \Sigma_1 \\
\text{wr } \sigma_2 &: \Sigma_2 \\
\text{pre} &\ldots
\end{align*}
\]
3.3. The syntax mapping

\[ \text{post } Post_{op}(i, o, \overline{\sigma}_1, \overline{\sigma}_2, \sigma_1, \sigma_2) \]

a definition for a function called \textit{post-op} is generated, which is defined as:

\[ \text{post-op : } I \times O \times \Sigma \times \Sigma \rightarrow \mathcal{B} \]

\[
\begin{align*}
\text{let } & \text{mk-}\Sigma(\sigma'_1, \sigma'_2, \sigma'_3) = \sigma', \\
& \text{mk-}\Sigma(\sigma_1, \sigma_2, \sigma_3) = \sigma \text{ in} \\
\text{Post}_{op}(i, o, \sigma'_1, \sigma'_2, \sigma_1, \sigma_2) 
\end{align*}
\]

The references to ‘old values’ in the post-condition (\overline{\sigma}_1 and \overline{\sigma}_2) which are only meaningful in operations (remember that \textit{post-op} is a function, not an operation) have been systematically replaced by newly introduced identifiers (\sigma'_1 and \sigma'_2, respectively).

**The transformation of expressions**

The transformation of expressions differs slightly from the transformations of other constructs, because the former transformation is context sensitive. The context-sensitivity arises from the fact that in some situations identifiers occurring in expressions - although syntactically the same - refer to different objects, whereas the dynamic semantics tries to treat these objects in more consistently.

**Pre-conditions of operations**

In the pre-condition of an implicitly defined operation, the state component \( a \) refers to the old value of that state component, whereas in the post-condition of that operation \( a \) denotes the new value of that state component, and \( \overline{a} \) refers to its old value. The syntax mapping, therefore, transforms all occurrences of state components in pre-conditions of operations into their corresponding old values.

**Guards of error handlers**

A similar situation as in the previous subsection occurs with guards of error handlers in operation definitions; syntactically, a reference is
made in such a guard to the 'new' value of a state component, but in fact a reference should be made to its old value. Such a transformation is specified by the syntax mapping.

**Quoted post-conditions of implicit operations**
The construction of the body for the function post-op, where op is an implicitly defined operation, requires the systematic replacement of references to old values of state components by new identifiers.

All these transformations are dealt with by adding an extra argument to the transformation functions for expressions, called \( \rho \). This argument is of the type

\[
Env = OAS.\text{Id} \xrightarrow{m} CAS.\text{Id}.
\]

and it contains information on how each identifier encountered while transforming an expression and its subexpressions should be transformed. If an OAS identifier is not in the domain of \( \rho \), then it is simply directly transformed into its CAS counterpart. This means that in all situations other than the ones described in the subsections above \( \rho \) is empty (\( \{\rightarrow\} \)).

**The transformation of type definitions**
Type definitions are reorganized into a two maps:

\[
CAS.\text{ValueId} \xrightarrow{m} CAS.\text{TypeDef}
\]

and

\[
CAS.\text{Id} \xrightarrow{m} CAS.\text{ExplFunDef}
\]

The first map has \( CAS.\text{ValueId} \) as its domain and is the types identifier in the CAS. This is used to reference the type definition after it has been translated into the CAS. \( CAS.\text{Id} \) is the the type's identifier pre-fixed by 'inv' and is used to reference the type's invariant. The invariant is changed to an explicit function definition, via a lambda expression, and this forms the range of the second map. Simplification of types is achieved by removing the distinction between tagged and untagged types by combining a tagged type's field list with its identifier, and reducing these to a composite type. This composite type can then be
treated in an identical manner to the *shape* of an untagged type.

**The transformation of value definitions**
All value definitions are extracted from the OAS definition so that the following map can be produced:

\[ CAS.\text{Pattern} \xrightarrow{m} CAS.\text{ValDef} \]

The domain holds the pattern of the value which must be unique, and the range contains the type and value expression of the pattern after both type and expression have been translated to the CAS.

**The transformation of the state**
The syntax mapping must convert the state from its OAS form: *name*\(_{OAS}\), *field reference*\(_{OAS}\), and *map* or *sequence reference*\(_{OAS}\), to the CAS representation of: *state*\(_{CAS}\), *fieldlist*\(_{CAS}\), *invariant*\(_{CAS}\), and *initialization*\(_{CAS}\). Transforming the state results in the following maps:

\[ CAS.\text{Id} \xrightarrow{m} CAS.\text{TypeDef} \]

and

\[ CAS.\text{Id} \xrightarrow{m} CAS.\text{ExplFnDef} \]

and an optional definition:

\[ State\text{Def} \]

The map *CAS.\text{Id} \xrightarrow{m} CAS.\text{TypeDef}*, represents the state’s identifier in the domain and the type of the state, reconfigured to a tagged type definition, in the range. The type of the state in the OAS comprises of the state’s identifier, its fields, and its invariant, making it identical to a tagged type in the OAS. Because of this, it is translated by the same functions used to translate other tagged types, and the result merged with the other type definition maps.

The second map, *CAS.\text{Id} \xrightarrow{m} CAS.\text{ExplFnDef}*, is the product of merging two maps containing detail on the invariant and initialization of the state. These are translated to the CAS as explicit function definition maps.
The third result, \textit{StateDef}, contains the transformed state definition. As a state definition in the CAS comprises of; \textit{stid} - the name of the state, \textit{tp} - the name of the state's type, and \textit{init} - a function describing possible initial states, it is quite simple to transform. A lambda expression is used to represent the state's initialization function, and is transformed along with the type of the state.

3.3.3 Conclusions

The syntax mapping provides a transformation of VDM-SL specifications expressed in terms of the OAS to (simplified) VDM-SL specifications expressed in terms of the CAS. As such, the process of learning the meaning of a VDM specification has two stages, the first of which is applying the syntax mapping, and the second of which is applying the dynamic semantic functions as mentioned in section 3.1.5.

Most of the transformations as defined in the syntax mapping follow the same pattern, but by making a distinction between two phases the dynamic semantics themselves are limited and size and complexity, and are therefore more easily accessible.

3.4 Conformance, compliance and tool support

In this section we will discuss the effect standardization has had on the development of tool support for VDM-SL, which tool support is needed for the effective application of VDM, and what it means to 'conform' to the standard.

The development of tool support for formal languages has evolved to a large extent in parallel to the development of those languages, and tools have become increasingly more important. Compilers, for example, remove the tedious and error-prone task of transforming a program expressed in a programming language into a program expressed in machine code. In fact, it is inconceivable that the current production of software could be achieved without such tools. For formal specification languages a wide range of tools can be envisaged. These tools can be divided into three categories:
• **Syntactic support.** Syntactic tools are tools that can be used for the manipulation of formal specification(s) (fragments). Examples in the latter category are: structure editors for specifications or proofs, type checkers, cross-reference generators, and pretty-printers.

• **Semantic support.** Semantic tools are tools that can either be used to manipulate the semantics of specifications or to validate the correctness of the specifications. Such tools can e.g. be used to develop new specifications from existing ones, or to ‘execute’ specifications. Semantic tools typically serve as active vehicles during development, or for verification of a development step. Examples of semantic tools are theorem provers, compilers, prototyping tools, semantic analysis tools and transformational tools.

• **Pragmatic support.** Pragmatic tools are used to support the management of the development of a specification. Typical examples of these are tools for version control, configuration control, journaling, status reporting, etc.

Having a standard for a specification language has a large impact on the availability of tools. Again, looking at VDM-SL, quite soon after the standardization had started, tools became available that supported the draft standard [Plat 89]. At the VDM'91 conference [VDM'91] and the FME'93 conference [FME'93] a wide range of tools were demonstrated supporting the standard. Apparently, the introduction of a standard stimulates the development of tools to a large extent. A number of reasons can be given:

• A formally defined standard leaves no space as to the interpretation of the language. It is therefore relatively straightforward to efficiently implement tools supporting the language.

• The availability of a standard can be seen as a recognition by a wide community that the language in question is ‘mature’ and accepted. This is an indication of the market potential of the language and tools supporting the manipulation of that language.

• The standard defines the requirements that specifications expressed in that language must meet, which makes them interchangeable between different tools.
The claim that a tool 'supports' a standard, however, raises another question: What does it mean for a tool to comply with a standard? The simple answer is: a tool complies with the standard if the syntax and the semantics manipulated by that tool comply with that standard. Unfortunately, some tools which intuitively would be included following this definition do not comply. Consider e.g. a prototyping tool for VDM-SL. Since VDM-SL is a non-executable language, such a tool can only support a subset of the language, i.e. an executable subset. Furthermore, the logic employed by standard VDM-SL is a three-valued logic which cannot be executed using a sequential programming language. Thus a conditional and/or logic must be employed which satisfies the standard semantics in all cases where the definedness of a logical expression can be found by evaluating the subcomponents from left to right\(^{11}\). A prototyping tool for VDM-SL necessarily accepts a language which semantically differs slightly from the standard semantics. What is needed is a mechanism for relating these language and tool variations to the standard they claim to support. This will be further discussed in section 3.4.2.

Tools for the manipulation of a specification language can differ in the type of support they provide, and in the exact definition of the language that they claim to support. As such, both the type of support provided by a tool, and the language supported by the tool, can be individually related to the standard. For example, a type checker need only be related to the syntax and static semantics described in the standard, whereas for the language supported by the tool it can be claimed that it is the same language as described in the standard, and thus also has the same dynamic semantics. It is therefore meaningful to make a distinction between language conformance to – and tool compliance with – a standardized specification language $SL$ (figure 3.3).

In the following subsections we will discuss these individually.

\(^{11}\) As an example of a case where the conditional and/or logic does not satisfy the standard semantics we can mention a disjunction between something undefined and true. According to the standard semantics, an expression like $\bot \lor \text{true}$ denotes true whereas an interpreter using the conditional and/or logic will enter an infinite recursion and thus yield $\bot$. 
3.4. Conformance, compliance and tool support

![Diagram showing conformance and compliance relationships]

Figure 3.3: Conformance and compliance

3.4.1 Language conformance

Language conformance is especially important when the specification language is merely used as a paper-and-pencil tool. A language \( L \) is said to 'conform' to a standard if it has the same syntax and semantics as the language defined in the standard \( SL \); in other words: that language is the language defined in the standard. As we saw in the previous section, sometimes it can be useful to consider a language that is to a large degree similar to the one defined in the standard, but has some (syntactic or semantic) deviations or extensions to the standard language\(^{12}\). Although such a language does not conform (in the strict sense) to the standard language, it is possible to classify deviations and extensions.

We consider language conformance at three levels:

- **Full conformance.** A language \( L \) with the same syntax, static semantics and dynamic semantics as defined by the standard, fully conforms to the standard language \( SL \). More formally:
  \[
  S_L = S_{SL} \wedge \forall s \in S_L \cdot DS_L(s) = DS_{SL}(s)
  \]

- **Extended full conformance.** A language \( L \), of which a subset can be defined having the same syntax, static semantics and dynamic semantics as defined by the standard, can be regarded as an extension of the standard language \( SL \). To claim extended full

\(^{12}\) We do not take deviations arising from machine dependencies into account, e.g. limitations on the length of identifiers, nesting depth, etc.
conformance to the standard language, the extensions must be formally related to the standard language. More formally:
\[ \forall s \in S_{SL} : Retrieve_{L \rightarrow SL}(DS_L(s)) = DS_{SL}(s) \]
where the function \( Retrieve_{L \rightarrow SL} \):
\[ Retrieve_{L \rightarrow SL} : M_L\text{-set} \rightarrow M_{SL}\text{-set} \]
is a function defining the relationship between models from the domain universe of \( L \) and models from the domain universe of \( SL \).

- **Partial conformance.** A language \( L \) that is similar to the language described by the standard, but in some respect is a (syntactic or semantic) deviation from the standard language \( SL \), can claim partial conformance to the standard language, if the deviations have been formally defined. In the extreme, the consequence of this definition would be that almost any language would ‘partially conform’ to the standard language, and therefore additional restrictions are required, limiting the kind of deviations that are allowed. For example, it could be required that deviations are only allowed if the semantics of the standard language indicates non-termination:
\[ S_L = S_{SL} \land \forall s \in S_{SL} : \]
\[ Retrieve_{L \rightarrow SL}(DS_L(s)) \neq DS_{SL}(s) \Rightarrow DS_{SL}(s) = \bot \]
This definition ensures that the meaning of a sentence in \( L \) differs from the meaning of that same sentence in \( SL \), because \( SL \) has no ‘appropriate’ meaning for that sentence.

These three types of conformance make it possible to relate languages in a sensible way to the language defined in the standard.

### 3.4.2 Tool compliance

Tools can provide different kinds of support for a specification language, and therefore tool compliance can be defined with a hierarchy at different levels:

- **Syntactic compliance.** A tool can be syntactic compliant to the standard if the type of support provided by the tool can be formally related to the syntax of the standard language. More formally, a tool \( T \) is said to be syntactic compliant with a standard
language $SL$ if it can be shown that:

$$S_T = S_{SL}$$

- **Static-semantic compliance.** A tool can be static-semantic compliant to the standard if the type of support provided by the tool can be formally related to the static semantics of the standard language. More formally, a tool $T$ is said to be static-semantic compliant with a standard language $SL$ if it can be shown that:

$$SS_T = SS_{SL}$$

- **Dynamic-semantic compliance** (or full compliance). A tool can be dynamic-semantic compliant to the standard if the type of support provided by the tool can be formally related to the dynamic semantics of the standard language. More formally, a tool $T$ is said to be dynamic-semantic compliant with a standard language $SL$ if it can be shown that:

$$DS_T = DS_{SL}$$

This classification also provides an indication of the usefulness of a tool. For example, a parser for the language (which provides only a limited form of support, i.e. syntactic support) can only claim syntactic compliance to the standard, never full compliance.

A prototyping tool for an executable subset of a non-executable specification language could for example claim partial conformance to the standard of the language supported by the tool (because the dynamic semantics of such a language is necessarily different from the dynamic semantics of the standard non-executable specification language), whereas the tool itself could claim both syntactic and static-semantic compliance to the standard; full compliance cannot be reached.

### 3.4.3 Checking compliance

Having defined what it actually means for a tool to comply with a standard, the question of how claimed compliance can be checked becomes important. Several ways can be envisaged to check the compliance of a tool with the standard:

- **The use of test suites.** The use of test suites for checking the compliance of tools (most notably compilers) for programming languages is well-known; it is the traditional way of checking com-
pliance. The strategy consists of providing a significant number of tests, which the tool must process in the way described by the standard. Using the same strategy for checking the compliance of tools for non-executable specification languages is not without difficulties, because the expected behaviour of a tool supporting non-executable aspects of the language cannot be checked in the same way, as illustrated by the earlier mentioned example of a prototyping tool. The test suite strategy is, therefore, useful for checking the syntactic and static-semantic compliance, but it can only be used for an executable subset for dynamic-semantic compliance of tools.

- **Proving compliance.** Carrying out formal proofs showing that a tool complies with the standard is possible in theory, provided that the specification language has been formally defined. Unfortunately, since such formal definitions can be very complex and large (e.g. the semantics of VDM-SL comprises roughly 200 pages of formulæ alone!), it will usually not be worth the investment, even when proof assistants are available. We do not foresee that it will be economically feasible to carry out such proofs in the near future.

- **Falsification.** The basic idea behind falsification is that a tool complies with the standard unless proven otherwise. Such an approach is not so unreasonable as it may seem at first sight, because tools designed with no serious intent to comply with the standard would soon be falsified, i.e. specifications would be constructed which are not correctly processed by that tool. The major disadvantage of this method is that the burden of 'checking' compliance does not lie with the tool vendor, but with the standardization body.

- **Rigorous arguments.** This is perhaps the most pragmatic way of checking compliance. If a tool vendor can show that there is a systematic translation from the (formal) definition of the standard to the tool, then it is reasonable to assume that such a tool complies with the standard (the tool may still contain bugs, but that is a different matter). So, e.g. when a parser for a specification language is based on a parser generating system, syntactic compliance can be checked by comparing the underlying grammar
to the grammar defined in the standard. Of course, compliance cannot be ensured following this strategy, but it can be made plausible.

Although as yet there is no easy way to ensure or check compliance, if a specification language has a formal definition then it is potentially easier to check compliance than when no formal definition is given. Clearly it is impossible to prove compliance to a standard without a formal definition.

3.5 Conclusions

In this chapter we have described our contribution to the work currently being undertaken to standardize the formal specification language VDM-SL. The standard can be considered unique in the sense that it:

- Contains a completely formal definition of a language which is non-executable. This requires a novel approaches in areas such as the semantics of the language and to language conformance and compliance;
- has a unique approach to the definition of the static semantics of the language.

Although the standard is indeed very interesting from a technical point of view, it remains to be seen whether it will also be appreciated by industry, for two reasons:

1. The fact that the standard contains a completely formal definition of the language makes it rather inaccessible; a thorough understanding of the structure of the standard is necessary to retrieve the information that one is interested in. By splitting the abstract syntaxes in two (OAS and CAS), so that the actual denotational semantics could be defined over the simpler of the two (CAS), and by providing a mapping between these two syntaxes, the effects of this problem with respect to understanding the meaning of a specification have been somewhat limited; the syntax mapping is relatively easy to understand, and because of this approach the size of the denotational semantics could be
kept smaller than with using an approach in which only one syntax was used.

2. The static semantics provide a fine grain approach for deciding the well-formedness of a VDM specification but are, because of this, relatively complex, while the specific advantages of the approach have not been proven in practical situations. The approach that we have outlined shows a less sophisticated, but simpler approach to defining the static semantics for VDM-SL.

3. As our work on conformance and compliance has shown, it will be rather difficult for VDM-SL tool manufacturers to show that their tool conforms to the standard, both due to the size of the language and to the fact that the VDM-SL semantics are non-executable.

The main merit of the standard is that it has been completely formally defined, which has clearly highlighted advantages and disadvantages of the choices that have been made for the various components of the language. Simplifications in such standards could be achieved if:

- different kinds of semantics are used, e.g. axiomatic semantics or operational semantics. Axiomatic semantics have the advantage that they are usually smaller in size. Operational semantics are more intuitive to understand, but can only be used for executable specification languages.
- specification languages are used which are smaller, and therefore less expressive.

It remains to be seen whether or not the VDM-SL standard will be a success in practice.
The application of VDM in DoD-STD-2167A

The aim of this chapter is to analyze the use and application of VDM in a widely accepted framework for software development, in this way making it possible to position the use of formal methods and evaluate their usefulness. DoD-STD-2167A was chosen as the framework in which to analyze the use of VDM, because it provides a precise description of phases generally accepted as a part of a software process.

The chapter starts with an overview of DoD-STD-2167A. Then, in section 4.2, the integration of VDM in DoD-STD-2167A is analyzed. In section 4.3 we discuss how VDM can best be used in DoD-STD-2167A. In section 4.4 a number of conclusions on where and how VDM can best be used in DoD-STD-2167A are presented.

This chapter has also been separately published as reference [Plat 92c].

4.1 Overview of DoD-STD-2167A

DoD-STD-2167A is the software development standard required on most
United States Department of Defence contracts today, and it has gained a certain popularity in application areas other than the military, both in the United States and in Europe.

The standard defines a collection of (adaptable) requirements for a software process by describing:

- the *phases* of a software process and the *activities* which have to be carried out during each phase, and
- the *documentation* that has to be produced during the software process and by which the progress of the software process can be measured.

The standard neither requires the use of a particular design method (although the design methods used have to meet criteria like well-documentedness and maturity) nor does it require that the software process is based on a particular life cycle model; conventional models like the Waterfall model can be used, but the use of rapid-prototyping or the Spiral model is possible as well.

### 4.1.1 Static system structure

One of the fundamental concepts of DoD-STD-2167A is the *static structure* for system and software development projects. This static structure provides the basis for top-down partitioning a system’s design, and consists of the following components:

- **Hardware Configuration Items (HWCI*s)**. HWCI*s are hardware deliverables. Although DoD-STD-2167A recognizes their existence as a part of the complete system, the development of HWCI*s themselves is not supported by DoD-STD-2167A.
- **Computer Software Configuration Items (CSCI*s)**. CSCI*s are (software system) deliverables. DoD-STD-2167A applies in particular to the development of CSCI*s.
- **Interface Requirements Specifications (IRSI*s)**. IRSI*s specify how CSCI*s interact with other CSCI*s and with HWCI*s, i.e. communication protocols, requirements on the data being interchanged, etc.
4.1. Overview of DoD-STD-2167A

- **Computer Software Components (CSCs).** CSCs are distinct parts of CSCIs, which may be further refined into other CSCs or Computer Software Units (CSUs).
- **Computer Software Units (CSUs).** A CSU is an element specified in the design of a CSC that is separately testable. CSUs can be seen as basic software components which are not further refined.

The development project itself is usually organized along the same line as the static structure. So, supposing that the aim of the project is to develop a single CSCI consisting of three different CSCs, the project team might be divided into three subteams, each responsible for the development of one CSC. An example of the static structure of a system in the defined terms is shown in figure 4.1.

![Diagram](image)

Key:
- System component, refined as shown in picture
- Refinable system component (refinement not shown)
- Not-refinable system component
- Consists-of hierarchy connection

Figure 4.1: Sample system breakdown using DoD-STD-2167A terminology
4.1.2 Phases and activities

DoD-STD-2167A requires that the software process is divided into a number of phases, including at least the following:

- system requirements analysis/design;
- software requirements analysis;
- preliminary design;
- detailed design;
- coding and CSU testing;
- CSC integration and testing;
- CSCI testing;
- system integration and testing.

During each of these phases, a number of activities are recognized, which can be grouped in the following categories of activities:

- software development management;
- software engineering;
- formal qualification testing;
- software product evaluation;
- configuration management.

The activities done are different in each phase. Therefore, the main part of the standard is structured as a matrix: for each phase it is described which activities per category of activities have to be carried out by the contractor, and the documentation that has to be produced is specified.

4.1.3 Documentation

DoD-STD-2167A is sometimes called ‘the documentation standard’ because, apart from the system being developed, a vast amount of electronic documentation must be produced during a project as well. The format of the required documentation is described in Data Item Descriptions (DIDs). These DIDs provide a way of recording and communicating information about the project and the system to the contractor. Just like the standard itself, the DIDs may be tailored to reflect the specific needs of the project. The elements that each DID should contain are precisely described in the standard.
4.2 Applying VDM in the context of DoD-STD-2167A

The following criteria can be defined to make the application of VDM in DoD-STD-2167A possible:

- VDM-SL specifications must be usable as (part of) a design document as defined by DoD-STD-2167A.
- A (part of) a development path addressed by VDM must be the same as a (part of) a development path addressed by DoD-STD-2167A.

4.2.1 The static structure of DoD-STD-2167A and VDM

A prerequisite for the use of any formalism in the software process is that there must exist a direct link between the constructs recognized by the software process model (in this case the static structure as defined by DoD-STD-2167A) and the constructs in the formalism (in this case VDM-SL language constructs). Such a relation between Ada-constructs and DoD-STD-2167A static structure constructs was investigated by Grau and Gilroy in [Grau 87].

The static structure of DoD-STD-2167A defines the decomposition of a system in terms of CSCIs, CSCs and CSUs. The relationship between the latter three constructs can be defined as a consists-of-hierarchy. Although the original versions of VDM-SL do not contain any structuring mechanism, the need for such a mechanism was recognized by BSI Panel IST/5-/19, which resulted in a proposal for a structuring mechanism based on modules [Bear 88]. The module mechanism, in its turn, was inspired by the structuring mechanism for the STC Reference Language [Shaw 85]. Currently, this module mechanism only exists as a syntactical notion, the underlying semantics are still under investigation.

The relationship between VDM-SL modules can be defined in terms of a uses-hierarchy; modules can import and export constructs to one another, but they cannot be defined in terms of one another. Consequently, CSUs can be mapped onto VDM-SL modules, but there exists no mapping from CSCIs or CSCs to VDM-SL modules.
4.2.2 The use of VDM for the software engineering activity in DoD-STD-2167A

The software engineering activity in the DoD-STD-2167A model of the software process is the activity directly concerned with the realization of the software system. DoD-STD-2167A prescribes exactly what must be done to design and implement the software system in a controlled way. VDM can be used for the software engineering activity, thus directly influencing the results reached for the software engineering activity. An overview of the benefits of each way of using VDM for both the software engineering activity and other activities in DoD-STD-2167A is given in table 4.1.

System requirements analysis/design – Software requirements analysis

In the System Requirements Analysis/Design Phase the contractor must analyze the system requirements and partition the system into HWCIs, CSCIs, and manual operations. Furthermore, the contractor must define preliminary sets of interface requirements for each IRS, and preliminary sets of engineering requirements for each CSCI. These two preliminary sets of requirements are completed during the Software Requirements Analysis Phase.

System requirements
Although no direct benefit can be expected from formally specifying system requirements, a consideration to do so anyway might be that in this way the ‘interface’ between the customer and the contractor can be made unambiguous. Software requirements specifications are often used as ‘contracts’ between customers and contractors. On the one hand, the consideration that the communication between these two parties should be as precise and consistent as possible (with respect to the software requirements) could lead to the conclusion that using VDM (being formal and thus precise) for this purpose would be advisable. On the other hand, most customers confronted with a Software Requirements Specification have little or no experience with VDM. This means that understanding the Software Requirements Specifi-
<table>
<thead>
<tr>
<th>DoD-STD-2167A</th>
<th>The use of VDM as a</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activity</strong></td>
<td><strong>Phase</strong></td>
</tr>
<tr>
<td>Software development management</td>
<td>n.a.</td>
</tr>
<tr>
<td>Software engineering</td>
<td>System requirements</td>
</tr>
<tr>
<td></td>
<td>analysis/design</td>
</tr>
<tr>
<td></td>
<td>Software requirements</td>
</tr>
<tr>
<td></td>
<td>analysis</td>
</tr>
<tr>
<td></td>
<td>Preliminary design</td>
</tr>
<tr>
<td></td>
<td>Detailed design</td>
</tr>
<tr>
<td></td>
<td>Unit testing</td>
</tr>
<tr>
<td></td>
<td>Code and System integration</td>
</tr>
<tr>
<td></td>
<td>and testing</td>
</tr>
<tr>
<td></td>
<td>CSC testing</td>
</tr>
<tr>
<td></td>
<td>System integration and</td>
</tr>
<tr>
<td></td>
<td>testing</td>
</tr>
<tr>
<td>Formal qualification testing</td>
<td>n.a.</td>
</tr>
<tr>
<td>Software product evaluation</td>
<td>n.a.</td>
</tr>
<tr>
<td>Configuration management</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

**Key:**
- No application of VDM possible
- Benefits from usage of VDM in other phases/activities
- Application of VDM possible, benefit doubtful
- Beneficial application of VDM possible.
- Supported by industrial experience

Table 4.1: The application of VDM in DoD-STD-2167A
cation requires an extra effort from the customer. Still, Craigen et al. [Craigen 93a] claim that one of the main uses of formal methods is in the area of communication. In [Alford 80] it is claimed that the use of natural language would most probably improve the communication between the customer and the contractor. A combination of using VDM and natural language can be a worthwhile compromise, especially in those situations where an unambiguous interpretation of the mentioned aspects that can be expressed using VDM is vital. An alternative is the use of formal specification methods especially tuned to specifying user requirements, such as PAISLey [Zave 81, Zave 82] and RML [Bordiga 85]. An overview of requirements modeling languages can be found in [Davis 88].

**Interface requirements**
The interface requirements are used to relate the CSCIs to real world entities. DoD-STD-2167A requires that these external interfaces should be expressed in the 'language' of this real world. A formal specification of the external interfaces would not be particularly helpful, therefore. Another aspect of interface requirements is the specification of the communication protocols between various components. The formal specification language LOTOS [Bolognesi 87, ISO8807] was especially designed for this purpose.

**Engineering requirements**
The engineering requirements for the CSCIs can be divided into capability requirements and internal interface requirements.

Capability requirements consist of detailed descriptions of the purpose and performance of a CSCI. In [Chedgey 87] the use of VDM for specifying software requirements is discussed; possibilities for using VDM-SL in this phase are identified, for example by using pre- and post-conditions for describing essential properties of functions to be provided by the system, and invariants to describe global properties and non-functional requirements.

During requirements analysis, little is known about the relations that
may exist between the different components that form the system. Using VDM for this purpose can therefore only concentrate on these isolated components, leading to separate 'chunks' of VDM specifications. This use of VDM as a rigorous specification method can be useful in those situations where there is a specific need for being formal. The benefit of using VDM as a rigorous specification method is limited at this point, since it is not possible to reason formally about such a specification (the formal specification is by definition incomplete).

**Preliminary design**

In the Preliminary Design Phase an architectural design of the software system is constructed. In DoD-STD-2167A terms this means that each CSCI is partitioned into several CSCs, and that engineering requirements of the CSCIs and the interface requirements of the IRS defined in the Software Requirements Analysis Phase are distributed between the CSCs. The contractor is furthermore required to document information that might be relevant in understanding the preliminary design, such as the rationale for design decisions made, results of analyses, etc., and to develop test requirements for conducting CSC integration and testing.

**CSCI design**

A key issue in the design of software systems is the construction of an (abstract) model of that system. VDM-SL offers a suitable notation for expressing such a model (the use of VDM as a formal specification method), with the following advantages over a non-formal specification:

- VDM supports both operational abstraction (pre- and post-conditions) and data abstraction (sets, maps, and sequences), in this way helping the designer in managing the complexity of the system being constructed.
- A formal specification in VDM-SL is unambiguous, because the meaning of VDM-SL itself has been formally defined. This makes it possible to:
  1. prove characteristics of the specification. Such proofs are useful for the verification of the result of the Preliminary
Design Phase against the system requirements;

2. use the formal specification for walk-through sessions. Walk-throughs result in the (early) detection of errors in the design (early detection means cost reduction) and a greater understanding for the members of the design team of the entire system.

This observation is supported by practical experience in the CIP project. Möller and Partsch report [Möller 87, page 507]: "As soon as we had at least created the skeleton of a formal specification [...] things improved considerably: "technical" discussions then became really technical (and thus more fruitful); any discovered "problem" either could be clearly identified on the basis of the skeletal specification (and sometimes even be solved) or recognized not to be a real problem at all." Shaw and others [Shaw 84, pages 64,65] support the observation as well. They report on the introduction of VDM in their development environment, and remark that “VDM provided a number of useful thinking tools” and “VDM enhanced the analytical power of the analysts because it allowed them to reason about the abstract models which were developed.” Austwick and Norris report [Austwick 86, page 24]: “[VDM] encourages the designer to think about the specification. This gives more correct specifications and early identification of inadequate specifications.”

3. use the formal specification as a prototype. Such a prototype can be used to validate the original user requirements. A prerequisite for such use of formal specifications is that the specification in question is executable, which a VDM specification is not necessarily. An interpreter for executable VDM specifications is described in [Larsen 91].

An already mentioned disadvantage of using VDM for the specification of large system models is that VDM does not offer well-defined support for the structuring of specifications.

Another limitation of VDM in this respect – as opposed to some other formal specification methods – is that it is very difficult to express concurrency in VDM-SL. Successors to VDM, such as RAISE [Prehn 87]
and VDM++ [Dürr 92, Dürr 93], have been developed to overcome this disadvantage.

A deficiency of formal methods in general for the formal specification of systems designs is that they do not offer any assistance in deciding which requirements of the CSCl can be best grouped together and allocated to a specific CSC. Crispin describes in [Crispin 87] the experiences within STC with VDM, and he recognizes the same deficiencies: "The architectural design [...] was not supported by a recognized 'method' at all". Other development methods, like JSD (Jackson System Design), SA/SD, or object-oriented design offer some decision support, however, non-formal. Crispin reports: "progress was slow until there was a project decision to adopt an 'object-oriented' design." and "The most that can be said is that the two approaches worked extremely well together." Further development of VDM as a formal design method, for example by combining VDM with techniques provided by non-formal analysis/design methods, will most likely improve the results reached for this subactivity; this is the subject of chapter 5 of this thesis.

**Interface design**

The design of an interface consists of a detailed description (a so-called data element definition table) of the data elements that are passed through the interface. The data element definition table contains for example a brief description of the data element, the units of measure, which CSCls and HWCls use the data element, the required precision, legality checks, etc. VDM can be used as a rigorous specification method for the specification of the data types of the data elements. This use of VDM is not without difficulties, however. Suppose that two CSCls interfacing with each other must be designed. If the data types of the data elements that are passed through the interface would be specified in VDM-SL, then it would very well be possible that the design team for the first CSCl would use a different representation for those data types at the coding level than the design team for the second CSCl. Needless to say that such a practice would cause serious problems when the two CSCls were integrated in the system. To solve this problem, it is necessary to first develop the interface in terms of the programming
language being used. The latter interface can then be used by the separate development teams.

As far as possible, concurrency should be specified implicitly, not explicitly, so that the system developers are free to choose what level of concurrency to use while implementing the system. This is certainly a contentious point and other authors, e.g. Zave [Zave 82] would argue in favor of explicitly modeling concurrency. The primary argument in favor of the implicit approach is that it does not involve making premature design decisions.

**Detailed design**

The first major activity during the Detailed Design Phase is the continued development of the CSCs: the CSCs that were identified during the Preliminary Design Phase are further partitioned into a number of CSUs. Design requirements for the CSCs are allocated to CSUs. Again, engineering information that is gathered during the detailed design process, such as results of analyses and trade-off studies, must be documented as well. The second major activity during the Detailed Design Phase is the definition of test requirements, test responsibilities and test cases for every CSC and CSU in the system.

**Design of CSUs**

VDM can be used as a *formal verification method* for the design of CSUs. Two elements can be identified for this use of VDM:

1. The use of VDM-SL to express an initial design (the target specification of the Preliminary Design Phase), (possibly) intermediate designs, and a detailed design which is a source specification for the Detailed Design Phase.

2. The use of refinement techniques, as illustrated in detail in [Bjørner 82] and [Jones 90a]. The initial design, expressed in VDM-SL in terms of high level components (CSCs) is refined into a low-level design in terms of CSUs. Such refinement is done in an (arbitrary) number of steps. In each step, data structures are refined into data structures closer to ones that can be ex-
pressed in the target language, implicit constructs are converted into explicit ones, and operations are decomposed. Ideally, the final specification is in such a form that transforming this specification into a source program is a minor step. Each step taken during this process results in proof obligations which have to be satisfied by the user.

A formal verification method reduces the need for testing whether a specification meets its functional requirements. In cases where it has been proved that the refinement is correct, theoretically spoken, no testing is necessary. Formally verifying *everything* that can be verified, however, can be a very time-consuming and tedious task. Furthermore, such proofs tend to get very complicated and thus need verification themselves, for example by using proof-assistants. In our opinion, formal verification methods should only be used when there is a necessity (either a requirement from the customer, or resulting from the application domain, for example safety-critical systems) to do so. In all other cases the use of a *rigorous verification method* suffices. Parts of the verification activities can then be replaced by testing or by rigorous arguments, thus saving time and money.

**Definition of test cases**
Although not very common, the usage of VDM can help to derive test cases, i.e. suitable sets of input values for CSUs. Scullard describes in [Scullard 88] a project where such an approach was used for test case selection based on VDM representations of hardware components. The procedure is to take each function and operation separately and to consider the structure of its domain to identify a reasonable sample. The signature of the function is reduced to basic types only; composite types are (possibly recursively) replaced by the cartesian product of their components. Further reductions are accomplished by considering the pre-condition of the function or operation and data type invariants on the domains in the signature. From this signature a collection of typical values is extracted; in the case of unordered types all values are chosen, for ordered types extreme values and one or more central values are chosen. This test set is extended by skolemizing the post-condition of the function or operation and adding those values that are
at the boundaries of the subdomains revealed in this way. Although the approach causes a rather large number of test cases, the possibility is created to construct an automated test generation system. This use of VDM can be considered as a benefit of the use of VDM as a rigorous specification method. This approach has been taken up further by Dick et al. [Dick 93].

Coding and testing

The last four phases recognized by DoD-STD-2167A concentrate on the development of test procedures, testing, and either making revisions in design documentation and code (followed by re-testing) if errors were found during testing, or integrating the tested components into higher-level components. This process starts by coding and testing the lowest-level components (CSUs) and ends with the integration tests for the highest-level component (the system).

Coding

In theory it is possible to regard the low-level specification, which is the result of the Detailed Design Phase, as the 'code' of the system, provided this specification contains no implicit constructs and is not loose. In general, however, refined specifications will not meet these requirements and, therefore, an additional transformation into a programming language (coding) will have to be made. VDM does not enforce the use of a specific programming language and is, in that sense, no hindrance during this stage.

Specific specification styles, however, do hint towards implementations in a certain class of programming languages to which a direct translation can take place. Functional specifications for example encourage the use of a functional programming language, and specifications using a lot of explicit operations encourage the use of an imperative language. In [O'Neill 88] an analysis of the translation of VDM specifications into Ada code is made. The relationship between implicit VDM specifications and logic programming (in casu Prolog) is touched upon in [Bloomfield 86].
More troublesome situations can occur when there is a large discrepancy between the language constructs present in the target programming language and VDM-SL. In [Chedgey 87] this is illustrated by a data encapsulation facility present in the Ada language: packages. Again, the lack of a well-defined structuring mechanism in VDM causes problems in this respect.

**Testing**
Testing is an activity that can only indirectly benefit from the application of a formal method at an earlier stage; formal specifications can be used as reference points against which can be decided what the expected behaviour of a piece of source code should be.

Theoretically spoken, formal verification methods could eliminate the need for testing the functionality of the system; once it has been proven that a function behaves according to its specification, there is no need to test such a function anymore. Testing, which is an obligatory activity in DoD-STD-2167A, cannot provide complete assurance that a system will behave correctly in all situations, and, therefore, it may be worthwhile to prove at least the critical functions in the system to be correct.

**4.2.3 The use of VDM for the other activities in DoD-STD-2167A**

Although VDM can only be directly applied for the software engineering activities, its use may also benefit the other activities required by DoD-STD-2167A, though indirectly. There are no benefits resulting from the use of VDM for the software product evaluation activities and the configuration management activities, but both the software development management activity and the formal qualification testing activity can benefit from its use. These benefits will be briefly discussed in the following subsections.

**Software development management**

DoD-STD-2167A puts much emphasis on the formal establishment of a software development process and the activities that have to be carried
out to control and plan this process (management).

Using VDM may improve the accuracy of the planning. The planning can be more accurate, because the use of VDM will minimize the number of mistakes made during the early phases of the development. Since the planning can be more accurate, deviations from this planning will be noticed sooner, and corrective actions can be undertaken earlier. The design will be better understood, so management is easier. This hypothesis, however, has only been supported by Sadler [Sadler 88] who mentions the positive experiences of Rolls Royce Inc. for this particular aspect of formal methods. Dandanell et al. report in [Dandanell 93] that according to their experience there were no difficulties in using the formal method RAISE within existing management standards and procedures, but that at the same time formal methods should not be regarded as a good remedy for poor management.

**Formal qualification testing**

Besides the tests that are carried out during the software engineering activity at the CSC level, DoD-STD-2167A also requires the establishment of a separate formal qualification testing activity during which CSCI tests and system integration tests are carried out. These tests are considered to be 'formal' (versus the non-formal tests that are part of the software engineering activity) in that the procedure of their execution is prescribed by DoD-STD-2167A.

Apart from this fact, there is no significant difference with the non-formal tests of the software engineering activity, so the same benefits resulting from these tests also apply for the formal software tests.

### 4.3 Modes of VDM application

#### 4.3.1 The use of VDM as a specification method

For the vast majority of software systems being built nowadays, the use of VDM as a *formal specification method* has benefits over using comparable non-formal methods:
1. a formal specification in VDM has a mathematically defined meaning. This means that it is possible to prove characteristics of the formal specification, which can be extremely important when dealing with for example safety-critical systems;

2. VDM-SL has extensive means to express representational and operational abstraction. So, when writing or studying VDM specifications it is possible to concentrate on the relevant issues (at that stage of development) only;

3. since a formal specification is precise, writing down a formal specification forces the designer to make decisions at an early stage, which will limit the implementation risk, and

4. formal specifications can be treated in the same way as the code of the implementation: ‘specification walk-throughs’ can be carried out by development teams. This means that errors can be detected at an early stage of the development, and a consistent view on the functionality of the system is obtained by the entire development team. This use of a formal specification as a ‘thinking-tool’ contributes to cutting down the costs of the total software process.

Beyond this, to assist the writers of specifications, computer-based tools are currently available. Examples of such tools are syntax directed editors, type checkers, and rapid prototyping tools. The availability of proof assistants is essential for carrying out all the proof obligations in an economical way.

The use of VDM as a rigorous specification method in DoD-STD-2167A is highly isolated and limited to the precise specification of part of the non-functional requirements. Although this can be beneficial occasionally, it is probably better to use a requirements modeling language for this purpose, which will, in addition, allow the specification of the entire collection of user requirements.

4.3.2 The use of VDM as a verification method

The use of VDM as a rigorous verification method is probably the most widespread use of VDM at the moment. The proof obligations presented by this way of using VDM can increase the confidence in the develop-
ment steps taken, but need not necessarily be carried out; usually rigorous arguments suffice. Actually carrying out the proof obligations becomes particularly important when systems are developed that are difficult to test (for example embedded missile software) or have high demands with regard to preventing possible failure (for example software systems used in nuclear power plants). So, the benefits of using VDM in this way can be considerable.

However, there are some problems when DoD-STD-2167A is used in combination with VDM as a formal verification method: the verification activity in DoD-STD-2167A focuses on testing, whereas the verification activities for VDM consist of fulfilling proof obligations, thus (at least partially) eliminating the need for testing.

Another drawback is that it turns out that in reality the proof obligations are hardly ever actually carried out by hand, and that automatic support, such as e.g. a proof assistant, is considered essential, because fulfilling proof obligations can be a very complex task.

4.3.3 The use of VDM as a rigorous development method

The use of VDM as a rigorous development method is not very common nowadays. The ideas behind this approach are briefly mentioned in chapter 2, but no data was available on how this approach works out in reality.

The importance of the extension of VDM with a design component is generally recognized, however. Experiences suggest that such an extension could lie in the connection of VDM to structured design methods, such as SA/SD, object-oriented design, JSD, etc. The connection between VDM and such structured design methods is discussed in chapter 5.

An application in the future of VDM as a formal development method requires the extension of the rigorous development method VDM with a collection of transformation rules that describe exactly how to transform a VDM specification, possibly in a number of steps, into an implementation. In our opinion, this approach has several benefits, which
are made even stronger by the fact that this transformation can be automated to a large degree (as argued in chapter 2):

1. the software process would become more predictable, and thus it would be easier to manage the software process;
2. since the transformation from a specification into an implementation could be done automatically, the software process could be carried out in a more effective and efficient way, which would lead to a reduction in the cost of the software system being developed.

Verification would still be necessary, since verification would be needed whether the transformation rules have been applied correctly.

4.3.4 Where to use VDM in DoD-STD-2167A

The best phase to start using VDM in DoD-STD-2167A is the Preliminary Design Phase. During this phase an initial system design must be constructed, for which VDM can be used as a formal specification method. Then, depending on the application domain, the use of VDM as a rigorous development method or as a rigorous or formal verification method, can continue through the Detailed Design Phase up until the Coding and Testing Phase. The use of VDM in other phases and for other activities is sometimes possible, but it is debatable whether this is beneficial with respect to the use of non-formal methods for the same purpose.

4.4 Conclusions

4.4.1 The application of formal methods in the software process

The analysis of the application of VDM in DoD-STD-2167A has shown that it is possible to use formal methods in the software process for a variety of activities, starting during preliminary design through detailed design and coding and testing. During these phases, formal methods provide distinct benefits over their non-formal counterparts:
they include formal notations which can be unambiguously interpreted and which – for a large class of these formal notations – provide extensive means to express (representational and procedural) abstraction. These notations can be used to verify formally characteristics of the design. Besides this, they are used as ‘thinking tools’ that have shown to be important aids during design;

- they provide means to formally verify the correctness of design steps made. Although it is clear that formal verification cannot be done when using non-formal methods, due to technical reasons it is currently not always feasible to completely prove the correctness of all design steps. The application area should be considered when deciding whether to prove the correctness of all design steps (completely formal), some of the design steps (rigorously formal) or none of the design steps (non-formal) made.

Some less important applications of formal methods have been identified as well. These applications have not yet been well developed, however, and should, therefore, only be applied after thorough consideration.

4.4.2 Which formal method to use?

The software process encompasses a wide range of activities. Currently, no formal method exists which is equally suited for supporting all these activities. When it is decided to use formal methods, a choice must be made from the available formal methods, based on (among other criteria) for what the mathematics employed by such a method are most relevant to the system being developed. VDM, for example, can best be applied for the development of functional aspects of a system. Other methods, for example Temporal Logic, put emphasis on a formal approach to the timing requirements of a system. Which formal methods can best be used thus does not only depend on the kind of support (specification, verification, development) offered, but also on the aspects of the system for which the mathematics are employed.
4.4.3 Improvements of formal methods

Until now, formal methods have been based on one strong concept, *mathematics*, using very simple paradigms for the development of software. They do not (completely) address the issues which are important for the development of large software systems. It has been shown that application of formal methods in an established software process model is indeed possible and beneficial, but problems occur with the integration of these methods.

The first area where improvements can be made is in the construction of the initial source specification in the Preliminary Design. None of the known formal methods provide support, either by providing guidelines for the actual construction, or by providing requirements that an initial source specification should fulfill, *how* such an initial source specification can be constructed (see e.g. [Dandanell 93]). To improve this, formal methods should be extended (for example with ideas from another already existing design method) to make a structured composition of an initial system model possible. This is the topic of chapter 5 of this thesis, in which the construction of VDM specification using Structured Analysis is investigated.

A second area for improvement is the connection with the coding phase of the software process. The step to go from a formal specification to source code may sometimes seem trivial, but this is certainly not true for large programs. In the case of VDM, the problem can be approached in two ways:

1. defining an executable subset of VDM-SL which can be used as a programming language. To make it possible to use this subset for large programs, it would then become even more necessary to extend VDM-SL with a proper structuring mechanism and, most likely, other constructs present in modern programming languages as well;

2. defining a formal relationship between VDM-SL and the programming language used. The existence of a formal relationship makes it possible to prove that the transformation from a low-level specification into a program expressed in that programming language is correct.
The importance of tool support for the development of software has increased during the last three decades, and will also be essential for the effective use of formal methods in industry. There are several computer-based tools that support working with formal methods. These tools, however, are suited for the support of the specification and verification activities only. However, by their nature formal methods seem to be very well suited to be supported by computer-based tools for the development activities performed during the software process.
Structured methods for software development were introduced in the late seventies, and have gained popularity during the last decade in industrial organizations. Structured methods are methods for software analysis and design, based on the use of heuristics for making analysis and design decisions. They provide a relatively well-defined path, often in a cookbook-like fashion (hence the term 'structured' methods), starting from the analysis of software requirements and ending at system coding. The design notations used are usually graphical and have no formal basis. In that sense structured and formal methods can be regarded as complementary. It is often suggested that the informal graphical notations as provided by structured methods are intuitively appealing to software analysts/designers. Therefore, a combined structured/formal method may not only increase our understanding of the use of formal methods in the software process, but may also increase the acceptability of formal methods to these people.

A quite natural approach to combining structured and formal methods seems to be what is usually called a parallel approach. In this ap-
The application of VDM in combination with SA

approach, the software system is developed using a conventional structured approaches while in parallel a second development takes place using a formal method. Although such an approach is useful as a basis for developing safety-critical systems [Leveson 91], we do not believe that the approach is useful to convince people to use formal methods themselves. Since the parallel step will often be done by designers trained in formal methods, those who have to be convinced, i.e. the ones taking the classical approach, may not be impressed at all. On the contrary, the effect might even be negative [Kemmerer 90]. In this chapter we will, therefore, pursue a integrated approach, i.e. an approach in which a combined structured/formal method is used. In the context of the transformational model for software development as introduced in chapter 2, a combination\textsuperscript{13} of two methods \( m_1 \) and \( m_2 \) involves the definition of a relationship between what is produced as a target specification of \( m_1 \) and the source specification of \( m_2 \), i.e. the relationship defines how the target specification of \( m_1 \) must be transformed so that it can be used as the source specification of \( m_2 \). If such a relationship is formally defined, and if \( m_2 \) is a formal method, then the source specification of \( m_2 \) can be regarded as a formal semantics of the target specification of \( m_1 \). In our situation, where \( m_1 \) is Structured Analysis and \( m_2 \) is VDM, we will discuss possible relationships between data flow diagrams (DFDs; the main specification notation used in Structured Analysis) and VDM specifications in section 5.2. A formal specification of this relationship has been defined but is not discussed in this thesis; the interested reader is referred to [Larsen 93a].

In section 5.3, the methodological aspects of the approach are discussed and a small example is presented. Section 5.4 contains a number of conclusions. Parts of this chapter have been published as references [Plat 91b], [Larsen 93b], [Larsen 93a], and [Plat 93].

\textsuperscript{13} Notice that the ‘combination’ of two methods is not necessarily the same as an ‘extension’ of a method (as e.g. the extension of VDM with rely/guarantee conditions to specify concurrent, shared-variable systems [Jones 87a], the object-oriented extension of VDM, called VDM++, [Dürr 92, Dürr 93], or the extension of VDM as defined in [Ledru 93] to model reactive systems).
5.1 Overview of Structured Analysis

Structured Analysis (SA) [Yourdon 75, DeMarco 79, Gane 79] is one of the most widely used methods for software analysis. Often it is used in combination with Structured Design (SD) [Constantine 79]; the resulting combination is called SA/SD. The approach to analysis taken in SA is to concentrate on the functions to be carried out by the system, 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 design product of SA, is called a data flow diagram (DFD). A DFD is a directed graph consisting of elementary building blocks. Each building block has a graphical notation (figure 5.1).

Figure 5.1: (a) Data transformer; (b) Data flow; (c) Data store; (d) External process

Through the years several dialects have evolved and extensions have been defined (e.g. SSADM [Longworth 86] and SA/RT [Ward 85]), but we will limit ourselves to DFDs with a small number of different building blocks¹⁴:

1. **Data transformers.** Data transformers are usually denoted as bubbles. They may have any number of inputs, and any number of outputs.

2. **Data flows.** Data flows are represented as arrows, connecting one data transformer to another. They represent a flow of data between the data transformers they connect. The flow of data is unidirectional in the direction of the arrow.

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

4. **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

---

¹⁴. The kind of DFDs that we use are powerful enough to model any kind of sequential system.
the output of the system is going to.

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. Therefore, DFDs are often combined with data dictionaries, control flow diagrams, state transition diagrams, decision tables and mini-specifications to provide a comprehensive view of all the aspects of the system. These additional notations are not used in our combined SA/VDM method.

The process of constructing a DFD is an iterative process. Initially, the system to be designed is envisaged as one large data transformer, getting input from and providing output to external processes. This initial, high-level DFD is called a context diagram. The next step is the decomposition of the context diagram into a network of data transformers, the total network providing the same functionality as the original context diagram. This process is repeated for each data transformer until the analyst/designer considers all the data transformers in the DFD to be primitive, i.e. each data transformer performs a well-defined, well-understood function. We call such a collection of DFDs, describing the same system but at different levels of abstraction, a hierarchy of DFDs. SA/SD prescribes a transformation into structure charts at this point. Structure charts are a notation for describing the structure of a system in terms of the system components (modules), they have been given a minor role in our combined SA/VDM method (see section 5.3).

5.2 The transformation of data flow diagrams

In this section we will describe the underlying principles of how DFDs can be transformed into VDM constructs. If certain restrictions to DFDs are applied, this transformation can be fully automated. Our approach to the construction of a formal specification is to formalize the intuitive semantics that can be given to DFDs in a structured way, which, in combination with knowledge of the representation of data in the system and knowledge of the control flow, will lead to a usable formal specification of the system.
5.2.1 Relation to other work

When DFDs were originally introduced, they were presented as a graphical notation. The intended semantics of this notation was defined verbally, but the need for a formal base is now more commonly recognized, see e.g. [Hofstede 92]. Work has been done on formalizing DFDs, with the intention of either disambiguating their meaning, or of using the formal semantics as a base for an integrated formal/structured method.

In [Randell 90] a translation back and forth between DFDs and Z specifications is described. [Alabiso 88] contains an explanation of how DFDs can manually be transformed into an object-oriented design. The paper touches upon some problematic issues arising in a transformation from DFDs. In [Semmens 91] a small example of how a DFD can be transformed in Z is presented. However, 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 into a Petri net variant combined with path expressions. In [Adler 88] 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. Jones uses VDM to provide a denotational style semantics of a non-conventional machine architecture (The Manchester DataFlow Machine) based on data flow graphs [Jones 87b]. In [Fraser 91] a rule-based approach for transforming SA products into VDM specifications is presented. Their VDM specifications are very explicit and hard to read, mainly because of the way decision tables have been taken into account. Polack et al. concentrate on the methodological aspects of combining SA notations and Z specifications [Polack 92, Polack 93], the resulting combination is known as SAZ. Tse and Pong use extended Petri nets for formalizing DFDs [Tse 89]. France discusses an algebraic approach to modeling control-extended DFDs in [France 92]. In [Semmens 92] an overview of several approaches to combining SA techniques and notations with formal methods (including our approach) is given.
5.2.2 A strategy for mapping DFDs onto VDM constructs

Our strategy for mapping DFDs onto VDM constructs first follows the traditional approach of SA to construct a model of the system (the DFD) by identifying data transformers and data flows, and refining the data transformers until all data transformers are considered to be primitive. We will refer to the DFD that is constructed in this way as the low-level DFD. Additionally, rules are provided for the transformation of a low-level DFD to appropriate VDM counterparts. Different transformations exist, two of which are described in sections 5.2.3 and 5.2.4, respectively. Other products of SA are not used, e.g. it is assumed that VDM domain definitions have been provided for the data descriptions of the data flows between the data transformers in the DFD. Such domain definitions can, to some extent, be automatically derived from typical SA products as data dictionaries and entity-relationship diagrams (ERDs).

Characteristics of the problem domain

Some characteristics of the problem domain (e.g. timing aspects, knowledge about the implementation model of the system) can influence the way a DFD can best be mapped onto a VDM specification. In this chapter we describe how to transform DFDs that only show the flow of data through the system, not making any assumptions on timing aspects or control flow. We call the transformation in which only the DFD (and some essential information on data types and functional aspects of the system) is taken into account a primary-mapping step in the overall transformation of informal requirements into the formal specification.

Yet, since the VDM specifications resulting from a primary-mapping step can look rather complicated, we want to keep the possibility to include characteristics of the problem domain (usually non-functional requirements imposed upon the system) in the transformation. We call this process the secondary-mapping step. As will be shown later, the secondary-mapping step can influence the readability of the formal specification to a large extent. Executing the secondary-mapping step corresponds to applying restrictions to the DFDs we want to be able to transform. The primary-mapping step and the secondary-mapping
5.2. The transformation of data flow diagrams

step are shown in figure 5.2.

Figure 5.2: The primary-mapping step and the secondary-mapping step

Mapping the various DFD constructs onto VDM constructs can be done in more than one way. More than two different mappings exist, but we will concentrate on two approaches which are opposites of each other: the PAC approach and the SAC approach; these will be discussed in sections 5.2.3 and 5.2.4, respectively. The distinction between the primary-mapping step and the secondary-mapping step is made to allow for a semantically clean description of these two approaches. In practice both steps would be integrated into one single step in which the informal aspects of the system (captured in DFDs, data dictionary and auxiliary knowledge of the problem domain) are transformed into a complete formal specification of the system.

Our work differs from the work of others in that we (in the primary-mapping step) do not assume any specific implementation model for DFDs; DFDs are envisaged as networks of connected processes through which the flow of data is shown, nothing more and nothing less. No assumptions are made on the way in which or when the data transformers execute. At the same time it is still possible to construct understandable VDM specifications, although mainly in the secondary-mapping step.

5.2.3 The PAC approach

In the PAC approach for mapping DFD constructs onto VDM constructs we make only few assumptions on how data is transported from one
data transformer to the other. We call this the \textit{P(allel) AC(ces) approach} (parallel as opposed to \textit{sequential} access, assumed in the second approach which is discussed in section 5.2.4). In the PAC approach, the primary-mapping step consists of 4 activities:

1. Create a VDM \textit{state component} for each \textit{data store} in the DFD.
2. Create a VDM \textit{operation} for each \textit{data transformer} and for each \textit{external process} in the DFD.
3. Create a VDM \textit{operation} for each (set of) \textit{data store(s)} and all the \textit{data transformers} which access that same (set of) data store(s). Replace these data store(s) and data transformers in the DFD by a single data transformer. The newly created VDM operation is a model for this data transformer.
4. Create a VDM \textit{operation} for each \textit{data flow} and the (two) \textit{data transformers} connected by the data flow. Replace each such data flow and the two data transformers in the DFD by a single data transformer. The newly created VDM operation is a model for this data transformer.

Steps 3 and 4 essentially correspond to a process of \textit{composition} (in contrast to the \textit{decomposition} process during which the context diagram was refined into a low-level DFD, containing only primitive data transformers) of data transformers into higher-level data transformers until only one data transformer remains; this data transformer corresponds to the context diagram that was used as the starting point for the SA process. The details of the steps are explained below.

\textbf{Data stores}

The first activity in the primary-mapping step in the PAC approach is the mapping of \textit{data stores} onto VDM \textit{state components}. We refrain from specifying \textit{how} access to a data store takes place nor do we make any assumption on the way in which data stores are accessed by data transformers. We will assume, however, that if two data transformers update the same data store, these updates are indivisible actions, i.e. they cannot occur simultaneously. An arrow between a data transformer and a data store means that the data transformer has access to the data store. Such an arrow is seen as a syntactical notation to make the data store visible to a data transformer, and therefore has no
5.2. The transformation of data flow diagrams

explicit VDM counterpart. If the arrow is directed from the data transformer to the data store it means that the data transformer may add objects to the data store. Write access also implies that the data store can be examined without affecting it. If the arrow is directed from the data store to the data transformer it means that the data transformer only has read access the data store, so the data transformer cannot change the data store.

Data transformers and external processes
In the PAC approach, data transformers and external processes are treated in the same way, therefore we will limit the discussion to data transformers. Since we do not know how or when a data transformer is triggered (this is part of the control flow), the only semantics we can give to a data transformer is a description of the effect of the data transformer, taking into account the presence or absence of data on all or some of its inputs. The most obvious construct in VDM-SL onto which a data transformer can be mapped is a VDM operation. We use implicit VDM operations since our main interest lies in what the effect of the transformation is, instead of how the effect is achieved which is, after all, also not directly shown by the DFD.

Inputs and outputs of a data transformer are dealt with as follows:

- **Inputs to the data transformers** are transformed into **input parameters** of the VDM operation. The input parameters of a VDM operation are typed. We determine the type of an input parameter from the information we have of the data flow connected to the data transformer (presented by the user in the form of a VDM-like data dictionary).

- **Outputs from the data transformers** are transformed into the **result parameter** of the VDM operation. A tuple expression is constructed if a data transformer has more than one output. The types of the components are determined in the same way as the types of the inputs of the data transformer.

The order in which input and output parameters are presented in the signature of the operation is left unspecified, because talking about the ‘order’ of inputs and outputs of a data transformer is not very
meaningful.

If a data transformation takes place we can neither assume that output values are generated for all the outputs of that data transformer nor that input values will be taken from all the inputs. Thus, data transformers are constructs that may show different behaviour, depending on the presence and the values of its inputs. We introduce a syntactical extension to VDM-SL by adding a special value $\tau$ to the input and output domains of operations to model this situation. We specify that this special value has been added to a domain by appending a subscript $\theta$ to the domain. So, $N_\theta$ is the notation for the domain of natural numbers extended with the special value $\tau$. The VDM-SL 'built-in' special value nil (created by the optional-domain constructor) cannot be used for this purpose due to the conflicts that can be caused by the possible use of optional domains in the data dictionary of the DFD.

**Example 5-1**
An example of a data transformer is shown in figure 5.3.

![Data transformer with two inputs and two outputs](image)

Figure 5.3: Data transformer with two inputs and two outputs

A suitable VDM operation to model this data transformer is:

$$D \left( i_1 : I_{1\theta}, i_2 : I_{2\theta} \right) o : O_{1\theta} \times O_{2\theta}$$

post let mk-(o_1, o_2) = o in

$$P_D(i_1, i_2, o_1, o_2)$$

The predicate $P_D$ depends on the *functionality* of the data transformer $D$. It cannot be derived from the DFD and must therefore be provided by the analyst/designer. The most general form of $P_D$ is the logical value *true*, which corresponds to a semantics for data transformer $D$ disregarding information on the functionality of $D$. *Any* implementation with a signature satisfying the types of the inputs $i_1$ and $i_2$ and
outputs $o_1$ and $o_2$ will satisfy the post-condition true.

To illustrate a possible further development step of a specification as general as the one above, we will assume that we know that data transformer $D$ produces an output $o_1$ for each input $i_1$ and an output $o_2$ for each input $i_2$. The corresponding VDM operation then becomes:

$$D(i_1 : I_1 \Theta, i_2 : I_2 \Theta) \circ O_1 \Theta \times O_2 \Theta$$

post let mk-(a_1, a_2) = o in

$$\left( P_{1D}(i_1, a_1) \land a_2 = \tau \right) \lor$$
$$\left( P_{2D}(i_2, a_2) \land a_1 = \tau \right)$$

The disjunction in the post-condition allows the post-condition to be satisfied by arbitrarily choosing between either the transformation for $i_1$ (in which case no output needs to be constructed for $o_2$) or the transformation for $i_2$ (in which case no output needs to be constructed for $o_1$). The ability to express looseness in VDM enables us in this case to disregard control flow.

$\Box$

**Data flows**

Data flow arrows show a connection between two data transformers or between a data transformer and an external process. *When* or *how* data flows is not considered, the meaning of a connection is the composition of the data transformers that it combines. Since a composition of two data transformers is a (higher-level) data transformer itself, and since we have chosen to map data transformers onto VDM operations, we map *data flows* onto VDM *operations* as well. The post-condition of the resulting operation must show the *existence* of the connection between the two data transformers or the data transformer and the external process.

**Example 5-2**

We start with an example (figure 5.4) in which two data transformers $D_1$ and $D_2$, each with two inputs, are connected to each other by a data flow $o_{1/D_1}$. 
The semantics of such a construct becomes clear when we consider the higher-level data transformer $D_1 \rightarrow D_2$, giving semantics to the data flow arrow. At first sight, the way to map this data transformer onto a VDM operation appears to be the following:

$$D_1 \rightarrow D_2 (i_{1D1} : I_{1D1} \theta, i_{2D1} : I_{2D1} \theta, i_{D1} : I_{D1} \theta)$$
$$o : O_{2D1} \theta \times O_{1D2} \theta \times O_{2D2} \theta$$

**post** let mk-\{o_{2D1}, o_{1D2}, o_{2D2}\} = o in

$$\exists o_{1D1} : O_{1D1} \theta \cdot post-D_1 (i_{1D1}, i_{2D1}, o_{1D1}, o_{2D1}) \land$$
$$post-D_2 (i_{D2}, o_{1D1}, o_{1D2}, o_{2D2})$$

In this specification, the output value of $D_1$ is directly used as the input value for $D_2$. Unfortunately, this specification is too strict: since we make no assumptions on the control and speed of execution of data transformers, we cannot directly connect the output of $D_1$ to the input of $D_2$. Suppose e.g. that $D_2$ receives an input value on its input $i_{D2}$ such that it can make a valid transformation, regardless of the input $o_{1D1}$. The specification above forces $D_2$ to take input $o_{1D1}$ into account as well, therefore we must seek a specification for $D_1 \rightarrow D_2$ that is more ‘loose’. We know, however, that the value of $o_{1D1}$ must once have fulfilled the post-condition of $D_1$. The VDM operation corresponding to the combination shown in figure 5.4 is:

$$D_1 \rightarrow D_2 (i_{1D1} : I_{1D1} \theta, i_{2D1} : I_{2D1} \theta, i_{D1} : I_{D1} \theta)$$
$$o : O_{2D1} \theta \times O_{1D2} \theta \times O_{2D2} \theta$$
5.2. The transformation of data flow diagrams

\[ \text{post let } mk-(o_{D1}, o_{D2}, \alpha_{D2}) = o \text{ in} \]
\[ (\exists o_{D1}; O_{D1}, \alpha_{D2}; O_{D2}, \alpha_{D2}) \land \]
\[ (\exists o_{D1}; O_{D1}, \alpha_{D2}; O_{D2}, \alpha_{D2}) \land \]
\[ \text{post-}D_1(i_{D1}; D_{D1}, o_{D1}, \alpha_{D1}) \land \]
\[ \text{post-}D_1(i_{D1}; D_{D1}, o_{D1}, \alpha_{D1}) \land \]
\[ \text{post-}D_2(i_{D2}; D_{D2}, o_{D2}, \alpha_{D2}) \land \]
\[ \text{post-}D_2(i_{D2}; D_{D2}, o_{D2}, \alpha_{D2}) \land \]

The post-condition of this operation can thus be satisfied by creating an \(o_{D1}\) which satisfies the post-condition of \(D_1\) (in effect we execute \(D_1\) to create such an \(o_{D1}\), which can then be used to satisfy the post-condition of \(D_2\). The total effect of \(D_1 To D_2\) is the execution of both \(D_1\) and \(D_2\), denoted by the conjunction between the two quantified expressions. The specification of more inputs and/or outputs follows the same line.

Example 5.3
A slightly more complicated example is shown in figure 5.5.

![Figure 5.5: Fragment of a DFD containing a loop](image)

The corresponding VDM operation is:

\[ D_1 To D_2 (i_{D1}; I_{D1}, o_{D2}; O_{D2}) \land \]
\[ \text{post} (\exists o_{D1}, o'_{D1}; O_{D1}, o_{D2}, O_{D2}) \land \]
\[ \text{post-}D_1(i_{D2}; D_{D2}, o_{D1}, \alpha_{D1}) \land \]
\[ \text{post-}D_2(i_{D2}; D_{D2}, o_{D1}, \alpha_{D1}) \land \]

This example shows how we deal with 'recursive' data transformers. Existential quantification is used to generate the data flowing between \(D_1\) and \(D_2\).
The order in which the combinations of data transformers are made can in the secondary-mapping step significantly influence the readability of the specification, and is therefore an important aspect of a combined SA/VDM method. In the primary-mapping step, however, this order is irrelevant, which increases the belief that the PAC approach is a consistent approach for mapping DFD constructs onto VDM constructs.

**Data transformers in combination with data stores**
Combinations of data transformers and data stores are modeled as operations accessing the state of the system.

**Example 5-4**
An example of a data transformer in combination with a data store is shown in figure 5.6.

![Data transformer diagram](image)

Figure 5.6: Access to a data store

The corresponding VDM specification is:

\[
D (i : I_\Theta) \circ : O_\Theta
\]

\[
\text{ext wr } ds : DS
\]

\[
\text{post } P_\rho(i, o, ds, \overline{ds})
\]

**Example 5-5**
Combining data transformers accessing the same data store is more
complicated. Consider the configuration in figure 5.7.

![Diagram of two data transformers and one data store]

Figure 5.7: Combination of two data transformers and one data store

We know that both $D_1$ and $D_2$ can or will access data store $ds$, but we cannot make any assumption on the order in which these accesses take place. We are able to enumerate the possibilities, however. In this case we only have two possibilities: either $D_1$ uses the data store first and then $D_2$, or the other way around. $D_1$ To $D_2$ must take both situations into account, which is achieved in the following specification:

$$D_1 \text{ To } D_2 (i_{D1} : I_{D1\theta}, i_{D2} : I_{D2\theta}) \text{ o } D_2 : O_{D2\theta}$$

ext wr $ds : DS$

post $\exists ds_{D1\theta}, ds_{D1n}, ds_{D2\theta}, ds_{D2n}, ds_d : DS$. 

$$(\exists o_{D1} : O_{D1\theta} \cdot \text{post-} D_1(i_{D1}, o_{D1}, ds_{D1n}, ds_{D1\theta})) \land$$

$$(\exists o_{D1} : O_{D1\theta} \cdot i_{D1} : I_{D1\theta} \cdot$$

$\text{post-} D_1(i_{D1}, o_{D1}, ds_{D2\theta}, ds_d) \land$$

$\text{post-} D_2(o_{D1}, i_{D2}, o_{D2}, ds_{D2n}, ds_{D2\theta})) \land$$

$$(ds = ds_{D1n} \land \overline{ds} = ds_{D2\theta} \land ds_{D1\theta} = ds_{D2n}) \lor$$

$$(ds = ds_{D2n} \land \overline{ds} = ds_{D1\theta} \land ds_{D1n} = ds_{D2\theta})$$

In the specification of this operation the fact is used that during the transformation three synchronization points can be distinguished: the first synchronization point is the point before the transformation (char-
acterized by \( \overline{ds} \), the second synchronization point is either after \( D_1 \) has accessed data store \( ds \) or after \( D_2 \) has accessed data store \( ds \) (since access to a data store was characterized as an indivisible action), and the last synchronization point is after the transformation performed by the data transformer (characterized by \( ds \)). To characterize the second synchronization point we introduce an intermediate value \( ds_{dc} \) for the state in the post-condition. The effect of the transformation can then be characterized by specifying that either \( ds_{dc} \) is produced by \( D_1 \) and then used as an input value for \( D_2 \), or vice versa.

Because of the ordering problem, all data transformers that use the same data store (or even the same set of data stores!) have to be combined in a single data transformer. The more data transformers use the same data store, the more complicated the specification for the combined data transformer will be.

5.2.4 The SAC approach

In the second approach that we present, called the \( S(\text{equential})\ AC(\text{ces}) \) approach, we use the fact that a data flow is a channel which can be used as an input or as an output by a data transformer, as a starting point. As in the PAC approach, the primary-mapping step in the SAC approach consists of a number of steps:

1. Map each data store onto a VDM state component.
2. For each data flow in the DFD, create a state component the type of which is a sequence of the elements that are transported by that data flow.
3. For each state component modeling a data flow, create two VDM operations which can be used to add and remove elements from those state components.
4. Create a VDM operation for each data transformer and for each external process in the DFD.

The details of these steps are explained below.

Data stores

Data stores are treated in the same way as in the PAC approach, i.e.
they are mapped onto state components.

**Data flows**

A data flow arrow is viewed as an arbitrary large (but finite) channel (queue), receiving input on one side (the tail of the data flow arrow) and providing output on the other side (the head of the data flow arrow). A data transformer can provide output to a data flow or receive input from a data flow, depending on the direction of the connection. However, a data transformer cannot just arbitrarily access any data element that is currently in one of its input channels, it can only access the first such data element.

Since in this approach, data flow arrows are explicitly regarded as static elements which can hold data and deliver them on demand, the most obvious way to represent them is as part of the state of the specification. The order of the elements that flow through the channel is important, therefore we will model data flows as sequences of the type of the data elements. We also associate access operations with these state components to ensure their correct use.

**Example 5-6**

Consider again the data flow arrow in figure 5.1. The corresponding component of the state, including operations for accessing this component, is:

\[
\text{state ... of} \\
\quad \ldots \ldots \\
\quad df : DF^* \\
\quad \ldots \ldots \\
\text{end}
\]

\[
\begin{align*}
\text{Put} & \ df (i : DF) \\
\text{ext wr} & \ df : DF^* \\
\text{post} & \ df = \overline{df} \cap [i]
\end{align*}
\]
\[ \text{Get-df ()} r : DF_\theta \]
\[ \text{ext wr } df : DF^* \]
\[ \text{post if } \overline{df} = [] \]
\[ \text{then } r = \tau \land df = [] \]
\[ \text{else } \overline{df} = [r] \cap df \]

Note that for the specification of \text{Get-df} it is again necessary to use the syntactical $\Theta$ extension.

---

**Data transformers and external processes**

Data transformers and external processes are modeled as VDM operations, just as in the PAC approach. The difference is that now they have neither input nor result parameters, because the state elements implementing the data flows can be accessed directly.

---

**Example 5-7**

Consider again the data transformer shown in figure 5.3. The VDM counterpart\(^{15}\) of this data transformer, under the first interpretation used in example 5-1, is:

\[ D () \]
\[ \text{ext wr } i_1 : I_1^* \]
\[ \text{wr } i_2 : I_2^* \]
\[ \text{wr } o_1 : O_1^* \]
\[ \text{wr } o_2 : O_2^* \]
\[ \text{post } \exists i'_1 : I_1, i'_2 : I_2, o'_1 : O_1, o'_2 : O_2 : \]
\[ (\text{post-Get-}i_1(i'_1, i_1, \overline{i_1}) \land \text{post-Get-}i_2(i'_2, i_2, \overline{i_2}) \land \]
\[ \text{post-Put-}o_1(o'_1, o_1, \overline{o_1}) \land \text{post-Put-}o_2(o'_2, o_2, \overline{o_2}) \land \]
\[ P_D(i'_1, i'_2, o'_1, o'_2)) \]

The data flows $i_1$, $i_2$, $o_1$ and $o_2$ are modeled as state components, made visible to the operation through an external clause. In the post-

---

15. The definitions of the \text{Put} and \text{Get} operations are not explicitly given in the examples of the SAC approach.
condition single elements for input and output are created by using quantified expressions. These single elements are then used as parameters to the \textit{Get} and \textit{Put} operations and to the predicate $P_D$ describing the effect of the data transformer on different input values. The binary expressions stating that new state values should be the same as the old values (e.g. $i_2 = \overline{i_2}$) are necessary to ensure that when one specific input value is transformed, the other input/output pair remains unaffected.

As in example 5-1, assume that we have some additional knowledge: $D$ will produce an $o_1$ if $i_1$ is present on its input, and an $o_2$ if an $i_2$ is present. Following the SAC approach the VDM operation then becomes:

\[
D() \\
\text{ext wr } i_1 : I_1^* \\
\quad \text{wr } i_2 : I_2^* \\
\quad \text{wr } o_1 : O_1^* \\
\quad \text{wr } o_2 : O_2^* \\
\text{post } \exists i : I_{1\theta} \mid I_{2\theta}, o : O_{1\theta} \mid O_{2\theta} : \\
\quad (\text{post-Get-}i_1(i, \overline{i_1}) \land \text{post-Put-}o_1(o, o_1, \overline{o_1}) \land \\
\quad i_2 = \overline{i_2} \land o_2 = \overline{o_2} \land P_{1D}(i, o)) \lor \\
\quad (\text{post-Get-}i_2(i, \overline{i_2}) \land \text{post-Put-}o_2(o, o_2, \overline{o_2}) \land \\
\quad i_1 = \overline{i_1} \land o_1 = \overline{o_1} \land P_{2D}(i, o))
\]

As in the PAC approach, looseness (as implied by the disjunction between the two possibilities) is used to leave the choice which transformation is done, which is after all part of the control flow of the system, unspecified. 

\[\square\]

**Example 5-8**
Consider the combination of two data transformers that was used in example 5-3. The VDM specification for this combination following the SAC approach is:
state . . . of

... : ...  
\( i_{p1} : I_{p1}^* \)  
\( o_{p1} : O_{p1}^* \)  
\( o_{1D2} : \overline{O_{1D2}}^* \)  
\( o_{2D2} : \overline{O_{2D2}}^* \)  
... : ...  
end

\( D_1() \)

ext wr \( i_{p1} : I_{p1}^* \)  
wr \( o_{1D2} : \overline{O_{1D2}}^* \)  
wr \( o_{p1} : O_{p1}^* \)  
post \( \exists i'_{p1} : I_{p1}^*, o'_{p1} : O_{p1}^*, o'_{1D2} : \overline{O_{1D2}}^* \)  
\( \text{post-Get-}i_{p1}(i'_{p1}, i_{p1}, i_{D1}) \wedge \)  
\( \text{post-Get-}o_{1D2}(o'_{1D2}, o_{1D2}, \overline{o_{1D2}}) \wedge \)  
\( \text{post-Put-}o_{p1}(o'_{p1}, o_{p1}, \overline{o_{p1}}) \wedge \)  
\( P_{p1}(i'_{p1}, o'_{1D2}, o'_{p1}) \)

\( D_2() \)

dex wr \( o_{p1} : O_{p1}^* \)  
wr \( o_{1D2} : \overline{O_{1D2}}^* \)  
wr \( o_{2D2} : \overline{O_{2D2}}^* \)  
post \( \exists o'_{p1} : O_{p1}^*, o'_{1D2} : \overline{O_{1D2}}^*, o'_{2D2} : \overline{O_{2D2}}^* \)  
\( \text{post-Get-}o_{p1}(o'_{p1}, o_{p1}, \overline{o_{p1}}) \wedge \)  
\( \text{post-Put-}o_{1D2}(o'_{1D2}, o_{1D2}, \overline{o_{1D2}}) \wedge \)  
\( \text{post-Put-}o_{2D2}(o'_{2D2}, o_{2D2}, \overline{o_{2D2}}) \wedge \)  
\( P_{p2}(o'_{p1}, o'_{1D2}, o'_{2D2}) \)

This VDM specification as a whole is a specification for the higher-level DFD comprising the combination of \( D_1 \) and \( D_2 \).

\[\square\]

5.2.5 Example: a simple spelling checker

To illustrate the PAC approach and the SAC approach we will show how a fragment from a DFD for a simple spelling checker can be trans-
formed into VDM. The problem is used in [Sommerville 82] to explain
the notions of data flow diagrams and structure charts. The component
which we will use to illustrate both approaches will expect a \texttt{file\_name},
and checks whether a file having this name is present in a file system
\texttt{docs}. An indication \texttt{file\_not\_found} for the absence or presence of the file
is returned. If the file is present it is split into a sequence of words (each
of which will be checked against a dictionary by another component).
The DFD for this component is shown in figure 5.8.

![Diagram](image)

Figure 5.8: Fragment of the DFD for the spelling checker

The following VDM data types are used:

\[
\begin{align*}
\text{Docs} & = \text{FileName} \rightarrow^{m} \text{File} \\
\text{FileName} & = \text{token} \\
\text{File} & = \text{char}^* \\
\text{File\_NotFound} & = \mathbb{B} \\
\text{WordList} & = \text{Word}^* \\
\text{Word} & = \text{char}^+
\end{align*}
\]

The spelling checker component according to the PAC approach

In the primary-mapping step, the following VDM specification can be
derived for the spelling checker component following the PAC approach:

\[
\text{state SpellingChecker of} \\
\quad \text{docs : Docs} \\
\text{end}
\]
GetFile (file_name : FileNameθ) r : FileNotFoundExceptionθ × Fileθ
ext rd docs : Docs
post let mk-(file_not_found, file) = r in
  if file_name ∈ dom docs
  then file_not_found = false ∧ file = docs(file_name)
  else file_not_found = true ∧ file = τ

Split (file : Fileθ) word_list : WordListθ
post if file = τ
  then word_list = τ
  else word_list = [file (i, ..., j) | i, j ∈ inds file ·
                    i ≤ j ∧ ¬ ∃ k ∈ {i, ..., j} · file(k) = ’’]

GetFileToSplit (file_name : FileNameθ)
  r : FileNotFoundExceptionθ × WordListθ
ext rd docs : Docs
post let mk-(file_not_found, word_list) = r in
  ∃ file : Fileθ ·
  post-GetFile(file_name, mk-(file_not_found, file), docs) ∧
  post-GetFile(file_name, file_name : FileNameθ, file_not_found : FileNameθ, docs) ∧
  post-Split(file, word_list)

The most apparent aspect of the specification is the complexity of the operation GetFileToSplit. The complexity is due to the looseness of the operation. When we take non-functional aspects of the problem into account, it soon becomes clear that the specification can be significantly improved in the secondary-mapping step. Suppose e.g. that we use the knowledge that the system is a sequential system. This means that timing assumptions can be made about the connection between GetFile and Split, resulting in the elimination of the looseness in GetFileToSplit.

Another optimization results from an analysis of the data types used. The θ extensions can be eliminated by changing the domain definitions such that special values are introduced the model the τ value. For the
domain $File$ the optional-domain constructor can be used because that
constructor has not already been used in the definition of the domain
$File$. For the domain $WordList$, the special value ‘[]’ can be used to
model the value $\tau$, since the empty sequence will not in any other
situation be used for that domain. For all the other domains used, the
$\Theta$ extensions can simply be removed, because the value $\tau$ is not used
for those domains in the specification. If no suitable replacements for
$\tau$ can be found, it is always possible to eliminate the $\Theta$ extensions
by constructing a union of the original domain with a special value
denoted as a quote literal.

The specifications of $GetFile$, $Split$ and $GetFileToSplit$ can now be
changed into:

$GetFile\ (file\_name : FileName)\ r : FileNotFound \times [File]$

ext rd docs : Docs

post let mk-(file\_not\_found, file) = r in
  if file\_name \in dom docs
  then file\_not\_found = false \land file = docs(file\_name )
  else file\_not\_found = true \land file = nil

$Split\ (file : [File])\ word\_list : WordList$

post if file = nil
  then word\_list = []
  else word\_list = \{file (i, \ldots, j) | i, j \in \text{inds} file \cdot
  i \leq j \land \exists k \in \{i, \ldots, j\} \cdot file(k) = ''\}

$GetFileToSplit\ (file\_name : FileName)\ r : FileNotFound \times WordList$

ext rd docs : Docs

post let mk-(file\_not\_found, word\_list) = r in
  \exists file : [File].
  post-GetFile(file\_name, mk-(file\_not\_found, file), docs) \land
  post-Split(file, word\_list)

The specification resulting from the secondary-mapping step is easier
to understand and more elegant than the specification resulting from
the primary-mapping step.
The spelling checker component according to the SAC approach

A VDM specification constructed for the spelling checker component in the primary-mapping step, following the SAC approach, is as follows:

```vdm
state SpellingChecker of
  docs : Docs
  file_name : FileName*
  file_not_found : FileNotFound*
  file : File*
  word_list : WordList*
end

GetFile ()
extrd docs : Docs
  wr file_name : FileName*
  wr file : File*
  wr file_not_found : FileNotFound*
post let file_name' : FileName be st
  post-Get-file_name(file_name', file_name, file_name) in
  if file_name' = \tau
    then file_not_found = file_not_found \land file = file
  else if file_name' \in dom docs
    then post-Put-file_not_found(false, file_not_found, file_not_found) \land
      post-Put-file(docs(file_name'), files, files)
    else post-Put-file_not_found(true, file_not_found, file_not_found) \land
      file = file

Split ()
ext wr file : File*
  wr word_list : WordList*
```
post let file' : File<∅> be st post-Get-file(file', file, file) in 
   if file' = τ 
   then word_list = word_list 
   else let word_list' = 
      [file'(i, ..., j) | i, j ∈ inds file' . 
       i ≤ j ∧ ∃ k ∈ {i, ..., j} : file'(k) = ''] in 
      post-Put-word_list(word_list', word_list, word_list) 

The external clause in this specification shows the state components 
to which the operation has read- and/or write access. Unfortunately, it 
is possible to change the state components modeling the various data 
flows without using the appropriate Put and Get operations due to the 
lack of data hiding facilities in VDM.

VDM requires that if a state component remains unchanged, this must 
be explicitly marked as such in the post-condition. This is the case for 
GetFile: if a file having the user-supplied file name is not present in the 
file system, then the file data flow remains unchanged. If we ‘forget’ to 
specify that a specific state component remains unchanged, then the 
implementation of that operation is allowed to change the value of that 
state component into any (legal) value because every such value will 
satisfy the post-condition.

As in the PAC approach, during the secondary-mapping step a few 
improvements can be made. The elimination of Θ extensions in this 
case can be realized by the observation that, since we had assumed 
that we were modeling a sequential system, both operations cannot do 
anything useful unless input values are present on their inputs. The τ 
values can then be modeled as empty sequences in combination with 
pre-conditions for the operations. The specifications for GetFile and 
Split then become:

GetFile ()

ext rd docs : Docs
   wr file_name : FileName*
   wr file : File*
   wr file_not_found : FileNotFound*
pre file.name ≠ []

post let file.name' : FileName be st
    post-Get-file.name(file.name', file.name, \text{file.name}) in
    if file.name' ∈ dom docs
    then post-Put-file.not.found (false, file.not.found, \text{file.not.found}) ∧
    post-Put-file(docs(file.name'), files, \text{files})
    else post-Put-file.not.found (true, file.not.found, \text{file.not.found}) ∧

    file = file

\(\text{Split}()\)

ext wr file : File*"n
    wr word_list : WordList*"

pre file ≠ []

post let file' : File be st post-Get-file(file', file, \text{file}) in
    let word_list' =
        \{ file'(i, \ldots, j) \mid i, j ∈ \text{inds file}' \cdot
        i ≤ j ∧ \nexists k ∈ \{i, \ldots, j\} \cdot file'(k) = '' \} in
    post-Put-word_list(word_list', word_list, word_list)"

The specification resulting from the secondary-mapping step is easier to understand and more elegant than the specification resulting from the primary-mapping step.

**The spelling checker component directly specified**

A VDM specification for the spelling checker component written directly from analyzing the problem description could be as follows:

```verbatim
state SpellingChecker of
    docs : Docs
end
```
5.2. The transformation of data flow diagrams

\[
\text{GetFileToSplit (file\_name : FileName) } r : \text{FileNotFoundException } \times [\text{File}]
\]

**ext rd docs : Docs**

**post let mk- (file\_not\_found, word\_list) = r in**

**let file\_not\_found = (file\_name \notin dom docs) in**

**if file\_not\_found**

**then r = mk- (true, nil)**

**else let file = docs(file\_name) in**

**let word\_list =**

\[
[file(i, ..., j) | i, j \in \text{inds file} : i \leq j \land \exists k \in \{i, ..., j\} . file(k) = ''] \text{ in}
\]

**r = mk- (file\_not\_found, word\_list)**

Since no explicit decomposition step was made (due to the simplicity of the problem), the above specification is shorter than the ones constructed following the PAC approach or the SAC approach. This doesn’t necessarily make the specification more understandable, but it does make it better manageable.

The quoting of post-conditions, which is present in the specifications constructed following the PAC or the SAC approach, is absent in the specification written from scratch. This is unfortunate, because the quotation mechanism provides handles for a further development of the specification into an implementation.

5.2.6 A comparison of the PAC approach and the SAC approach

Both the PAC approach and the SAC approach provide a sound base for the complete and consistent modeling of DFDs. A significant difference between the two approaches is that in the PAC approach it is assumed that a data flow only expresses the existence of a (directed) connection between two data transformers, whereas in the SAC approach it is assumed that data flows are sequential channels which communicate data between two data transformers.

The PAC approach can perhaps best be seen as a means to provide a formal semantics for DFDs (although to validate this statement more
characteristics of the approach should be examined) because it closely models the intuitive semantics most system analysts give to DFDs. The disadvantage is that its use results in rather complex specifications; the ‘loose’ interpretation of data flows considerably complicates the post-condition associated with the operation modeling that data flow. This becomes even worse when we take the influence of data stores into account: as we have seen in example 5-5, even the specification for only two data transformers accessing the same data store is hard to understand, and when the number of data stores involved increases, this complexity increases as well. The complexity of a combination of data transformers can be decreased by using explicit definitions instead of the implicit ones shown in this chapter. Such definitions, however, are less abstract than their implicit counterparts.

The SAC approach provides a more operational approach for transforming DFDs, and therefore seems to be a practical base for software development. A minor technical problem with the use of VDM for the specification of the result of the transformation is that, although we assume the presence of Put and Get operations, it is possible to violate the rule of sequential access by directly accessing the data flow through sequence indexing, and it is possible to use an input data flow as an output data flow. Consistency checking of DFDs when following the SAC approach is, therefore, more complicated. When following the PAC approach the consistency of the DFD is largely induced by the syntax of VDM-SL. Furthermore, the specification of all Put and Get operations themselves forms a significant part of the total specification.

A second minor problem appearing in specifications when following the SAC approach is that even if a specific state component is not used in the description of the effect of the transformation, we still have to specify that fact explicitly. In such a case we would specify that the state component has not been changed during the transformation. This leads to rather lengthy post-conditions as illustrated by the spelling checker example in section 5.2.5.

By applying restrictions to the DFDs better specifications can be obtained. In the spelling checker example, we saw that under the assumption that we were modeling a sequential system, the specifications re-
sulting from following both the PAC approach and the SAC approach could be significantly improved. Furthermore, the \( \Theta \) extensions could be replaced by the standard VDM optional-domain constructor, or by other ‘special’ values.

Despite the different character of the two models, both provide consistent mappings from DFDs to VDM constructs.

5.3 Methodological aspects of combining SA and VDM

Having considered two possibilities for mapping DFDs to VDM specifications, we are now in the position to describe a method in which aspects of SA and VDM are combined. The method is called Structured VDM (SVDM). 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 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 is possible to check the DFD for any syntactic and semantic inconsistencies.
5.3.1 Principles of SVDM

SVDM is based on the following principles:

- During different phases of development, different aspects of the problem domain should be dealt with; '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 chapter, but can be found in [Larsen 93a].

There is no specific reason why this approach should be limited to using SA and VDM; such a combined method can be expanded by constructing 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.

5.3.2 Overview of SVDM

The paradigm used for 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.\textsuperscript{16} 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\textsuperscript{17} 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. Also it is now possible to (automatically) generate a level 2 structure chart description.

\textsuperscript{16} The various 'levels' we distinguish refer to the different stages of development of the design products.

\textsuperscript{17} For an explanation of the symbols used in structure charts we refer to [Constantine 79].
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 5.9.

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 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 step in more detail. We will illustrate some of the steps by means of the same example as used in section 5.2.5, i.e. the simple spelling checker.

**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.
**Figure 5.9: An overview of SVDM**

**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 data transformer 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 hierarchy of DFDs.

**Example: the spelling checker component**
The hierarchy of DFDs for this component after analyzing the problem, constructing and decomposing the context diagram is shown in figure 5.10. Whereas in the previous section we illustrated the PAC and SAC approach with the combination of *GetFile* and *Sort*, for the methodological aspects we will focus on *Split* and *Sort*.

![Diagram](image)

*Figure 5.10: Two-level hierarchy DFD for SplitToSort*

**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.
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. These type definitions were already given in section 5.2.5; for this part of the spelling checker we need only one additional type definition for a type SortedWordList:

types
  SortedWordList = Word*

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

values

  WordSeparator : char is not yet defined

functions

  LexicallySmaller : Word \times Word \rightarrow \mathbb{B}

  \text{LexicallySmaller} (-,-) \triangleq

  \text{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.

In our combined approach we will use a transformation based on the
PAC approach\textsuperscript{18}, as described in the previous section, with some minor adjustments. 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 section 5.2.

**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.

\begin{verbatim}
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-Split(file_name, docs) \land
       \exists file_name' : FileName, docs' : Docs, word_list : WordList .
       pre-Split(file_name', docs') \land
       post-Split(file_name', word_list, docs') \land
       pre-Sort(word_list)
\end{verbatim}

\textsuperscript{18} Notice that the distinction made in the previous section between a primary and a secondary mapping step plays no role in this process.
post \exists \text{word\_list} : \text{WordList} . 
pre\text{-}\text{Split}(\text{file\_name}, \text{docs}) \land 
post\text{-}\text{Split}(\text{file\_name}, \text{word\_list}, \text{docs}) \land 
\exists \text{file\_name}' : \text{FileName}, \text{docs}' : \text{Docs}, \text{word\_list} : \text{WordList} . 
pre\text{-}\text{Split}(\text{file\_name}', \text{docs}') \land 
post\text{-}\text{Split}(\text{file\_name}', \text{word\_list}, \text{docs}') \land 
pre\text{-}\text{Sort}(\text{word\_list}) \land 
post\text{-}\text{Sort}(\text{word\_list}, \text{sorted\_word\_list})

At this stage, the post-conditions of \text{Split} and \text{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 \text{Split} and \text{Sort}, hence their respective post-conditions are specified as ‘true’.

The higher-level data transformer in Figure 5.10 is in the VDM specification represented by the operation \text{SplitToSort}. The VDM operation \text{SplitToSort}, which defines the relation between \text{Split} and \text{Sort}, must express the looseness present in the DFD notation. This is achieved by using quantified expressions ‘introducing’ values which will satisfy the pre- and post-conditions of \text{Split} and \text{Sort}.

**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.

To avoid over-specification we recommend to use implicit definitions; in this way it can be expressed \textit{what} should be done, not \textit{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*.

```plaintext
operations

Split (file_name : FileName) word_list : WordList
ext rd docs : Docs
post if file_name ∉ 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 ∃ word_list : WordList ·
        post-Split(file_name, word_list, docs) ∧
    ∃ file_name' : FileName, docs' : Docs, word_list : WordList ·
        post-Split(file_name', word_list, docs') ∧
        post-Sort(word_list, sorted_word_list)
```
functions

\[ \text{ChangeIntoWords} : \text{File} \rightarrow \text{WordList} \]
\[ \text{ChangeIntoWords}(\text{file}) \triangleq \]
\[ (\exists k \in \{i, \ldots, \text{len file}\} \cdot \neg \exists k \in \{i, \ldots, j\} \cdot \)
\[ \text{file}(k) = \text{WordSeparator} \]
\[ | i \in \text{inds file} \cdot \text{file}(i) \neq \text{WordSeparator} \land \]
\[ (i = 1 \lor \text{file}(i - 1) = \text{WordSeparator})] \]

These mini-specifications for \textit{Split} and \textit{Sort} replace the original post-
conditions. Since neither \textit{Split} nor \textit{Sort} has a pre-condition, the calls

to \textit{pre-Split} and \textit{pre-Sort} can be removed from \textit{SplitToSort}.

**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 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.

**Generating the level 2 VDM specification**

At this point it is possible to (automatically) generate a complete, ab-
stract (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 point the designer can present the
specification in a readable way using two different views.

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

The spelling checker is 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:

\[
\text{SplitToSort (file\_name : FileName) sorted\_word\_list : SortedWordList}\\
\text{exit docs : Docs}\\
\text{post } \exists \text{ word\_list : WordList} .\\
\text{post-Split(file\_name, word\_list, docs) } \land \text{post-Sort(word\_list, sorted\_word\_list)}
\]

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.

**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 5.11.

**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 91]), 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.
Figure 5.11: Level 1 structure charts for SplitToSort

Generating the level 3 VDM specification
It is now possible to automatically generate a textual representation of
the design as 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 of 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:

operations

\[ Split : FileName \rightarrow WordList \]

\[ Split (file\_name) \triangleq \]

\[ \text{if } file\_name \notin \text{dom docs} \]

\[ \text{then return } [] \]

\[ \text{else let } file = docs(file\_name) \text{ in} \]

\[ \text{return ChangeIntoWords(file);} \]
Sort : WordList \rightarrow SortedWordList

Sort (word_list) \triangleq
(dcl sorted_word_list : SortedWordList := [];
dcl word_list' : WordList := word_list;
while word_list' \neq []
do let word : Word be st
   \forall i \in \text{inds } word_list'.
      \text{LexicallySmaller}(word, word_list'(i)) in
      (sorted_word_list := sorted_word_list \cup [word];
      word_list' := [word_list'(i) | i \in \text{inds } word_list' \cdot word_list'(i) \neq word]);
return sorted_word_list);

SplitToSort : FileName \rightarrow SortedWordList

SplitToSort (file_name) \triangleq
(dcl word_list : WordList := Split(file_name);
dcl 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 5.12.
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.
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 intro-
ducing 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 $\rightarrow$ ()
    Split (file_name) $\triangleq$
        if file_name $\notin$ dom docs
        then buffer := []
        else def file = docs(file_name);
            buffer := ChangeIntoWords(file);

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

SplitToSort (file_name) $\triangleleft$

  (dcl word : Word;
   dcl sorted_word_list : SortedWordList;
   Split(file_name);
   while buffer $\neq []$
   do (word := Sort();
       sorted_word_list := sorted_word_list $\cap$ [word]);
   return sorted_word_list )

The level 3 structure charts for the spelling checker component are shown in figure 5.13.

![Structure chart for SplitToSort](image)

Figure 5.13: Level 3 structure charts for SplitToSort

5.3.3 Summary of 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 5.14 a short summary of the design products is given.

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

Figure 5.14: An overview of the design products of SVDM

5.4 Conclusions

In this chapter the combined structured/formal method SVDM was introduced, based on the PAC approach relationship between DFDs and VDM specifications.
5.4. Conclusions

With respect to the methodological aspects of SVDM no hard conclusions can be drawn before the method has been used for some larger examples. A first step in that direction is made in chapter 6, which contains a case study in which SVDM is applied for the specification and development of a simple lift system.

With respect to the underlying basis of SVDM, i.e. the relationship as defined between DFDs and VDM specifications, we have shown that it is possible to define a number of different relationships, two of which were discussed in this chapter (the SAC and the PAC approach), including a comparison between the two. Each relationship can be regarded as an interpretation of DFDs, thus defining a semantics for DFDs. Therefore, by accepting a specific relation, teams of designers are forced to have the same interpretation of the DFDs they design, in this way decreasing the ambiguity of their products.

The type of DFDs which we discussed in this chapter is rather simple in the sense that we allowed only four kinds of building blocks as the components of a DFD. Because no further constraints are put on the DFDs built in this way, all kinds of sequential systems can be modeled, and this makes our approach quite powerful. Extensions to DFDs which introduce new concepts – e.g. the notion of time as introduced in SA/RT [Ward 81] – can impose problems, however, because this means that a similar concept must be present in VDM-SL to be able to map that concept to a construct in a VDM specification. If this is not so, two approaches can be followed:

1. a different formalism (instead of VDM-SL) can be used which caters for the representation of the extension.
2. the relationship can be defined in a partial manner, i.e. the sequential aspects of the DFD are modeled using VDM-SL, and for mapping the extension a different formalism is used. This requires, however, and additional step in the development process during which the VDM specification and the specification of the newly introduced formalism are merged.

The latter approach is also illustrated in the next chapter.
The application of VDM in combination with SA
In the previous chapter we discussed technical aspects of the combined structured/formal method SVDM, and we illustrated the approach by means of a small example. The aim of this chapter is to evaluate SVDM in the context of a complete development (i.e. starting from a set of informal system requirements and ending with implementation) of a somewhat larger example, in this way making the strengths and weaknesses of combined structured/formal methods more clear than can be done with small examples alone. Furthermore, since most software systems nowadays are not purely sequential – the kind of systems for which SVDM is primarily intended – we decided to use a concurrent system as our case study, in this way making it possible to evaluate SVDM as a ‘partial’ method used in combination with other techniques dealing with the concurrent aspects of the system.

For reasons of space and also because we do not consider all specifications that are produced when using SVDM equally interesting (or easy to produce without appropriate tool support) we limit ourselves to non-secondary specifications. Furthermore, after the initial steps of SVDM we also limit ourselves to the development of a (non-trivial) component
of the case study.

For the evaluation of the case study a number of criteria are needed; the criteria used for this evaluation are based on the ones also used in [Ventouris 92], where they are used for the comparison of various formal specification techniques for abstract data types. The criteria are general enough to be useful in the more general setting of this case study as well. They are:

1. **Ease of construction of the specifications.** This is a significant factor that affects the applicability of any method in practical software development. If a method requires extended resources than it has little practical value.

2. **Comprehensibility of the specifications.** A comprehensive approach allows the designer to reconstruct without major difficulties the information put down in the specifications.

3. **Completeness of the specifications.** This criterion addresses the degree to which the specifications describe all relevant aspects of the problem under development.

4. **Usefulness of the specifications.** This criterion addresses the usefulness of the specifications to construct other (lower-level) specifications or the implementation of the system. If no guidelines exist how specifications should be used, then their practical value is rather limited.

### 6.1 Introduction of the case study

The case study that was chosen to illustrate the use of SVDM is the specification and development of a *lift system*. The lift system is a small, but non-trivial example of an embedded computer system. The problem\(^{19}\) is described as follows\(^{20}\) [Spec. and Design 87]:

An *n* lift system is to be installed in a building with *m* floors. The lifts and the control mechanism are supplied by

\(^{19}\) The lift problem was originally formulated by N. Davis from STCIDECE Ltd.

\(^{20}\) The requirements of the lift system are incomplete. We will rely on our intuitive interpretation of what a lift should do in situations in which the stated requirements are not adequate.
the manufacturer. The internal mechanisms of these are assumed (given). The problem concerns the logic to move lifts between floors according to the following constraints:

- Each lift has a set of buttons, one for each floor. These are illuminated when pressed and cause the lift to visit the corresponding floor (R1). The illumination is cancelled when the corresponding floor is visited by the lift (R2).

- Each floor has two buttons (except ground and top floor), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed (R3). The illumination is cancelled when a lift visits the floor (R4). A lift is either moving in the desired direction, or has no outstanding requests. In the latter case, if both floor request buttons are pressed, only one should be cancelled (R5). The algorithm to decide which to service first should minimize the waiting time for both requests (R6).

- When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests (R7).

- All requests for lifts from floors must be serviced eventually, with all floors given equal priority (R8).

- All requests for floors from within lifts must be serviced eventually, with all floors being served in the direction of travel (R9).

- Each lift has an emergency button which, when pressed, causes a warning signal to be sent to the site manager. The lift is then deemed 'out of service'. Each lift has a mechanism to cancel its 'out of service' status (R10).

Analyzing the informal problem statement, it soon becomes apparent that the problem at hand has a non-trivial functional component (the scheduling of the various lifts in the system) but also that the system as a whole is inherently concurrent. The concurrent aspects of the system are primarily that request for lifts and requests from within lifts

---

21. The label identifies the requirement, and is used later on in this chapter to refer back to the requirement.
can be made simultaneously and have to be dealt with accordingly; the control of the various components of the lift system is, therefore, a concurrent component. Since SVDM can only be used for the development of the functional aspects of the system, another technique must be selected to deal with the concurrent aspects of the system. In this case study we choose Petri nets [Peterson 77, Agerwala 79, Brauer 82] for this purpose. Petri nets are a well-defined formalism in which (concurrent) transitions of a system can be represented; as such they are suitable to specify multiple threads of control in a system, where specific transitions can be associated with the execution of actions. The choice for Petri nets is in a sense rather arbitrary; other alternatives are available for the same purpose. The motivation for choosing Petri nets is their concentration on transitions, providing no support for expressing complicated transformations of input values to output values. Thus, they can be regarded as complementary to the functional ‘view’ as provided by SVDM.

The case study is intended to illustrate the complete development path from a problem statement to a working system. However, the final deliverable of SVDM is not an executable concurrent system, but at most an executable sequential system. A further transition is therefore needed into a programming language which supports the expression of both functional as well as concurrent behaviour. For this purpose the programming language Ada was chosen.

6.2 The analysis phase: functional aspects

6.2.1 SVDM step 1: Analyze the problem

The first step in SVDM consists of a global analysis of the required functionality of the system, by determining which components are inside the system and which components are outside the system. We will assume that the hardware components are coordinated by hardware controllers, modeled as external entities in the context diagram which receive information from and send information to the software components of the lift system. The context diagram for the lift system is shown in figure 6.1.
The external entities shown as double boxes indicate that one such external entity exists for each lift or floor. In the formal specification (shown in later sections) additional parameters are used to indicate from which lift or floor the data originates.

### 6.2.2 SVDM step 2: Decompose the context-diagram

The first decomposition step is based on the identification of functional subsystems. Analyzing the requirements, it seems reasonable to distinguish the following subsystems:

- **A global controller**, which is responsible for accepting lift requests from each of the $m$ floors, and for assuring that one of the $n$ lifts is scheduled to honor the request.
- **A local controller** for each of the $n$ lifts. A local controller is responsible for controlling the movements of a lift, including its components, such as its motor, the doors, its brake, etc.
- **A passenger interface** for each of the $n$ lifts. A passenger interface is used to accepts passenger requests from within lifts (either a request to go to a specific floor, an emergency indication, or a request to open the lift doors) and to show the responses of the
lift system by illuminating the appropriate lights.

- An information display subsystem which monitors the status of each lift and informs the site manager in case of abnormalities.
- A diagnosis subsystem which detects system malfunctions and alarms a technician to take appropriate measures.

The top-level DFD for the lift system is shown in figure 6.2; since it is basically the same as the one given in [St. Denis 87] we will not discuss here how it is derived from the informal requirements.

![Top-level DFD for the lift system](image)

**Figure 6.2: Top-level DFD for the lift system**

Since each of the subsystems still has a rather complicated functional behaviour, a second decomposition step has to be performed. In this case study the decomposition step is limited to the subsystems 'Global controller', 'Local controller' and 'Passenger interface', because the other two subsystems are less relevant with respect to the interesting properties of the lift system.

The refinement of the components 'Global controller', 'Local controller', and 'Passenger interface' are shown in figures 6.3, 6.4, and 6.5, respect-
6.2. The analysis phase: functional aspects

Thereby, the controller for the hardware is still rather complicated; an additional refinement step is therefore performed.

At this point, we have created a hierarchy of DFDs that consists of 4 levels:

Level 1: The context diagram (figure 6.1).
Level 2: The top-level DFD (figure 6.2).
Level 3: The refinements of three components of the top-level DFD (figures 6.3, 6.4, and 6.5).
Level 4: A further refinement of the hardware control component (figure 6.6).

6.2.3 SVDM step 3: Provide type information

The type information is collected by analyzing the data flows in the hierarchy of DFDs; for each data flow an associated type should be defined. No formal connection between the data flows and their types is made yet, this is postponed until the miniSpecifications have been
Figure 6.4: Refinement of the component 'Local controller'

Figure 6.5: Refinement of the component 'Passenger interface'

created. The two values \( m \) and \( n \) are also formally introduced at this
Figure 6.6: Refinement of the component ‘Control hardware’

stage. The specification is incomplete in the sense that not all type definitions are given; we will limit ourselves to those type definitions that we consider to be relevant for the remainder of the case study.

Considering the fact that the case study is much larger than the spelling checker example in the previous chapter, we use a simple structuring mechanism to divide the specification into separate components\(^{22}\).

\begin{verbatim}
module ValuesAndTypes

values

1.0  n : N \_1 is not yet defined;

2.0  m : N \_1 is not yet defined
\end{verbatim}

\(^{22}\) Notice that ISO/VDM-SL has no such structuring mechanism; although a proposal has been made to include modules in the language [Bear 88] – the same syntax of this proposal is used in this case study – until now no satisfactory semantics could be given to this proposal.
1.0, 2.0 The informal requirements indicate that the constants \( n \) and \( m \) can be assumed given, but no specific values are suggested.

end annotations;

3.0 \( \text{EmptySchedule} : \text{Schedule} = [] \quad \)

\section{types}

4.0 \( \text{Lift} = \mathbb{N}_1 \quad \)

1. \( \text{inv lift} \triangleq \text{lift} \leq n; \)

5.0 \( \text{Floor} = \mathbb{N}_1 \quad \)

1. \( \text{inv floor} \triangleq \text{floor} \leq m; \)

6.0 \( \text{Direction} = \text{UP} | \text{DOWN}; \quad \)

7.0 \( \text{LiftSchedules} = \text{Lift} \xrightarrow{m} \text{LiftSchedule}; \quad \)

8.0 \( \text{LiftSchedule} = \text{Floor}^* \quad \)

1. \( \text{inv ls} \triangleq \text{is-Circular} (ls) \quad \)

\section{annotations}

8.1 The invariant ensures that a lift can only be scheduled such that it moves in a circular fashion.

end annotations;

9.0 \( \text{LiftPositions} = \text{Lift} \xrightarrow{m} \text{Position}; \quad \)

10.0 \( \text{Position} = \text{Floor} | (\text{Floor} \times \text{Floor}) \quad \)

\section{annotations}

10.0 A lift can either be at a floor, or it can be in between two floors. In the latter case its position is modeled as a tuple of two components: the first component is the floor \( f \) which the lift has already passed, the second component is the floor the lift will pass next (either \( f - 1 \) or \( f + 1 \)).

end annotations;

11.0 \( \text{LiftStatus} = \text{IN\_SERVICE} | \text{OUT\_OF\_SERVICE}; \quad \)

12.0 \( \text{LiftLightControl} = \text{LiftLightOn} | \text{LiftLightOff}; \quad \)

13.0 \( \text{LiftLightOn} :: \text{light} : \text{LiftLight}; \quad \)

14.0 \( \text{LiftLightOff} :: \text{light} : \text{LiftLight}; \quad \)

15.0 \( \text{LiftLight} = \text{Floor} | \text{Direction} | \text{DOORS\_OPEN} | \text{ALARM}; \quad \)
6.2. The analysis phase: functional aspects

16.0  \( PassengerRequest = FloorRequest \ | \ OPEN\_DOORS \ | \ EMERGENCY; \)

17.0  \( FloorRequest = Floor; \)

18.0  \( FloorLightControl = FloorLightOn \ | \ FloorLightOff; \)

19.0  \( FloorLightOn :: floor : FloorLight; \)

20.0  \( FloorLightOff :: floor : FloorLight; \)

21.0  \( FloorLight = UP \ | \ DOWN; \)

22.0  \( AtDestination = \mathbb{B} \)

functions

23.0  \( is\_Circular : LiftSchedule \rightarrow \mathbb{B} \)

1.  \( is\_Circular (s) \triangleq \)

2.  \( \exists i, j \in \text{inds} s \cdot (\forall k \in \{1, \ldots, i-1\} \cdot s(k) < s(k+1) \land \)

3.  \( \forall k \in \{i, \ldots, j-1\} \cdot s(k) > s(k+1) \land \)

4.  \( \forall k \in \{j, \ldots, \text{len} s-1\} \cdot s(k) < s(k+1) \land \)

5.  \( s(1) \geq s(\text{len} s)) \lor \)

6.  \( (\forall k \in \{1, \ldots, i-1\} \cdot s(k) > s(k+1) \land \)

7.  \( \forall k \in \{i, \ldots, j-1\} \cdot s(k) < s(k+1) \land \)

8.  \( \forall k \in \{j, \ldots, \text{len} s-1\} \cdot s(k) > s(k+1) \land \)

9.  \( s(1) \leq s(\text{len} s)) \)

annotations

23.0  The functions checks if a lift schedule follows an up-down-up pattern or an down-up-down pattern, both of which are allowed. Notice that the function yields true when an empty schedule is passed as its argument.

end annotations

end ValuesAndTypes

6.2.4 SVDM step 4: Produce mini-specifications

The next step in SVDM involves the construction of mini-specifications for the low-level DFDs. For reasons of space, in this chapter this will be limited to the component 'Global controller'. The specification of this component is based on the low-level DFD as shown in figure 6.3; according to the mapping defined between a DFD and a formal specification, the data stores are transformed into state components and
each data transformer is transformed into a VDM operation with the same name.

module GlobalController

24.0 state LiftSystem of
   .1 lift_schedules : LiftSchedules
   .2 lift_positions : LiftPositions
   .3 init mk-LiftSystem (ls, lp) ≜
   .4   ls = {lift ↦ EmptySchedule | lift : Lift} ∧
   .5   ps = {lift ↦ 1 | lift : Lift}
   .6 end

operations

25.0 RecordLiftStatus (lift : Lift, status : LiftStatus)
   .1 ext wr lift_schedules : LiftSchedules
   .2 post status = IN_SERVICE ⇔
   .3   lift_schedules = lift_schedules ⊕ {lift ↦ EmptySchedule} ∧
   .4   status = OUT_OF_SERVICE ⇔
   .5   lift_schedules = lift_schedules ⊣ lift ;

26.0 HandleRequestFromFloor (at_floor : Floor, direction : Direction)
   .1 ext wr lift_schedules : LiftSchedules
   .2 rd lift_positions : LiftPositions
   .3 pre at_floor = 1 ⇒ direction = UP ∧
   .4   at_floor = m ⇒ direction = DOWN
6.2. The analysis phase: functional aspects

post let lift ∈ dom lift_schedules be st
    \( \text{SameSchedulesApartFrom} \ (\text{at} \_ \text{floor}, \)
    \( \phantom{\text{same_schedules}} (\text{lift} \_ \text{schedules} (\text{lift})), \)
    \( \phantom{\text{same_schedules}} \text{lift} \_ \text{schedules} (\text{lift}) \) ∧
    \( \phantom{\text{same_schedules}} \) if \( \text{direction} = \text{UP} \)
    \( \phantom{\text{same_schedules}} \text{then} \text{at} \_ \text{floor} ∈ \text{AscendingPartOf} (\text{lift} \_ \text{schedules} (\text{lift})) \)
    \( \phantom{\text{same_schedules}} \text{else} \text{at} \_ \text{floor} ∈ \text{DescendingPartOf} \)
    \( \phantom{\text{same_schedules}} (\text{lift} \_ \text{schedules} (\text{lift})), \)
    \( \phantom{\text{same_schedules}} \) if \( \text{direction} ∈ \text{DirectionsOf} (\text{lift} \_ \text{schedules} (\text{lift})) \)
    \( \phantom{\text{same_schedules}} \text{then Distance} (\text{lift} \_ \text{positions} (\text{lift}'), \text{at} \_ \text{floor}) < \)
    \( \phantom{\text{same_schedules}} \text{Distance} (\text{lift} \_ \text{positions} (\text{lift}), \text{at} \_ \text{floor}) \)
    \( \phantom{\text{same_schedules}} \text{else Distance} (\text{lift} \_ \text{positions} (\text{lift}'), \text{at} \_ \text{floor}) > \)
    \( \phantom{\text{same_schedules}} \text{Distance} (\text{lift} \_ \text{positions} (\text{lift}), \text{at} \_ \text{floor}) \) in
    \( \phantom{\text{same_schedules}} \text{lift} \_ \text{schedules} = \{ \text{lift} ← \text{ift} \_ \text{schedules} (\text{lift}) \} \);
Case study: a lift system

```
.2  post to_floor ∈ elems lift_schedules(lift) ∧
.3    ∀ floor ∈ elems lift_schedules(lift).
.4  floor ∈ elems lift_schedules(lift);
```

30.0 SendLiftToNewDestination

```
.1  (lift : Lift) result : Destination × Destination × LiftSchedules
.2  ext rd lift_schedules : LiftSchedules
.3  pre lift_schedules(lift) ≠ EmptySchedule
.4  post let nd = hd lift_schedules(lift) in
.5    result = mk-(nd, nd, lift_schedules)
```

functions

31.0 DirectionsOf : Schedule → Direction-set

```
.1  DirectionsOf (schedule) ⌟
.2    if schedule = EmptySchedule
.3    then {direction | direction : Direction}
.4 elseif Is-AscendingAndThenDescending (schedule)
.5    then {UP}
.6    else {DOWN};
```

32.0 ScheduledFloors : LiftSchedules → Floor-set

```
.1  ScheduledFloors (ls) ⌟
.2    ∪ {elems ls(lift) | lift ∈ dom ls}
```

annotations

32.0 The function ScheduledFloors returns the set of all floors that have been scheduled for at least one lift, either resulting from requests from the floor or request from within a lift.

end annotations

end GlobalController

Analyzing this formal specification, the following observations can be made with respect to the validation against the original (informal) requirements:
• Operation *RecordLiftStatus* plays a role in satisfying requirement R10; although R10 does not completely specify which action should be taken when a lift is out of service, it seems reasonable to remove that lift from the lift schedules.

• Operation *HandleRequestFromFloor* satisfies (in combination with operation *ControlFloorLight*) requirement R3. When a request from a floor has been received, a lift is allocated to visit that floor by selecting a lift from the available lifts and adding the floor to the schedule for that lift. *HandleRequestFromFloor* also satisfies requirement R6, although 'minimizing the waiting time' is in fact rather difficult to guarantee and therefore this requirement has been interpreted as 'use the lift that is nearest to the floor at which the request was issued and is traveling in the right direction'.

• Operation *HandleRequestFromLift* satisfies requirement R1. When a floor has been selected from within a lift, then that floor is added to the schedule for that lift.

### 6.3 The analysis phase: concurrent aspects

*Petri nets* are used in this case study to model the concurrent aspects of the various components. A Petri net is an abstract, formal model of information flow. Its major use has been the modeling of systems of events in which it is possible for some events to occur concurrently, with the added value that a net may be analyzed using Petri net theory [Peterson 77]. A Petri net can be regarded as a bipartite, directed graph $N = (T, P, A)$, where

1. $T = \{t_1, t_2, ..., t_n\}$ is a set of transitions;
2. $P = \{p_1, p_2, ..., p_m\}$ is a set of places; $(T \cup P$ form the nodes of $N)$, and
3. $A \subseteq \{ T \times P \} \cup \{ P \times T \}$ is a set of directed arcs.

A *marking* $M$ of a Petri net is a mapping

$$M : P \rightarrow I$$

where $I = \{0, 1, 2, \ldots\}$. $M$ assigns tokens to each place in the net. A Petri net with an associated marking is called a marked Petri net.
Pictorially, places are represented by circles, transitions by bars, and tokens by small black dots.

The semantics of a Petri net are specified by defining certain simulation rules: The set of input places of a transition \( t \) is given by \( I(t) = \{ p \mid (p, t) \in A \} \). The set of output places of a transition \( t \) is given by \( O(t) = \{ p \mid (t, p) \in A \} \). A transition \( t \) is said to be enabled in a Petri net \( N = (P, T, A) \) with marking \( M \) if \( M(p) > 0 \) for all \( p \in I(t) \). An enabled transition can fire by removing a token from each input place and putting a token in each output place. This results in a new marking \( M' \). The firing of transitions is asynchronous.

The Petri net formalism as such has no direct link with any specific problem domain. Thus, when using Petri nets for modeling purposes, the components of the net must be associated with the components of the system to be modeled; this is sometimes referred to as an interpretation of the Petri net. In the lift system case study, therefore, an association must be made between the components of a Petri net and the components of an SVDM specification. This association results in a special kind of Petri net, called Predicate/event net [Brauer 82], in which predicates may be associated with places and in which events may be associated with transitions. In the context of SVDM, we will assume that these events correspond to the execution of the operation in the SVDM specification with the same name. Predicates correspond to the detection of the presence of one or more input values for the corresponding operation.

Given this interpretation, the Petri net (including its initial marking) that can be constructed modeling the flow of control for the global controller component is shown in figure 6.7.

The construction of the Petri net is based on an analysis of the DFD for ‘Global controller’ (figure 6.3), in which we can see that the data transformers ‘Record lift status’, ‘Handle request from floor’, ‘Handle request from lift’, and ‘Register lift position’ all have write access to data store ‘lift schedules’. By making sure that these data transformers are combined in a setting with only one thread of control, we can avoid problems due to possible simultaneous access of the latter data store.
The data transformer ‘Control floor lights’ has no such interference with the other data transformers, hence it can be executed concurrently with the other components. Since it can be executed in two different settings (once after ‘Handle request from floor’ and once after ‘Send lift to new destination’, it occurs in two different threads of control. ‘Global controller’ thus has three threads of control in total; in the Petri net this corresponds to an initial marking with three tokens.
6.4 The design phase

6.4.1 SVDM step 5: Add control information

The 5th step in the SVDM methodology comprises the incorporation of control information. In the development of the spelling checker in chapter 5, this step was trivial because an assumption was made that the system was a sequential system, and thus the required control information was already captured in the DFDs that had been produced.

For the lift system the situation is different; lifts are moving independently, and external events can occur which require immediate response. The control aspect of the lift system is thus more complicated, and the control component is, therefore, developed as a separate abstraction aspect, and merged together with the functional specification at a later stage.

6.4.2 SVDM step 6: Refine mini-specifications

The implicit specifications that were produced in step 4 of the SVDM approach have to be made explicit in step 6. It is hard to provide concrete guidelines for this step; it can be argued that specific choices for data refinements, control structures are motivated by the stated requirements (e.g. performance requirements), but the key issue – that of transforming a relationship over input and output values (post-condition) into an explicit prescription of state transformations (explicit operation in terms of statements) is essentially a creative step, and in practice depends very much on the experience of the designer.

```
module GlobalController

33.0 state LiftSystem of
   .1 lift_schedules : LiftSchedules
   .2 lift_positions : LiftPositions

   .3 init mk-LiftSystem (ls, lp) Δ
       ls = {lift ← EmptySchedule | lift : Lift} ∧
       lp = {lift ← 1 | lift : Lift}

   .6 end
```
6.4. The design phase

operations

34.0 \hspace{1em} \text{RecordLiftStatus : Lift} \times \text{LiftStatus} \xrightarrow{\circ} ()

34.1 \hspace{1em} \text{RecordLiftStatus (lift, status)} \triangleq

34.2 \hspace{1em} \text{if status = IN\_SERVICE then lift\_schedules := lift\_schedules \uparrow \{lift \leftrightarrow \text{EmptySchedule}\}}

34.3 \hspace{1em} \text{else lift\_schedules := lift\_schedules \downarrow lift;}

35.0 \hspace{1em} \text{Handle RequestFromFloor : Floor} \times \text{Direction} \xrightarrow{\circ} ()

35.1 \hspace{1em} \text{Handle RequestFromFloor (at\_floor, direction)} \triangleq

35.2 \hspace{1em} \text{let lift} \in \text{dom lift\_schedules be st}

35.3 \hspace{1em} \text{Same\_Schedules\_Apart\_From (at\_floor, lift\_schedules (lift)),}

35.4 \hspace{1em} \text{lift\_schedules (lift)) \wedge}

35.5 \hspace{1em} \text{if direction = UP then at\_floor } \in \text{Ascending\_Part\_Of (lift\_schedules (lift))}

35.6 \hspace{1em} \text{else at\_floor } \in \text{Descending\_Part\_Of (lift\_schedules (lift))} \wedge

35.7 \hspace{1em} \exists \text{lift}' \in \text{dom lift\_schedules .}

35.8 \hspace{1em} \text{if direction } \in \text{Directions\_Of (lift\_schedules (lift))}

35.9 \hspace{1em} \text{then Distance (lift\_positions (lift'), at\_floor) <}

35.10 \hspace{1em} \text{Distance (lift\_positions (lift'), at\_floor)}

35.11 \hspace{1em} \text{else Distance (lift\_positions (lift'), at\_floor) >}

35.12 \hspace{1em} \text{Distance (lift\_positions (lift'), at\_floor) in}

35.13 \hspace{1em} \text{lift\_schedules := lift\_schedules \uparrow \{lift \leftrightarrow lift\_schedules (lift)\}}

35.14 \hspace{1em} \text{pre at\_floor = 1 } \Rightarrow \text{ direction = UP} \wedge

35.15 \hspace{1em} \text{at\_floor = m } \Rightarrow \text{ direction = DOWN ;}

36.0 \hspace{1em} \text{ControlFloorLight : [Floor\_Light\_Control] } \times \text{[Floor\_Light\_Control]} \xrightarrow{\circ}

36.1 \hspace{1em} \text{Floor\_Light\_Control}

36.2 \hspace{1em} \text{ControlFloorLight (control}_1, \text{control}_2) \triangleq

36.3 \hspace{1em} \text{let floor\_light\_control be st (floor\_light\_control } = \text{control}_1 \lor \text{ floor\_light\_control } = \text{control}_2 \land \text{ floor\_light\_control } \neq \text{ nil in}

36.4 \hspace{1em} \text{return floor\_light\_control}

36.5 \hspace{1em} \text{pre control}_1 = \text{nil } \lor \text{ control}_2 = \text{nil ;}
\[ RegisterLiftPosition : \text{Lift} \times \text{Position} \times \text{Floor} \xrightarrow{\sigma} \text{LiftLightControl} \]
\[ \]
\[ RegisterLiftPosition \,(\text{lift}, \, \text{position}, \, \text{destination}) \triangleq \]
\[ (\text{lift_positions} := \text{lift_positions} \uplus \{\text{lift} \mapsto \text{position}\}; \]
\[ \text{if position} = \text{destination} \]
\[ \text{then (lift_schedules} := \text{lift_schedules} \uplus \{\text{lift} \mapsto \text{hd lift_schedule}\}; \]
\[ \text{return mk-Off (destination)} \)
\[ \text{else return nil}; \]
\[ \]
\[ HandleRequestFromLift : \text{Lift} \times \text{Floor} \xrightarrow{\sigma} () \]
\[ \]
\[ HandleRequestFromLift \,(\text{lift}, \, \text{to_floor}) \triangleq \]
\[ \text{let first_part} \uplus \text{second_part} = \text{lift_schedules} \,(\text{lift}) \text{ be st} \]
\[ \text{is-LiftSchedule} \,(\text{first_part} \uplus \text{to_floor} \uplus \text{second_part}) \text{ in} \]
\[ \text{lift_schedules} \,(\text{lift}) := \text{first_part} \uplus \text{to_floor} \uplus \text{second_part}; \]
\[ \]
\[ SendLiftToNewDestination : \]
\[ \text{Lift} \xrightarrow{\sigma} \text{Destination} \times \text{Destination} \times \text{LiftSchedules} \]
\[ \]
\[ SendLiftToNewDestination \,(\text{lift}) \triangleq \]
\[ \text{let nd} = \text{hd lift_schedules} \,(\text{lift}) \text{ in} \]
\[ \text{return mk-(nd, nd, lift_schedules)} \]
\[ \text{end GlobalController} \]

6.4.3 Combining the final SVDM specification and the Petri net

After the final VDM specification has been produced, the last step in the development of the lift system consists of combining the VDM specification with the Petri net specification into an Ada implementation (see figure 6.8). Ada was chosen, since this language provides facilities for concurrency.

Although not formally defined, the following heuristic transformation rules are used for the systematic transformation of VDM and Petri nets into Ada code:

- VDM 'modules' (our informal structuring mechanism) are trans-
formed into Ada packages. The structuring mechanism for VDM that we have adopted and ‘packages’ in Ada roughly provide the same facilities and can thus be used for the same purpose.

- Each thread of control in a Petri net is transformed into an Ada task. A Petri net may contain labeled places, which correspond to conditions being triggered by the outside world, possibly in a concurrent manner. Ada tasks are the only language constructs that provide such a possibility through select statements.

- Labeled input places are transformed into entry calls. Marks can be put in a labeled place in a Petri net, which corresponds with the event that some autonomous external component has performed some action which affects the Petri net under consideration. These external components can be seen as Ada tasks too, and since Ada tasks communicate with each other by means of a rendez-vous mechanism, the input places can be transformed into entry calls. The thread of the Ada code associated with the entry call then follows the actions prescribed by the Petri net.

- Labeled transitions are transformed into calls to Ada functions or procedures. As was argued before, labeled transitions in the Petri net correspond to the ‘execution’ of VDM operations. As VDM
operations are transformed into Ada procedures (or functions), labeled transitions correspond to procedure (or function) calls in the Ada world.

- The Ada code for these functions or procedures is derived from the VDM specification. This transformation is fairly straightforward because VDM-SL and Ada provide roughly the same statements for purely sequential storage changes. The only care that should be taken is in the transformation of the pattern matching mechanism of VDM-SL (Ada has no pattern matching) and in the transformation of types: Ada has strong type equivalence whereas VDM has weak type equivalence. As it turns out, neither provides serious problems in this case study.

The Ada code produced by following this scheme for component ‘Global controller’ is given below; the rest of the code has not been included in this thesis.

6.4.4 The Global Controller component

The Ada code for the global controller is based on the observation that the Petri net for the global controller has three threads of control. Therefore, the Petri net has been partitioned into three components, each of which corresponds to an Ada task (figure 6.9). This particular configuration was chosen because all the entry calls can now be concentrated in one task, in this way making it possible to keep the interface of the component simple.

```ada
with ValuesAndTypes;
use ValuesAndTypes;

package GlobalController is

  -- The interface contains task declarations with an entry
  -- for each labeled input place in the Petri net.

  task GlobalController is
    entry StatusReceived (l: in Lift; s: in LiftStatus);
    entry UpDownRequestReceived (f: in Floor; d: in Direction);
    entry FloorRequestReceived (l: in Lift; f: in Floor);
    entry PositionReceived (l: in Lift; p: in Position);
  end GlobalController;

end GlobalController;
```
6.4. The design phase

Figure 6.9: Partitioning of the Petri net for the component ‘Global controller’

with ValuesAndTypes;
use ValuesAndTypes;

package body GlobalController is

  task type ControlFloorLightsTask is
    entry Switch (f: in Floor; flc: in FloorLightControl);
  end ControlFloorLightsTask;

-- State components in the VDM specification:
lift_schedules: LiftSchedules;
lift_positions: LiftPositions;
-- The tasks controlling the floor lights cannot be called
-- from outside, hence they are declared in the package body
-- of ‘GlobalController’.
ControlFloorLightsTask1: ControlFloorLightsTask;
ControlFloorLightsTask2: ControlFloorLightsTask;

-- An Ada procedure or function is defined for each VDM
-- operation in module ‘GlobalController’.
procedure
  RecordLiftStatus (l: in Lift; s: in LiftStatus)
  is separate;
procedure
  HandleRequestFromFloor (f: in Floor; d: in Direction)
  is separate;
procedure
  RegisterLiftPosition (l: in Lift; p: in Position)
  is separate;
procedure
  HandleRequestFromLift (l: in Lift; f: in Floor)
  is separate;
function
  SendLiftToNewDestination (l: in Lift) return Destination
  is separate;
procedure
  ControlFloorLights (f: in Floor; flc: in FloorLightControl)
  is separate;

task body ControlFloorLightsTask is

  flc_temp: FloorLightControl;
  f_temp: Floor;

begin
  loop
    accept Switch (f: in Floor; flc: in FloorLightControl) do
      flc_temp := flc;
      f_temp := f;
      end Switch;
      ControlFloorLights (f_temp, flc_temp);
    end loop;
  end ControlFloorLightsTask;

task body GlobalController is

begin
  loop
    select
      accept FloorRequestReceived (l: in Lift; f: in Floor) do
        HandleRequestFromLift (l, f);
      end FloorRequestReceived;
      or
      or
    end select;
end GlobalController;
accept UpDownRequestReceived
    {f: in Floor; d: in Direction} do
    HandleRequestFromFloor (f, d);
    ControlFloorLightsTask1.Switch (f, ON);
end UpDownRequestReceived;

or
accept PositionReceived {l: in Lift; p: in Position} do
    d := SendLiftToNewDestination (l);
    RegisterLiftPosition (l, p);
    if p.at_or_in_between = at_floor
    then ControlFloorLightsTask2.Switch (p.at_floor, OFF);
    end if;
end PositionReceived;

or
accept StatusReceived {l: in Lift; s: in LiftStatus} do
    RecordLiftStatus (l, s);
end StatusReceived;
end select;
end loop;

end GlobalController;

begin

lift_schedules := {1 .. n => EmptySchedule};
lift_positions := {1 .. n => (at_floor, l)};

end GlobalController;

6.5 Evaluation

We can now proceed with an evaluation of the case study against the criteria as presented at the beginning of this chapter.

6.5.1 Ease of construction of the specifications

A classic difficulty with the isolated use of formal methods is the effort and skill needed to create the required formal specifications. With respect to the skill needed to create the various specifications in the case study, it can be observed that considerable progress has been made: the developer who has experience with the use of Structured Analysis to create a model of the system, can apply these same techniques for the creation of a formal specification, as illustrated through the SVDM approach. The same does not hold for the creation of the Petri net
specification where – in this case – it is anticipated that the Petri net is created from scratch, i.e. directly from the informally stated user requirements. The Petri net in this case study is, however, a rather simple one, yet it plays a useful role. By explicitly modeling the concurrency aspects of the system, it becomes possible to reason about these concurrency aspects (although this has not been demonstrated in this case study), while at the same time the net puts constraints on the further development of the SVDM specification, in this way increasing the control over the development.

The effort needed to create the formal specifications in the case study has been decreased by the use of the SVDM approach; some specifications can be automatically generated (although no tool support for such creation yet exists), and in addition, by choosing a low level of granularity of the steps involved, the changes that have to be made from one specification to the other are small and well-defined.

### 6.5.2 Comprehensibility of the specifications

The comprehensibility of a specification is very much tied to the level of abstraction that has been achieved; by focusing on relevant details only it becomes easier to fully understand what the specification aims to express. In the case study this has been achieved in three ways:

1. By the use of formal methods. VDM-SL supports abstraction mechanisms such as representational abstraction and operational abstraction, both of which are extensively used in the case study.

2. By the use of SVDM: the use of SA in combination with VDM augments the purely textual representation of VDM with diagrams (the data flow diagrams from SA); this provides two different views on the same subject matter.

3. By the use of orthogonal techniques: SVDM is used to model the functional aspects of the system, Petri nets are used to model the concurrency aspects of the system. This ‘separation of concerns’ can be seen as an abstraction mechanism where specific aspects of a system can be textually separated.
The disadvantage of the use of these various techniques is that in the development process a large number of specifications are created, in this way increasing the effort needed to maintain and comprehend these specifications.

### 6.5.3 Completeness of the specifications

The degree to which the specifications produced provide a complete description of all aspects of the lift system is hard to judge. We have the impression that the system specifications are indeed complete, but this is impossible to prove. The most that can be said is that by using a combined structured/formal method, it was possible to describe a complete software process, i.e. a process which addresses the entire path from informal user requirements to an implementation of the system.

### 6.5.4 Usefulness of the specifications

Not all specifications produced in the case study are useful in the same way. It can easily be argued, e.g. that the VDM specifications that are produced at the early stages of applying SVDM, have little practical value; they are not directly used in the next steps of SVDM, but merely are a (formal) indication of what has been modeled with the data flow diagrams at that particular stage; as such they can be seen as a formal meaning of the DFDs. The integration that has been achieved between SA and VDM thus is useful in that it makes the relationship between these two methods clear, and it creates clarity as to what the use and place of the respective notations in the software process is. The case study also showed that a number of specifications that are produced have a clear practical purpose; they are directly used as starting points for subsequent development steps.

### 6.6 Conclusions and future research

The case study has shown that an approach to software development based on a combined structured/formal method has a number of ad-
vantages. First of all, the control over the software process is increased using SVDM. By 'control over the process' we mean the degree to which it is possible at all points to steer and guide the process in a desired direction. To be able to do this it is necessary that at all points in the process assessments can be made on its current status. In SVDM this is achieved in two ways:

1. Multiple views (an informal graphical one and a formal textual one) on the system under development are provided, which allow different aspects of the system to be clearly highlighted;

2. A more structured transition path from software requirements (almost) up until system implementation has been defined, which provides suitable points at which assessment can take place.

The resulting combination's ability to analyze a specification from different viewpoints provides a better insight in different aspects of the system. However, at the same time it must be admitted that the limitations of such an approach are not yet entirely clear. 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.

A second advantage is that the quality of the elementary transformation steps in the process has been increased through the application of SVDM, because:

1. At the beginning of the development process, where modeling and analyzing is important, the designer is supported through SA heuristics. When the process continues, and technical aspects play a more important role, the formal method plays a more predominant role. Thus, the right support is provided at the right time.

2. Each software process has a point where the informal real world inevitably has to be mapped onto a very formal implementation, to be executed on a computer. In the case of SVDM, this particular transition is relatively well-defined.

So, the fact that a formal method has been embedded in the process, in combination with the fact that this embedding has well-defined
connections with its surroundings (e.g. other methods and techniques used), makes that the overall quality of the design steps has been increased.

As with many other research projects, apart from a number of conclusions, this research projects also has a number of open ends, to be discussed below.

**Analysis of the value of specifications**

One of the differences between formal specifications produced with SVDM and the formal specifications produced by applying VDM directly is that the former seem to be rather structured and voluminous, whereas the latter are more flat and sober. This is mainly due to the fact that SVDM specifications are produced hand-in-hand with the construction of a hierarchy of DFDs, i.e. the decomposition process of SA has its direct counterpart in the formal specification. The advantage is that the transitions that have to be performed by the designer once these formal specifications have been produced are localized in the mini-specifications (in terms of VDM-SL) which in general are rather simple. The disadvantage is that it is unclear what the consequences will be on performing proofs on the formal specification. Because of the structured nature, such proofs may become more complicated. Further research is needed to clarify this issue.

**Tool support for combined structured/formal methods**

The presentation of the combined structured/formal method SVDM showed that a number of formal specifications can be produced (semi) automatically. In fact, those specifications are usually the ones that merely serve as documentation, and do not provide any direct benefit to the designer. Clearly, if such specifications are to be produced by hand, than this would make the combined method much less acceptable to the user. Tool support may, therefore, not be seen as a direct research goal in itself, but as a prerequisite for doing further research with respect to further maturing combined structured/formal methods.
Industrial case studies
Academic exercises such as this research project may end in strong arguments in favor or against some novel concept, but final acceptance very much depends on whether the approach can be scaled up to handle real-world sized problems and whether the approach is robust enough to survive in an industrial environment. This holds for the application of combined structured/formal methods as well.
In this thesis, the findings of research exploring the application of formal methods in software engineering have been presented; the global aim of the research project (as formulated in section 1.4) was to increase the understanding in the relationships between formal methods and more conventional software engineering by studying several perceived problems with the industrial application of formal methods. With respect to this aim the research project can be considered to be a success; the respective problem areas have been investigated, solutions have been considered and analyzed, and an inventory of remaining problems has been made, in this way opening a window on new research possibilities. We will evaluate each problem area individually in the next three sections, and then evaluate the overall research project in section 7.4.

7.1 Standardization and support for VDM

The first part of the research project consisted of a contribution to the work currently being undertaken to standardize the formal specifica-
tion language VDM-SL. Although the VDM-SL standard can be considered very interesting from a technical point of view, there are pragmatic disadvantages which, in our opinion, hinder the acceptance of the standard by industrial organizations. One major disadvantage is the fact that the standard contains a complete (voluminous) formal definition of VDM-SL makes the standard as a whole rather inaccessible and makes the language difficult to understand. Of course, this in itself need not necessarily be seen as a disadvantage; if we can accept that the standard can only be used by the 'happy few' who have had extensive training and experience with formal language definitions, then accessibility is not much of an issue. If, on the other hand, we consider it important that formal techniques will be used by a much broader range of the software producing community, then the way the language is defined is of extreme importance.

The effect of in-accessibility can be decreased by providing an adequate reference manual for the language, explaining its features in a more informal manner, but this does not make the standard itself better accessible. At the same time it is clear that formally defining the standard language has important advantages over non-formal language definitions; advantages that become even more important in standardization where preciseness is one of the main motivations for wanting to have a standard in the first place. We believe therefore, that the current trend in increasing use of formal language definitions in standards should be further supported and encouraged, but that at the same time these formal definitions should be made more 'usable', e.g. by:

- using a different kind of semantics, e.g. axiomatic semantics or operational semantics. Axiomatic semantics have the advantage that they are usually smaller in size. Operational semantics are more intuitive to understand, but can only be used for executable specification languages.
- using specification languages which are smaller, and therefore less expressive.

We believe that in this way the accessibility of formal language definitions can be increased, thus increasing their chances of being used by industrial organizations.
7.2 The application of VDM in DoD-STD-2167A

The second part of the research project consisted of an analysis of how and where VDM can best be applied in a commonly accepted framework for software development, namely DoD-STD-2167A. This work has resulted in a clear picture of the general benefits of using formal methods with respect to their non-formal counterparts, aggravated on the activities for which formal methods can be used and their mode of application, i.e. for specification, development, or verification purposes.

The work furthermore showed that most formal methods that are currently available are partial, i.e. they provide support for the specification and/or development of a limited number of aspects of the system for a limited part of the development path. This in itself need not be a disadvantage; in fact, such a focus on a particular system aspect or part of the development provides possibilities to concentrate on what is considered to be particularly important for the problem at hand. On the other side, however, this introduces problems with respect to how formal methods fit in with other techniques used in the development process. This particular problem has been the subject of the third part of the research project.

7.3 A combination of SA and VDM

The third part of the research project consisted of an examination of combinations of structured and formal methods, conducted through the analysis of possible relationships between the major notations of a ‘typical’ structured method and a ‘typical’ formal method (SA and VDM, respectively), and through the formulation of methodological guidelines for a combination of SA and VDM, called SVDM, marking the path from requirements analysis to implementation of a software system. SVDM was illustrated by means of a small case study.

The research showed that an approach to software development based on a combined structured/formal method has several advantages (as discussed in detail in section 6.6), e.g. the fact that the resulting combination’s ability to analyze a specification from different viewpoints
provides a better insight in several aspects of the system. However, it must be admitted that the limitations of approaches like these are not yet entirely clear. To make a 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.

Another point which needs further consideration is the fact that the structure of formal specifications produced using SVDM directly reflects the structure of the DFD hierarchy, and it is unclear what effect this will have on the ease (or unease) with which characteristics of a specification can be proven. Larger case studies are needed to investigate these issues. We are confident, however, that these larger case studies will confirm our initial findings with the respect to the validity of our approach.

### 7.4 Overall evaluation

The main outcome of the research project described in this thesis is that we have shown that a controlled transition from completely informal software processes to software processes with an increased degree of formality is possible through the gradual introduction of formal methods. Such a gradual introduction can be achieved by constructing well-supported combinations of conventional methods for software development and formal methods. This research has shown both which technical aspects must be taken into account as well as what the appropriate place of such combinations is in common models for software development. The improvements over conventional software development are:

- the advantages of formal methods can be utilized;
- a smooth transition from the use of conventional techniques to more formal techniques can be achieved.

The use of a formal method which has been standardized may be an advantage in this respect, because such a standard stimulates the production of tools supporting the standard, but the actual style of the
standard is a critical factor in whether or not the standardized formal method will actually be used.

Concluding, we can say that the gate to an increased use of formal methods by industrial organizations – not limited to the specification, development and verification of safety-critical systems alone – is open. Formal methods can be used meaningfully for other types of systems as well, and the research reported on in this thesis has shown how remaining obstacles on the road to a successful integration of formal methods in industrial software engineering can be overcome.
References


References


References


Index

Index entries for VDM functions and operations only refer to the page at which the function/operation is defined.

A
abstract data structures, 26
abstraction, 3, 22, 23
    operational, 23, 33, 93
    representational, 22, 33, 93
abstraction function, 34
access operations, 115
ACT ONE, 27
Ada, 32, 54, 150
    entry calls, 167
    implementation, 166
    packages, 91, 167
    tasks, 167
algebraic methods, 27
analysis phase, 128
ANNA, 27
application-based classification, 28
applicative constructs, 33
ASF, 27
ASLAN, 27
automated support, 28, 175
automatic transformations, 128
axiomatic
    methods, 27
    semantics, 2, 76
B
Backus Naur Form, 2
Bird-Meertens formalism, 30
BNF, 2
bottom-up tree walk, 46, 54
British Standards Institution, 32
BSI Panel, 32
bubbles, 101
C
calculus, 30
canonical notation, 24
CAS, 37, 41
case study, 14, 147
CCITT, 12
CCS, 27, 30
ChangeIntoWords, 137
channel, 115
CICS, 8
CIP, 30
   project, 86
Clear, 27
coding and CSU testing, 80
COLD, 27
combined method, 100
compile-time check, 55
complexity control, 3
composition, 128
Computer Software Component, 79
Computer Software Configuration Item, 78
Computer Software Unit, 79
concurrency, 88
concurrent system, 147
configuration control, 69
configuration management, 80
conformance clause, 37
consists-of-hierarchy, 81
constructive methods, 30
context diagram, 102, 106, 130
context-free grammar, 24
control flow, 109
diagram, 102
control information, 130, 137
control process, 19
control-extended DFDs, 103
ControlFloorLight, 159, 165
Core Abstract Syntax, 37, 41
CSC, 79
CSC integration and testing, 80
CSCI, 78
CSCI testing, 80
CSP, 27
CSU, 79

D
data dictionary, 102, 104, 107
data flow, 101, 114
data flow abstraction, 101
data flow arrows, 109
data flow diagram, 100, 101

Data Item Description, 80
data reification, 34
data store, 101, 106, 114
data transformer, 101, 106, 107
   primitive, 106
   recursive, 111
decision table, 102, 103
decomposition, 3, 22, 106, 128
definitely well-formedness, 44
Delete, 40
denotational semantics, 32, 44, 75
design phase, 128
design philosophy, 3
detailed design, 80
DFD, 101
   control-extended, 103
   formal semantics, 125
   hierarchy, 102, 103, 132, 133
   low level, 104
diagnosis subsystem, 152
DID, 80
directed graph, 101, 161
DirectionsOf, 160
discrete mathematics, 10
distributed systems, 12
DoD-STD-2167A, 13, 77
dynamic semantics, 41, 43
dynamic-semantic tool compliance, 73
DynamicCheck, 57, 58, 60

E
EBNF, 38, 39
EHDM, 27
elementary building blocks, 101
elementary transformation step, 17, 20, 28, 174
Encloses, 49
entity-relationship diagrams, 104, 129, 132
ERD, 104
ESTELLE, 12
EvalIfExpr, 43
EVES, 27
executable
language subset, 70
representation, 3
specifications, 10, 86
existential quantification, 111
explicit
requirements, 6
specifications, 33
expression trees, 46
expressions, 33
Extended Backus Naur Form, 38
extended full language conformance, 71
extended method, 100
external process, 101, 106, 107
formal specification method, 29
formal specifications, 2, 23
executable, 10
non-executable, 73
formal systems, 3
formal verification method, 29
FSM, 29
full language conformance, 71
full tool compliance, 73
functional requirements, 128
FVM, 29
G
GetFile, 120–123
GetFileToSplit, 120, 121, 125
global controller, 151
refinement of, 152
graphical theory, 103
graphical view, 127
H
HandleRequestFromFloor, 158, 165
HandleRequestFromLift, 159, 166
hardware, iii
Hardware Configuration Item, 78
hardware controller
refinement of, 153
heuristics, 174
hierarchy of DFDs, 102, 103, 132, 133
high-level programming languages, v, 2
higher-order logic, 27
HOL, 27
HWCI, 78
I
IfExpr2IfExpr, 42
imperative constructs, 33
implementation
bias, 27
phase, 5
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>risk</td>
<td>93</td>
</tr>
<tr>
<td>implicit requirements</td>
<td>6</td>
</tr>
<tr>
<td>specifications</td>
<td>33</td>
</tr>
<tr>
<td>industrial case studies</td>
<td>176</td>
</tr>
<tr>
<td>information display subsystem</td>
<td>152</td>
</tr>
<tr>
<td>information flow modeling</td>
<td>102</td>
</tr>
<tr>
<td>input places</td>
<td>162</td>
</tr>
<tr>
<td>integrated approach</td>
<td>100</td>
</tr>
<tr>
<td>Interface Requirements Specifi-</td>
<td></td>
</tr>
<tr>
<td>cation</td>
<td>78</td>
</tr>
<tr>
<td>International Standards Organi-</td>
<td></td>
</tr>
<tr>
<td>zation</td>
<td>12</td>
</tr>
<tr>
<td>invariants</td>
<td>33, 47</td>
</tr>
<tr>
<td>IRS</td>
<td>78</td>
</tr>
<tr>
<td>is-Circular</td>
<td>157</td>
</tr>
<tr>
<td>IsCompatible</td>
<td></td>
</tr>
<tr>
<td>IsCompatible_w</td>
<td>58</td>
</tr>
<tr>
<td>ISO</td>
<td>12</td>
</tr>
<tr>
<td>ISO/VDM-SL standard</td>
<td>37</td>
</tr>
<tr>
<td>overview</td>
<td>38</td>
</tr>
<tr>
<td>J</td>
<td></td>
</tr>
<tr>
<td>Jackson System Design</td>
<td>87</td>
</tr>
<tr>
<td>JSD</td>
<td>87</td>
</tr>
<tr>
<td>L</td>
<td></td>
</tr>
<tr>
<td>language conformance</td>
<td>70, 71</td>
</tr>
<tr>
<td>language symbols</td>
<td>24</td>
</tr>
<tr>
<td>Larch</td>
<td>27</td>
</tr>
<tr>
<td>level 0 VDM specification</td>
<td>129, 133, 134</td>
</tr>
<tr>
<td>level 1 structure charts</td>
<td>129, 138</td>
</tr>
<tr>
<td>level 1 VDM specification</td>
<td>129, 136</td>
</tr>
<tr>
<td>level 2 structure charts</td>
<td>129, 140</td>
</tr>
<tr>
<td>level 2 VDM specification</td>
<td>129, 137</td>
</tr>
<tr>
<td>level 3 structure charts</td>
<td>130</td>
</tr>
<tr>
<td>level 3 VDM specification</td>
<td>129, 139</td>
</tr>
<tr>
<td>level 4 VDM specification</td>
<td>130, 141</td>
</tr>
<tr>
<td>LexicallySmaller</td>
<td>133</td>
</tr>
<tr>
<td>life cycle models</td>
<td>78</td>
</tr>
<tr>
<td>rapid-prototyping</td>
<td>78</td>
</tr>
<tr>
<td>Spiral</td>
<td>78</td>
</tr>
<tr>
<td>Waterfall</td>
<td>78</td>
</tr>
<tr>
<td>lift system</td>
<td>14, 148</td>
</tr>
<tr>
<td>local controller</td>
<td>151</td>
</tr>
<tr>
<td>refinement of</td>
<td>152</td>
</tr>
<tr>
<td>logic</td>
<td>10</td>
</tr>
<tr>
<td>loose specification</td>
<td>43</td>
</tr>
<tr>
<td>looseness</td>
<td>23, 43, 109, 117, 135</td>
</tr>
<tr>
<td>LOTOS</td>
<td>12, 27, 84</td>
</tr>
<tr>
<td>low-level DFD</td>
<td>104</td>
</tr>
<tr>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Manchester DataFlow Machine</td>
<td>103</td>
</tr>
<tr>
<td>mathematical model</td>
<td>23</td>
</tr>
<tr>
<td>MET</td>
<td>51</td>
</tr>
<tr>
<td>Meta-IV</td>
<td>32</td>
</tr>
<tr>
<td>method</td>
<td></td>
</tr>
<tr>
<td>combined</td>
<td>100</td>
</tr>
<tr>
<td>extended</td>
<td>100</td>
</tr>
<tr>
<td>mini-specifications</td>
<td>102, 129, 130, 135, 157</td>
</tr>
<tr>
<td>minimal constraints</td>
<td>27</td>
</tr>
<tr>
<td>minimal enclosing type</td>
<td>51</td>
</tr>
<tr>
<td>model domain</td>
<td>23</td>
</tr>
<tr>
<td>model-oriented formal methods</td>
<td>26</td>
</tr>
<tr>
<td>multi-sorted algebras</td>
<td>27</td>
</tr>
<tr>
<td>multiple views</td>
<td>174</td>
</tr>
<tr>
<td>N</td>
<td></td>
</tr>
<tr>
<td>NATO conference</td>
<td>v</td>
</tr>
<tr>
<td>natural language</td>
<td>2, 10</td>
</tr>
<tr>
<td>non-executable specification lan-</td>
<td></td>
</tr>
<tr>
<td>guage</td>
<td>73</td>
</tr>
<tr>
<td>non-functional requirements</td>
<td>84,</td>
</tr>
<tr>
<td>104, 128</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
</tr>
<tr>
<td>OAS</td>
<td>37, 39, 41</td>
</tr>
<tr>
<td>OBJ3</td>
<td>27</td>
</tr>
<tr>
<td>object-oriented design</td>
<td>87, 103</td>
</tr>
<tr>
<td>one-dimensional verification</td>
<td>20, 28</td>
</tr>
</tbody>
</table>
operation modeling, 34
operational
   abstraction, 23, 33, 93
   semantics, 32, 76
optional-domain constructor, 108
Outer Abstract Syntax, 37, 39
output places, 162

P
PAC approach, 105, 117, 125, 132,
   133, 144
PAISley, 27, 84
paper-and-pencil tool, 71
parallel approach, 99
partial language conformance, 72
passenger interface, 151
   refinement of, 152
path expressions, 103
Petri net, 103
   interpretation, 162
   labeled places, 167
   labeled transitions, 167
   marking, 161
   semantics, 162
   transitions, 162
Petri nets, 27, 103, 150, 161
PL/I, 32
planning, 4
possible types set, 46, 47
possibly well-formedness, 45
post-condition quotation, 125
pragmatic support, 69
pre- and post-conditions, 23, 33
Predicate/event net, 162
preliminary design, 80
primary-mapping step, 104, 119
primitive data transformer, 106,
   129
primitive data transformers, 130
problem, 18
problem representation, 18
process algebra methods, 27
ProCos, 27
programmer productivity, 5
programming languages, 22
Prolog, 90
proof assistant, 11
proof obligations, 29
property-oriented methods, 26, 27
prototype, 10, 86
prototypers, 69, 70
PTO, 52–54

Q
quote literal, 121

R
RAISE, 27, 86, 92
rapid-prototyping, 78
reactive systems, 100
RecordLiftStatus, 158, 165
recursive data transformers, 111
refinement, 3, 25, 88
RegisterLiftPosition, 159, 166
rely/guarantee conditions, 100
representational abstraction, 22,
   33, 93
requirements
   functional, 128
   non-functional, 128
requirements specification, 5, 82
research
   aim, 12
   evaluation, 177
   outline, 12
retrieve function, 35
rigorous
   development method, 30
   specification method, 29
   verification method, 29
RML, 84
RSM, 29
rule-based approach, 103
run-time check, 55
RVM, 29
S
SA, 101
SA/SD, 87, 101, 102
SAC approach, 105, 114, 125, 132
safety-critical systems, 89, 93
satisfiability, 3
SAZ, 103
SC22/WG19, 32
ScheduledFloors, 160
SD, 101
SDL, 12
secondary document, 129, 134, 136, 140
secondary-mapping step, 104, 112
semantic foundation-based classification, 26
semantic support, 69
semantics, 2, 22, 100
axiomatic, 2
denotational, 32
of DFDs, 102
operational, 32, 76
SendLiftToNewDestination, 160, 166
separation of concerns, 22, 128
sequential systems, 101
set theoretic methods, 27
silver bullet, v
software
costs of, iv, 4
origin of, iii
systematic production of, v, 1
testing, 7, 30
software development management, 80
Software Engineering, v
software engineering, 80
software process, 78
activities, 78
control over, 174
conventional, 13
documentation, 78
introduction of errors, 5
model, 17
phases, 78
planning, 4, 92
position of formal methods, 9
progress, 78
software product evaluation, 80
software requirements
explicit, 6
implicit, 6
software requirements analysis, 80
software requirements document, 18
software specification, 17
solution, 18
solution representation, 18
Sort, 134, 136, 140, 142
source specification, 20, 28
specification
completeness, 173
comprehensibility, 148, 172
domain, 23
ease of construction, 148, 171
language, 21
style, 90
usefulness, 148, 173
spelling checker, 118, 130
Spiral model, 78
Split, 120–122, 124, 134, 136, 139, 142
SplitToSort, 134, 136, 138, 140, 143
SRD, 6, 18
standardization, 11, 13
state machine methods, 27
state transition diagrams, 102
Statecharts, 27
statements, 33
static semantics, 24, 37, 44, 45, 76
static system structure, 78
static-semantic tool compliance, 73
StaticCheck, 56, 58, 59
StaticCheck, 57
STC Reference Language, 81
Strong Policy, 56
strong type equivalence, 168
structure charts, 102, 129
Structured Analysis, 14, 97, 100, 101
Structured Design, 101
structured methods, v, 14, 99
structured transition path, 174
Structured VDM, 127
SubTypesOf, 50
SVDM, 127, 144, 147
   principles, 128
synchronization points, 113
syntactic support, 69
syntactic tool compliance, 72
syntax, 21
syntax checker, 11
syntax mapping, 37, 41, 61
   generation of quoted functions, 63, 66
   introduction of identifiers, 62
   transformation of
      error handlers, 65
      expressions, 65
      pre-conditions, 65
      the specification, 61
      the state, 67
      type definitions, 66
      value definitions, 67
      use of pre-conditions, 61
system integration and testing, 80
system requirements analysis/design, 80

T
target specification, 20, 28
τ, 108, 120
telecommunication industry, 12
Temporal Logic, 27
temporal logic, 27
term rewriting, 61
test case generation, 89
test suites, 73
testing, 7, 91
textual view, 127, 137
theorem provers, 69
Θ extension, 108, 116, 120, 123
thinking-tool, 93
thread of control, 162
three-valued logic, 70
TK, 30
tool compliance, 70, 72
transformation, 20
   process, 19
transformational methods, 30
transformational model, 100
two-dimensional verification, 20, 28
type
   checker, 11, 13
   checking, 47
   constraint checking, 46, 47, 54
   ordering, 48
   propagation, 46, 47
   system, 46

U
union types, 47
US Department of Defence, 78
uses-hierarchy, 81

V
validation, 6
VDL, 32
VDM, 11, 13, 27, 31
   development paradigm, 34
   domains, 33
VDM++, 87, 100
VDM-SL, 13, 31
   abstract syntax, 38
   ASCII representation, 40
   automated support, 68
   concrete syntax, 38
   conformance clause, 37
   dynamic semantics, 41, 43
executable subset, 138
language overview, 33
lexis, 40
mathematical representation, 40
standard, 37
standard overview, 38
standardization, 32
static semantics, 37, 44, 45, 76
structuring mechanism, 81, 155
symbol representations, 40
syntax mapping, 41, 61
verification, 6
one-dimensional, 20, 28
two-dimensional, 20, 28
version control, 69
Vienna Definition Language, 32
Vienna Development Method, 13, 31
view
  graphical, 127
textual, 127
W
walk-through session, 86
Waterfall model, 78
Weak Policy, 57
weak type equivalence, 168
well-formedness, 37
Z
Z, 12, 27, 103
Summary

Formal methods for software development (i.e. mathematically based methods for the specification and verified development of software systems) potentially enable larger software systems to be developed than can be done nowadays: they provide for complexity control by offering abstraction facilities, they make unambiguous and consistent specification possible at lower cost, and enable the verified design of software components. Unfortunately, the industrial take-up of formal methods has, so far, been rather slow. A number of reasons for this phenomenon can be given, the following of which are addressed in this thesis:

- The exact role of formal methods in the software development process has not been resolved. A positioning of formal methods in the framework of conventional software development is necessary so that they can be used in conjunction with traditional methods and techniques.
- There is little high-quality tool support available to support the application of formal methods.
- With a few exceptions, there exist no industrial standards for formal methods.

In this thesis, these issues are approached through a number of experiments, each involving the formal method VDM, and its specification language: VDM-SL.
In chapter 3, the issue of standardization and tool support is discussed. The ISO/VDM-SL standard – currently still under construction – has several interesting aspects. The following of these are addressed:

1. the static semantics of the language. Since VDM-SL is a weakly-typed language with union types, the notion of 'type correctness' is non-trivial. An approach is outlined which provides for an efficient yet simple way for type-checking a VDM-SL specification.

2. the syntax mapping between the syntax of the language and a normalized form of this syntax, which is used for the dynamic semantics of the language. This approach was chosen by the standardization committee to keep the size of the dynamic semantics as small as possible.

3. the meaning of tool conformance and language compliance in the context of VDM-SL. VDM-SL is (in general) a non-executable language, and, therefore, it is impossible to check whether or not a tool supporting the language conforms to the VDM-SL dynamic semantics. To accommodate tool builders a more fine-grained definition of tool conformance and language compliance is proposed.

The ISO/VDM-SL standard contains a completely formal definition of the language. This does, however, place severe restrictions on its accessibility and use.

In chapter 4 the application of VDM in the framework of a widely used standard for software development is analyzed: DoD-STD-2167A. The analysis mainly consists of a literature survey, placing published experiences with VDM against the phases in the software process as defined in DoD-STD-2167A, and the various kinds of activities carried out in each phase. The results of this analysis confirm some of the problems already noticed with the use of formal methods (lack of standardization, tool support, relation to other techniques and methods needs further investigation) and identifies the phases/activities for which formal methods can best be used when following DoD-STD-2167A. Since the latter is a very general model, the results can most likely be generalized for similar models for software development as well.
In chapter 5 the relationship between formal methods and the commonly used 'structured' methods for software development is analyzed, through examining mappings from data flow diagrams (a main design notation used in Structured Analysis (SA)) to VDM specifications, and through the construction of a prototype combined structured/formal method, called SVDM, based on one such mapping. SVDM is further illustrated in chapter 6 through a case study in which a simple lift system is developed. The main merits of SVDM are that a development path is provided which enables the construction of a formal specification in a structured way, starting using a conventional technique, and that the designer has two 'views' on the development: a graphical one (the design products of SA) and a textual one (the VDM specifications) so that depending on his experience and/or preference the most appropriate product can be used for the next development step. A disadvantage of the approach is that some of the specifications that are produced are difficult to read and are not very likely to be of much help to the designer; these specifications can, however, be regarded as a formal semantics of the data flow diagrams and are as such valuable. Further case studies and – above all – tool support are necessary to further develop and validate the use of combined structures/formal methods such as SVDM.

The research in this thesis shows that the industrial take-up of formal methods can best be pursued through a gradual introduction of formal methods in software processes. Conventional methods and techniques have proven their value in practice. Yet, at the same time, these conventional techniques can effectively be used in combination with formal methods, in this way utilizing the benefits that formal methods have to offer.
Samenvatting

Formele methoden voor software ontwikkeling (d.w.z. methoden met een wiskundige basis, geschikt voor de specificatie en bewijsbaar correcte ontwikkeling van software systemen) bieden de mogelijkheid om grootschaliger programmatuur te ontwikkelen dan tegenwoordig mogelijk is: ze bieden faciliteiten voor het uitdrukken van abstractie waar door een betere contrôle ontstaat over de complexiteit van software, ze maken consistent en niet-ambiguë software specificaties mogelijk, en ze bieden faciliteiten voor het maken van bewijsbaar correcte ontwikkelingsstappen. Het gebruik van formele methoden in de software industrie is tot nu toe echter beperkt omdat: van dit proefschrift:

- De exacte rol en plaats van formele methoden in het ontwikkeltraject van software zijn nog niet volledig bepaald. Een positionering van formele methoden in het ontwikkeltraject is noodzakelijk om de relatie tussen formele methoden en conventionele methoden en technieken voor software ontwikkeling duidelijk te maken.
- Er is onvoldoende automatisch ondersteuning (tools) van produktie-kwaliteit op de markt beschikbaar om het effectief kunnen werken met formele methoden mogelijk te maken.
- Op een paar uitzonderingen na bestaan er geen gestandaardiseerde formele methoden.
In dit proefschrift worden deze problemen aangepakt door het uitvoeren van een aantal experimenten waarbij de formele methode VDM en haar specificatietaal VDM-SL een centrale rol spelen.

In hoofdstuk 3 worden de problemen m.b.t. standaardisatie en automatische ondersteuning geanalyseerd aan de hand van de ISO standaard voor VDM-SL die momenteel wordt voorbereid. Met name worden de volgende aspecten van deze standaard onder de loep genomen:

1. **de statische semantiek** van VDM-SL. Omdat VDM-SL een zwak-getypeerde taal is waarin zgn. union types voorkomen is de notie van het type-correct zijn van een VDM specificatie niet triviaal. In dit proefschrift wordt een aanpak voorgesteld die een efficiënte doch simpele manier om een VDM specificatie op type-correctheid te controleren mogelijk maakt.

2. **de syntax afbeelding** tussen de syntax van de taal en een genormaliseerde versie van deze syntax, die gebruikt wordt voor de dynamische semantiek. Deze aanpak werd door de standaardisatiecommissie voorgesteld om de omvang van de dynamische semantiek zo veel mogelijk te beperken.

3. **de betekenis van tool conformance en language compliance** in de context van VDM-SL. VDM-SL is (in het algemeen) een niet-executeerbare taal, en daarom is het onmogelijk om vast te stellen of ondersteunende programmatuur daadwerkelijk de juiste taal-definitie hanteert. Om producenten van dit soort programmatuur tegemoet te komen wordt in dit proefschrift een meer fijnschalige definitie van de onderhavige begrippen voorgesteld.

De ISO/VDM-SL standaard bevat een compleet formele definitie van de taal. Deze aanpak beperkt echter de toegankelijkheid van de standaard en daarmee het gebruik van de taal.

In hoofdstuk 4 wordt de toepassing van VDM geanalyseerd in de context van een algemeen geaccepteerd kader voor software ontwikkeling, nl. DoD-STD-2167A. Deze analyse bestaat voornamelijk uit een literatuuronderzoek, waarbij ervaringen van VDM gebruikers geplaatst worden in de ontwikkelingsfasen zoals onderscheiden in DoD-STD-2167A, en de verschillende soorten activiteiten binnen deze fasen. De uit-
komst hem van deze analyse bevestigen een aantal van de reeds eerder vastgestelde problemen met formele methoden (gebrek aan standaardisatie, gebrek aan hoogwaardige automatische ondersteuning, onduidelijke relatie met conventionele methoden en technieken) en identificeren bovendien de fasen en activiteiten waarvoor formele methoden het beste kunnen worden toegepast binnen het kader van DoD-STD-2167A. Aangezien DoD-STD-2167A een vrij algemeen model voor software ontwikkeling representeert, is het aannemelijk dat deze resultaten eenvoudig kunnen worden gegeneraliseerd.

In hoofdstuk 5 wordt de relatie tussen formele methoden en de meer algemeen toegepaste ‘gestructureerde’ methoden voor software ontwikkeling besproken. Dit wordt gedaan d.m.v. een analyse van mogelijke afbeeldingen tussen data flow diagrammen (een van de voornaamste notaties gebruikt bij de gestructureerde methode Structured Analysis (SA)) en VDM specificaties, en door de ontwikkeling van een op één zo’n afbeelding gebaseerd prototype voor een gecombineerd gestructureerde/formele methode, SVDM genaamd. SVDM wordt in hoofdstuk 6 geïllustreerd aan de hand van een casus, waarin de ontwikkeling van een simpel lift systeem centraal staat. De voornaamste voordelen van SVDM bestaan uit het geboden ontwikkeltraject waarbij uitgaande van het gebruik van een conventionele, ‘vertrouwde’ techniek op gestructureerde wijze een VDM specificatie kan worden geconstrueerd. Daarnaast wordt de ontwikkelbaar twee ‘gezichtspunten’ op de ontwerpprodukten geboden: een grafisch-georiënteerde (de ontwerpprodukten van SA) en een tekstueel-georiënteerde (de VDM specificaties) zodat op basis van ervaring en/of voorkeur een produkt kan worden uitgekozen als basis voor een volgende ontwikkelstap. Een nadeel van de aanpak is dat sommige van de VDM specificaties die worden geproduceerd moeilijk leesbaar zijn en weinig waarde hebben voor de ontwikkelaar; deze specificaties kunnen echter worden gezien als een formele semantiek van het betreffende dataflow diagram en zijn als dusdanig wel nuttig. Het uitwerken van verdere (grotere) voorbeelden en vooral de ontwikkeling van automatische ondersteuning voor het werken met SVDM zijn een volgende stap in het onderzoek naar gecombineerd gestructureerde/formele methoden.
Het onderzoek zoals vastgelegd in dit proefschrift laat zien dat industriële acceptatie van formele methoden eenvoudiger wordt door een *geleidelijke* introductie. Conventionele methoden en technieken hebben hun waarde in de praktijk bewezen. Niettemin is het mogelijk om deze methoden en technieken te combineren met formele methoden, waardoor het mogelijk wordt om de voordelen die formele methoden bieden te benutten in de praktijk.
Curriculum Vitae


Naast zijn werk aan de Technische Universiteit Delft is hij van 1989 tot 1991 part-time werkzaam geweest als consultant bij West Consulting B.V. In deze hoedanigheid heeft hij bijdragen geleverd aan een door dit bedrijf uitgevoerd software project en heeft hij enkele malen een Ada-cursus verzorgd.

Sinds januari 1993 werkt hij bij CAP Gemini Innovation (tegenwoordig onderdeel van CAP Volmac B.V.).