Software Development Using VDM

SEI Curriculum Module SEI-CM-16-1.1

December 1989

Jan Storbank Pedersen
Computer Resources International A/S

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Education Program is developing a wide range of materials to support software engineering education. A **curriculum module** identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in *designing* a course. A **support materials** package includes materials helpful in *teaching* a course. Other materials under development include model curricula, textbooks, educational software, and a variety of reports and proceedings.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

SEI curriculum modules may be copied or incorporated into other materials, but not for profit, provided that appropriate credit is given to the SEI and to the original author of the materials.

Comments on SEI educational publications, reports concerning their use, and requests for additional information should be addressed to the Director of Education, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Comments on this curriculum module may also be directed to the module author.

Jan Storbank Pedersen
Computer Resources International A/S
Bregnerødvej 144
DK-3460 Birkerød
Denmark

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position.
It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

Software Development Using VDM

Acknowledgements

This module is based on two courses that have been offered several times by Prof. Dines Bjørner as part of a Masters of Science curriculum at the Technical University of Denmark.

I want to thank Lionel Deimel, Linda Pesante, and Bob Glass for several suggestions that helped improve both the content and the form of this module.

Contents

Software Development Using VDM

Module Revision History

Version 1.1 (December 1989)	Minor changes and updates Approved for publication
Version 1.0 (April 1988)	Draft for public review

Software Development Using VDM

Capsule Description

This module introduces the Vienna Development Method (VDM) approach to software development. The method is oriented toward a formal model view of the software to be developed. The emphasis of the module is on formal specification and systematic development of programs using VDM. A major part of the module deals with the particular specification language (and abstraction mechanisms) used in VDM.

Philosophy

During the past 10 to 15 years, a number of software development methods have emerged that stress the importance of using formal descriptions of the systems to be developed. VDM (the Vienna Development Method) is one such method, with many applications in industry [Bjørner87b, Bloomfield88].

VDM is a formal, mathematically oriented method for the specification and development of software. VDM is a model-based method. Its main idea is that of giving descriptions of software systems as models. Models are specified as objects and operations on (or functions between) objects, where the objects represent input, output, and internal state of the software system. Classes of objects are explicitly defined as so-called "domains," which correspond to types in a programming language.

VDM encourages layered, top-down development of software, based on use of abstraction at the uppermost levels of system description.

At the highest level, a specification is typically given as a rather abstract model. The objects do not capture details of representation; they are restricted to capturing only properties necessary for expressing the essential concepts of the operation of the intended software system.

More concrete descriptions (designs) are then derived by transforming (or refining) abstract objects and operations into more detailed, concrete ones.

Even though these formal descriptions could be used as a basis for proofs (proving properties of a given specification or proving that one specification actually "implements" another more abstract specification), such proofs are not generally carried out when using VDM. The reason is that the proofs would be lengthy, and generally not justifiable, for systems that solve realistic problems. Moreover, carrying out such proofs requires a deeper knowledge of the underlying mathematics than that required of a normal user of VDM.

This module deals with the software development phases normally referred to as specification and design; it does not address the preceding process of requirements analysis and the later process of maintenance. However, both these processes can benefit from the existence of formal documents. In addition, implementation is facilitated by the availability of a final, low-level formal specification.

Objectives

The following is a list of possible educational objectives based upon the material in this module. Objectives for any particular unit of instruction may be drawn from these or related objectives, as may be appropriate to audience and circumstances.

Knowledge

- Define the basic terminology of formal software development, model-oriented specification methods, and stepwise-development approaches.

Comprehension

- Explain the underlying ideas and concepts of formal software development (such as abstraction and correctness).

- Describe a number of specific formal models for well-known software concepts (like data models for database systems).

Application

- Use the specification language of VDM for writing formal specifications within an application area where domain-specific approaches exist.
- Use the principles of stepwise refinement within the framework of VDM.

Analysis

- Establish proof obligations for the formal steps in a stepwise development and prove such steps correct.

Synthesis

- Apply VDM to new application domains.
- Produce guidelines for applying VDM within new application domains.

Evaluation

- Evaluate new guidelines for applying stepwise development principles within VDM.

Prerequisite Knowledge

Students must be familiar with at least one high-level programming language (such as Pascal) and must have some programming experience using such a language. They must at least be able to utilize structured types and recursive subprograms.

Some knowledge of discrete mathematics (sets, relations, and functions) corresponding to [Stanat77] is needed. Also, one hopes that students' previous experience will have led to an appreciation of the value of mathematical abstraction.

Module Content

Outline

I. Introduction to VDM
  1. Formal Software Development
  2. Origin and Uses of VDM
  3. Overview of VDM
    a. The Specification Language Meta-IV
    b. Development Guidelines
    c. Tool Support

II. Formal Specification of Software Using VDM
  1. Meta-IV and the Construction of Abstract Models
    a. Meta-IV Type Constructors and Their Use
    b. Abstract Syntax
    c. Representational and Operational Abstraction
    d. The Components of Abstract Models
    e. Imperative Programming
    f. Applicative vs. Imperative Models
    g. Denotational vs. Mechanical Semantics
  2. Modeling of Programming Language Concepts
    a. Types and Values
    b. Variables, Storage, and Locations
    c. Blocks
    d. Subprograms and Macros
    e. Flow of Control

III. Software Design Using VDM
  1. Systematic Program Development
    a. Stepwise Development
    b. Object Transformations
    c. Operation Transformations
  2. Formal Development of Programs
    a. Proofs and Proof Systems
    b. Relations between Specification and Design
  3. Interpreters and Compilers
  4. Data Models and Database Management Systems
    a. Relational Data Models
    b. Hierarchical Data Models
    c. Network Data Models

IV. Future Directions of VDM

Annotated Outline

I. Introduction to VDM

1. Formal Software Development

VDM (the Vienna Development Method) is a method for formally developing software. A number of methods call themselves "formal," but they do not all use the word in the same way. A minimal criterion that a software development process must meet in order to be called formal is that it must lead to a set of interrelated formal documents. A formal document is one that is written using a formal language, and a formal language is one with a mathematically defined syntax and semantics. This minimal criterion is satisfied by a number of methods, including algebraic approaches like Clear [Burstall-81] and OBJ-2 [Futatsugi85]. Additional formality can be introduced by the rigor applied in the development of the formal documents. For example, a formal document being developed could be *proven* correct with respect to some prior formal document.

2. Origin and Uses of VDM

A formal method must include a formal language—often referred to as a *specification language*. In the case of VDM, the specification language is called Meta-IV [Bjørner78]. The name came from the first application of VDM. Meta-IV (and VDM) was first applied in the early seventies at the IBM Vienna Research Lab to give a formal semantics definition of a large PL/I subset [Bekic74]. A language that is used in defining the semantics of another language is often referred to as a *metalanguage*. The "IV" in Meta-IV has no real meaning—Meta-IV does not have three predecessors; instead, its name is a play on the word "metaphor."

VDM has since been used in the definition of the formal semantics of a number of programming languages including CHILL [Haff80], Modula-2 [Andrews88], and Ada [Bjørner80c, Botta87, Astesiano87]. Some of those definitions have been used in the systematic development of compilers following the VDM approach [Bjørner80a, Oest86]. Outside the areas of programming language semantics and compiler construction, VDM has been employed in a number of areas, such as:

- database models [Bjørner82a, Bjørner82c]
- office system models [Bundgaard81]
- tools for VDM itself [Crispin87, Hansen85]
- electronic mail systems [Crispin87]

- communication systems [Crispin87, Letschert87]
- communication protocols [Pedersen88]
- test-case selection [Scullard88]
- graphical kernel systems [Ruggles88]

3. Overview of VDM

a. The Specification Language Meta-IV

Meta-IV can be called a "wide-spectrum" language, in that it allows one to write specifications at different levels of abstraction. This means that it can be used as the (single) specification language throughout a number of steps in the development process. For a discussion of wide-spectrum languages see [Bauer82] and [CIP85].

Meta-IV has a model-oriented view of the world. A model-oriented specification explicitly defines the mathematical objects and operations used to describe the software system. The models are defined using a number of type definitions (for the objects) and function definitions (for the operations). This is different from the algebraic approach to specification, where the models (algebras) are implicitly defined by the properties captured in the axioms of the algebraic specifications. Expressing specifications in terms of explicit models seems to be easier for many software engineers than using algebraic specifications with axioms, especially for large systems that would require many axioms.

Meta-IV is aimed at supporting abstraction in writing specifications. Abstraction is obtained through mathematical concepts, such as sets and functions, rather than through the mechanisms offered by any particular implementation language. The abstraction provided by Meta-IV is not oriented toward any particular application area, but rather offers a set of mathematically based primitives that allow the construction of application-specific models.

A detailed presentation of Meta-IV is given in [Bjørner78].

b. Development Guidelines

Abstraction plays a central role in VDM. The principle of abstraction is applied both to the definition of objects (data structures) and operations (functions applicable to data structures).

Following a requirements analysis, the first abstract formal specification is developed. This specification describes the objects and the operations of the system. The guidelines of VDM identify general components of such specifications (see section II.1.d) as well as domain-specific standard modeling techniques (as described in sections II.2 and III.4).

Based on a high-level description of objects and operations, more concrete descriptions (designs) are developed in a stepwise fashion, still using Meta-IV as the specification language. The constructs used in the models are refined so as ultimately to resemble those found in the final implementation language. In this process, aspects such as efficient implementation of data structures and avoidance of recursion (if required) are taken into account.

The guidelines described in [Jones80] and [Jones86] cover the development of the first abstract specification and subsequent increasingly detailed designs. They do not apply to requirements analysis, testing, or maintenance. The stepwise development approach used in VDM can be seen as a special case of a waterfall life cycle model.

The only kinds of systems for which a VDM development is not appropriate are those that can be created by (proven) generators. Examples are programming language parsers (generated by parser generators) and systems developed using fourth-generation languages (high-level declarative descriptions). VDM is inappropriate simply because one should not develop systems "by hand" if efficient and correct implementations can be automatically generated from adequate high-level descriptions. Currently, however, this method of development is only possible within limited application areas.

c. Tool Support

Tools supporting the use of VDM have been developed only recently. Most of the tools are related to Meta-IV and the handling of formal specifications only, and do not provide direct support of the development process [Hansen85]. They include:

- editors for writing Meta-IV specifications
- syntax analyzers
- context condition checkers (type-checkers, etc.)
- databases for formal specifications
- output tools for screen and paper

A general discussion of environments supporting VDM can be found in [Jones87].

II. Formal Specification of Software Using VDM

1. Meta-IV and the Construction of Abstract Models

This section introduces the specification principles that are applied when using VDM and the specification language constructs used when writing speci-

fications. The composite data types of Meta-IV (sets, lists, mappings, and trees) are defined, and their proper use is illustrated by examples. The difference between several levels of abstraction is discussed by contrasting applicative and imperative models. Material covering these topics can be found in [Bjørner78], [Bjørner82b], [Cohen86], and [Jones80].

a. Meta-IV Type Constructors and Their Use

Meta-IV contains a number of primitive types, such as Booleans and integers, and type constructors for defining composite types. The classes of composite types include sets, tuples (sequences), and functions; these classes correspond to the traditional mathematical objects. Moreover, map- and tree-types are constructible, as described below.

Maps are one of the most extensively used data types of Meta-IV. They can be viewed as special functions that have one parameter and whose domain is finite. Maps are typically used to describe sets of uniquely identifiable objects, *e.g.*, the records of a file in a direct-access file system.

Cartesian products, or *trees* as they are normally called in Meta-IV (since the composite value can be viewed as the root and the components as the branches of a tree), are similar to records or structures in programming languages, *i.e.*, they have a fixed structure with a number of components of (possibly) different types. They are used to define both structured input to a system (such as commands with several parameters) and the internal structure of the system.

b. Abstract Syntax

An *abstract syntax* is a set of so-called domain equations that define classes (domains) of objects. It can be seen as a collection of mutually dependent type definitions. Abstract syntaxes provide the means for *combining* the composite types mentioned earlier.

c. Representational and Operational Abstraction

Abstract definitions of data objects and the operations on such objects are central to VDM.

- *Representational abstraction* is the abstraction applied in defining the types of data objects. Representational abstraction focuses on information content and ignores the physical layout (representation) of data.
- *Operational abstraction* is the abstraction applied in defining the operations (or functions) needed to manipulate data objects. Operational abstraction focuses on the effect of the operation and ignores the algorithmic details of how the effect is obtained.

d. The Components of Abstract Models

When using VDM, an abstract model traditionally contains three components:

- Semantic domains (defining the state of the system) and invariant predicates (defining conditions that must be satisfied for each object belonging to a semantic domain).
- Syntactic domains (defining the information contents in the commands, etc., of the system) and well-formedness predicates (defining the conditions under which a given object belonging to a syntactic domain can be given a meaning by the semantic functions).
- Semantic functions (defining the meaning of objects from the syntactic domains using objects from the semantic domains).

e. Imperative Programming

An *imperative* (specification) language is a language based on the concepts of state and state changes. The *state* is constituted by the current values of all variables. The language offers assignable variables for designating parts of the state and statements for changing the state. Since Meta-IV is intended as a wide-spectrum language, it needs to have not only high-level applicative constructs, but also lower-level imperative constructs such as declared variables and assignment statements. Including such imperative features ensures that, when a series of formal specifications is produced in order to finally arrive at a program, the last step of writing the actual program is not too far, conceptually, from the last formal specification.

f. Applicative vs. Imperative Models

Functional or applicative specification is basically specification without an underlying state, and hence without variables whose value can change. This leads to a particular style of specification, where functions (which may be recursive) and parameters are the fundamental means of expressing the behavior of the system. In VDM, this style is encouraged in the higher-level models. To avoid having common parameters carrying state information for every function in a specification, however, it is often convenient to use imperative mechanisms. And since the expressive powers of the applicative and imperative models are the same, it is largely stylistic concerns that determine the choice between them.

g. Denotational vs. Mechanical Semantics

The principles of denotational semantics have traditionally been applied primarily in the formal definition of programming languages [Stoy77]. The denotational approach, in that case, gives each identifier that occurs in the program an associated denotation (meaning), usually a function; and the semantics of a composite construct is expressed as a function of the semantics of its constituent parts. This approach, however, can also be applied to systems in general, *e.g.*, database systems. Denotational models are normally used in the early stages of systems development, since the denotations are based on functions and functional composition.

Mechanical semantics, also called computational or operational semantics, describes *how* to obtain the semantics (values) of a language construct [Plotkin81]. Generally, a mechanical semantics describes a sequence of abstract system states, thereby being more algorithmic than a denotational semantics. Often, due to the explicit use of intermediate states, a mechanical semantics will use an imperative style, allowing the use of the metalanguage state to capture the system state. Models using mechanical semantics are usually applied in the later stages of systems development.

2. Modeling of Programming Language Concepts

In VDM, modeling techniques similar to those used when describing high-level programming language concepts have been found applicable to a wide class of problems outside the field of formal programming language semantics. One reason for this is that most systems employ some kind of language (if not a "real" programming language) by which the user communicates with the system. Such languages often embody concepts found in traditional programming languages, like *types* and the requirement that certain expressions denote values belonging to particular types. The purpose of the following sections is to discuss the modeling of individual programming language concepts and the general applicability of the techniques. (See [Jones82] for a detailed description of the VDM approach to modeling programming language concepts.) It is important to keep in mind that the programming language concepts are not seen as a means of *implementing* a VDM model (in the final development step) but as concepts similar to those found in the system to be developed.

a. Types and Values

A type identifies a set of values, and introducing the concept of types into a system leads to a partitioning of all values into subsets (integers, files, etc.). A type is also characterized by the operations that are applicable to the identified values (arithmetic operations for integers, file operations like "open" and "read" for files, etc.). Hence, expressions involving operators must respect the limitations on the applicability of the operations. Expressing those constraints is often referred to as "type checking." In VDM this is expressed by the following:

- Syntactic domains defining the classes of syntactically correct expressions and type definitions.
- Semantic domains defining the classes of descriptors for expressions and types (atomic as well as composite).
- Functions defining the descriptors corresponding to a given expression or type definition.
- Functions defining denotationally (by a Boolean result) whether a given expression or type definition (object belonging to a syntactic domain) is correct, using the descriptors defined by the semantic domains.

These four parts are always used in the formal description of programming languages having types. For any specific language, the definition of the syntactic domains is almost trivial (similar to a BNF grammar); the major decision is how to define the descriptors (semantic domains). VDM has a number of standard ways of defining descriptors, depending on the type equivalence rules of a language (name equivalence, structural equivalence, etc.).

Most systems have a concept of types and enforce rules related to types. This means that the above approach is applicable not only when describing the (static) semantics of programming languages, but for other systems as well.

b. Variables, Storage, and Locations

Imperative programming languages like COBOL, FORTRAN, Pascal, and Ada have, as one of their basic concepts, *assignable variables*, along with the related notion of *changing* the values of these variables. Having variables naturally leads to the idea of a *storage* (for the current values) and *locations* (for holding the values in storage). When using VDM, the storage is always described as a mapping from locations to values, and choosing an appropriate storage model is a matter of determining the different forms of locations and values, depending on the characteristics of the programming language. More specifically, the presence or absence of programming language constructs for manipulating subcomponents of structured objects determines whether locations and the stored values should be described as

atomic or composite. Decisions are made in a way that makes the storage operations of *allocation*, *assignment* or *update*, getting the *contents* of a location, and *freeing* locations as simple as possible. These operations are often found in other state-based systems, thereby allowing for a wider use of the approach.

c. Blocks

Blocks in programming languages allow the local introduction of new entities and names for these entities. This *locality* means that the entities (as well as their names) are only to be used within the block. From a formal semantics point of view, the important thing to capture is the *visibility* of the declared names, including their *binding* to the declared entities. In VDM, the binding is described using mappings. The effect of nesting blocks is described by passing the mappings as parameters from one Meta-IV function "down" to the subfunctions dealing with the nested blocks. This scheme utilizes the normal parameter-passing techniques for functional languages, since they apply to Meta-IV functions. In imperative languages, a block often has an effect on the global state—the values of global variables may have changed as a result of executing the block. In order to describe this while still allowing local variable declarations to hide global ones (new bindings), a general scheme involving two mappings is used. The first one, the *environment*, binds variable names to locations; the second one, the *storage*, binds locations to values. This scheme not only handles blocks and their nesting, but it is capable of describing essential parts of parameter passing for subprograms and the use of alternative names for entities (renaming or aliases).

d. Subprograms and Macros

Subprograms, whether applicative or imperative, all share the property that nonlocal names referenced within them are bound at the point of the definition of the subprogram. This is in contrast to macros, for which such names are bound at each point of call.

Calling an applicative subprogram must generate a value (since no side effects are possible). Since all global names are bound at the point of subprogram definition, a natural "meaning" of such a subprogram in VDM is: a function that, given any actual parameters (values), returns a value. This is the kind of denotation that one associates in VDM with the name of such a subprogram in the environment.

Imperative subprograms are also described as functions, but they need the value of global variables at the point of call. Hence, they are described by functions from actual parameters and a storage to an optional value (absent in the case of "pure" procedures) and to a potentially changed storage (due to side effects).

Macros, due to their name-binding rules, are described as functions from actual parameters, an environment (name bindings), and a storage to an optional value and a potentially changed storage (due to side effects).

e. Flow of Control

This section deals with GOTOs and other programming language constructs requiring an abnormal (nonsequential) flow of control during program execution.

There are traditionally two ways of abstractly describing such program constructs [Jones78, Bjørner80b]. One is the direct semantics style, or the **exit** style, and the other is the continuation style. In VDM, both approaches are applicable, but the **exit** style is preferred, since it matches the intuition of most software engineers.

III. Software Design Using VDM

The previous section discussed the VDM approach to formally specifying a system. This section addresses the process of moving from abstract specifications to more implementation-oriented ones. Such implementation-oriented specifications are what we call *designs*. The process typically involves defining a series of designs that progressively approach the implementation.

A key issue in VDM is the relation between two formal specifications, where one is supposedly "implementing" the other. The degree to which this implementation relation should be demonstrated is not prescribed by VDM. Instead, each project should adopt an appropriate level of formality in establishing this relationship.

The following sections describe the different degrees of formality that can be applied in the design process. They discuss the application of the VDM principles to interpreters and compilers, as well as to database systems.

1. Systematic Program Development

When starting with the initial high-level Meta-IV specification of a system, one has a specification that expresses the system's functionality, but pays little or no attention to the nonfunctional requirements. Moreover, a number of specification language constructs used (such as sets, mappings, and implicit construction of values) are most likely not supported by the particular implementation language to be used. By providing examples as in [Bjørner82b] and [Jones80], VDM offers guidelines for selecting parts of an abstract specification and

for creating a new formal specification in which those parts have been made more concrete.

a. Stepwise Development

Designing a system based on a specification corresponds to making a number of commitments on behalf of the implementation. The idea behind a stepwise development process is that all commitments should not be made at one time, but should be made sequentially.

In the VDM framework, most commitments take the form of an object transformation (selecting new data representations) or an operation transformation (choosing more algorithmic definitions). Object transformations often necessitate certain operation transformations, since the original operations used the characteristics of the corresponding data type. Also, because most VDM specifications aim at being compositional (*i.e.*, isolating properties of components of a model), parts of a specification can be developed in isolation.

b. Object Transformations

The most abstract formal specifications extensively use data types such as sets and maps, which are not present in most implementation languages. Even in cases where these types are almost directly implementable, the direct implementation might not be efficient enough for the system to be developed. Hence, for each high-level data type, a more "implementable" one must be chosen, depending on the characteristics of the implementation language and the nonfunctional (*e.g.*, space or time) requirements of the system. (For a mapping, a table or an ordered binary tree might be chosen.) Of course, knowledge of existing efficient algorithms available for different data structures, as presented in [Horowitz76], [Horowitz78], and elsewhere, plays a major role in determining the data structures. Meta-IV, however, allows one to define such data structures at an abstract level without getting into the details of their implementation. (For example, ordered binary trees can be defined using recursive data types without thinking about the use of "pointers," etc., for dynamic allocation.)

c. Operation Transformations

In addition to the operation transformations incurred by the object transformations, one might want to define an operation in a more efficient or implementable way, considering implementation language capabilities.

If, for example, an iterative operation structure is preferred over a recursive one, this will lead to an imperative formal specification (see II.1.f-g). A number of existing standard schemes for removing recursion have been adopted by VDM (such as [Darlington76] and [Burstall77]).

2. Formal Development of Programs

a. Proofs and Proof Systems

Since testing cannot, in general, guarantee the correctness of a program (or of a formal specification), a proof is the only means of guaranteeing correctness. Proofs can be either formal or informal. A formal proof is an argument constructed by symbol manipulations according to a set of inference rules. An informal proof contains the major steps of the argument but avoids the more tedious details of symbol manipulation. An informal proof can be made formal by filling in all the details. Most proofs carried out in VDM are informal proofs, and they follow the inference rules of the predicate calculus [Jones86].

b. Relations between Specification and Design

In this section we see a design as a formal model *implementing* what is defined by a specification. The relation between objects in the design and corresponding objects in the specification is formalized in VDM by a number of *retrieve functions* that, for each object in the design, yields the corresponding object in the specification [Jones86]. They are called retrieve functions because they regain the abstraction from the implementation. In general, the relation between objects in the specification and those in the design is a one-to-many relation, in that several design objects all represent the same abstract object. It is, however, still important to show the *adequacy* of a design with respect to its specification. A design is adequate if there is at least one representation of every abstract value in the specification.

Once a design has been shown to be adequate, one can start proving it correct. First, the *proof obligations* are established. They express that operations in the design must behave analogously to those of the specification, thus preserving the properties of the specification. Second, the proof itself is carried out using the definition of the retrieve functions. In practical examples, one often only establishes the proof obligations, without carrying out the actual proofs, since such proofs are cumbersome, and most design errors are revealed by establishing the proof obligations alone.

3. Interpreters and Compilers

One of the areas in which VDM has been used most extensively is the development of compilers for programming languages [Bjørner77, Bjørner80a, Bjørner82e, Oest86]. Interpreters and compilers are similar, in that they they both implement the seman-

tics of the programming language, the first by "direct" execution and second by a translation into a target language. Both require a deep understanding of the semantics of the programming language. Hence, the first step in a VDM development of a compiler is the formal definition of the programming language semantics. Such a semantics consists of the following components [Pedersen87]:

- an abstract syntax called AS1
- a static semantics (well-formedness) definition based on AS1
- an abstract syntax AS2
- an AS1 to AS2 transformation
- a dynamic semantics based on AS2

The abstract syntax AS1 describes a grammar of the programming language using Meta-IV type definitions. The static semantics defines the context conditions, *i.e.*, nonsyntactic language rules whose violation leads to illegal programs that are to be rejected by a compiler (or interpreter). The abstract syntax AS2 describes the syntax of statically correct programs. It resembles AS1, but certain changes are introduced to facilitate the definition of the dynamic semantics. The AS1 to AS2 transformation formalizes the relation between AS1 and AS2. The dynamic semantics describes the runtime behavior of statically correct programs.

From a compiler construction point of view, AS1 and AS2 correspond to internal representations of a program during compilation. The static semantics corresponds to the front end (semantic analyzer) of a compiler, and the dynamic semantics corresponds to a combination of the back end (code generator) and the runtime system.

The most abstract formal semantics of a programming language is expressed in VDM using a denotational semantics style. In the case of languages with parallelism, a similar but more elaborate scheme is used [Haff80, Bjørner80c, Astesiano87].

Based on this initial formal definition of the programming language semantics, a series of more concrete definitions is often developed, depending on the complexity of the programming language and the capabilities of the implementation language of the interpreter or the target language for the compiler [Bjørner82e]:

- a first-order applicative semantics (no functions as semantic domains)
- an abstract state machine semantics (using high-level Meta-IV types)
- a concrete state machine semantics (using more implementable types)

The next step in the case of compiler development is to define a so-called *compiling algorithm* that associates a sequence of abstract instructions with each language (AS2) construct. The abstract instructions reflect the physical instructions of the target computer. A formal description of the target system is made using these abstract instructions and an abstraction of its state. The compiling algorithm is usually defined by a set of mutually recursive Meta-IV functions using a "dictionary" to hold the necessary context information.

Depending on the language in which the compiler is to be implemented, a series of formal specifications for the compiler itself are developed. Often an attribute semantics definition [Bjørner82e] of the compiling algorithm is given. Looking at synthesized and inherited attributes assists in the analysis of different multi-pass or single-pass compiler designs.

Recently, the guidelines for developing compilers using VDM have been further formalized. For an example of such a formalization for the development of an Ada compiler, see [Oest86].

4. Data Models and Database Management Systems

A data model defines data objects and operations applicable to the data objects. A database management system supports the storing of data objects and the execution of the operations. The functions of a database are divided into meta-functions that operate on descriptions of data (such as database schema operations) and functions that operate on the data objects themselves (such as queries and general data manipulation).

In the following sections, VDM formalizations of different known data models are introduced. Further information can be found in [Bjørner82c].

a. Relational Data Models

In VDM, a relational data model is seen as a collection of named relations. These relations are described using maps. A relation is seen as a set of rows, and a row as a tuple of values or a mapping from row names to values. Based on this model, it is easy to define the traditional relational database operations of *select*, *project*, *join*, and *divide*. General predicate calculus query languages, such as SQL, are easily described also [Bjørner82c].

b. Hierarchical Data Models

Basic concepts in a hierarchical data model are those of *records* (of particular *record types*), which are arranged in *tree structures*. The hierarchy among record types is part of the *schema* of the data model. This is described in VDM as a hierarchy-diagram type that is recursively defined as a mapping from record type identifiers to hierarchy-diagram types (the sub-hierarchies of

the record type); the mapping is empty for the bottom elements of the hierarchy. All one has to add to this are the descriptors of the actual fields in the record types (at each level) in order to get the complete schema information. Generally, throughout the formal description of the hierarchical data model, recursively defined mappings are used to capture hierarchical tree structures, rather than using Meta-IV trees, which would lead to a fixed number of sub-hierarchies for each record type.

c. Network Data Models

The basis for a VDM description of network data models is the formalization of Bachman's concept of data structure diagrams [Bachman69]. Such a diagram consists of boxes and arrows. A box denotes a set of records, and it is described as such in VDM. An arrow denotes a relation among records and is described as a mapping from records to sets of records (a one-to-many relation). The *syntactic* form of any particular data structure diagram can be seen as a set of unique box names and a set of uniquely named arrows (a mapping from arrow names to pairs of "from" and "to" box names). One gets a model of data structure diagrams by combining this syntactic view with the above semantic view (of what boxes and arrows *denote*) and adding the necessary invariants. The model allows one to describe easily the concept of "navigating" through the database.

IV. Future Directions of VDM

Even though VDM has proven successful in a number of applications and is one of the more mature formal system development methods, there is still room for improvement. In particular, lessons learned from other formal description techniques, such as the algebraically based ones (*e.g.*, regarding structuring and parameterizing specifications) suggest ideas for extensions to the specification language [Bear88]. Moreover, a way of describing parallel systems should be introduced as an integral part of the specification language. The formalization of software development projects in the form of software development graphs, as described in [Bjørner86], is expected to become an essential part of VDM. A project graph identifies the formal documents to be developed as part of a project and describes their relationship. Each project has its own project graph, but projects within a particular problem domain, such as compiler development, have graphs that exhibit similar structures. Finally, a more extensive tool support is needed to ensure a wider industrial use of formal methods. One project that currently aims at solving those problems is the RAISE (Rigorous Approach to Industrial Software Engineering) project [Nielsen88]. It can be seen as the next generation of VDM-like development methods. [Prehn87] discusses general improvements to VDM in the context of RAISE.

Teaching Considerations

The instructor should be familiar with formal methods in general; SEI curriculum module *Formal Specification of Software* [Berztiss87] provides an overview of the appropriate information.

Suggested Schedules

The material covered by this module can be taught at various depths. At the Technical University of Denmark, the material has been presented in two full courses: one covers essentially Part II, "Formal Specification of Software Using VDM," and some of the mathematical prerequisites; the other covers Part III, "Software Design Using VDM," including a thorough treatment of examples of applying VDM to large systems. Below is a sample schedule for a one-semester course (34 lecture hours):

I. Introduction to VDM (1 hour)

II. Formal Specification of Software Using VDM

1. Meta-IV and the Construction of Abstract Models (9 hours)
2. Modeling of Programming Concepts (7 hours)

III. Software Design Using VDM

1. Systematic Program Development (6 hours)
2. Formal Development of Programs (4 hours)
3. Interpreters and Compilers (3 hours)
4. Data Models and Database Management Systems (3 hours)

IV. Future Directions (1 hour)

For a short 1- to 2-hour overview of VDM to be presented as part of a course in software specification and development, [Bjørner79] presents a concise introduction to VDM, including examples suitable for illustrating the central ideas of abstract specification and stepwise refinement in VDM.

Worked Examples

Examples of VDM applications play an important role in teaching the proper use of VDM. The following are references to literature containing examples appropriate for classroom presentations.

There are a number of useful examples in [Bjørner78]: small examples of the use of individual Meta-IV language constructs; a formal specification of Algol 60; and an example of a formal specification of an operating system command language.

[Bjørner79] includes an example of object refinements applied to the specification of a file system.

[Bjørner82b] contains examples of VDM applications within these areas:

- programming language semantics
- compiler development
- stepwise development in general, illustrated by a file system example
- data models and database systems

[Jones80] has examples of stepwise development of a parser for a small language and a telegram analysis program.

Exercises

[Jones80] and [Jones86] provide exercises in the use of VDM for constructing formal specifications, as well as exercises in the systematic development of programs including proofs of correctness.

The following exercises can be used to demonstrate the proper use of Meta-IV, corresponding to [Bjørner78]:

1. Define, using Meta-IV, bounded stacks with the traditional operations (Empty, Push, Pop, Top, Is-Empty, Is-Full). In addition to type definitions and operations (functions), define invariants and preconditions, or justify not using them.

2. Define an abstract syntax for arithmetic integer expressions, including (at least) unary minus, dyadic minus, plus, and multiplication. The abstract syntax must cover all possible expressions even those that in a concrete (BNF-like) syntax make use of parentheses, for example, –(5+7)·(8–3). Define a function that evaluates such expressions.

3. Define a system for keeping information about students in a school. For each student, the system should keep track of the exams passed by that student. The system does not have to account for exams not passed. Define operations corresponding to:
   1. A student's passing an exam
   2. Determining whether a given student has passed a particular exam
   3. Getting all the exams that a given student has passed
   4. Getting all the students that have passed a particular exam

4. This exercise illustrates the process of stepwise refinement. The step consists of creating a more implementable specification (using a set of pairs) from an abstract specification (using a map). It shows two different levels of abstraction (that of a mapping and that of a set of pairs) and the formal relation between them (see section III.1).

   Assume that we want to "represent" a function (or mapping) from integers to integers as a set of pairs:

   ```
   SET-FUNC = PAIR-set
   PAIR     :: s-argument: INTG   s-result: INTG
   ```

   1. Define an invariant (function) expressing that a value of type SET-FUNC actually represents a function and not a general relation.
   2. Define a (retrieve-) function that, given a SET-FUNC object, yields the corresponding map object.
   3. Define operations on such function representations corresponding to the Meta-IV "map" operations of: application, domain, range, override (+), and restriction (\). These operations should be defined without using the retrieve operation mentioned in (2).

5. This exercise also illustrates stepwise refinement. Here a binary tree is chosen as the "more implementable" specification.

   We want to "represent" a function (or mapping) from integers to integers as an ordered binary tree.

   The operations to be defined on the binary tree should be defined using operations pertaining to that data type. The operations mentioned in (4) and (5) should be defined without using the retrieve operation mentioned in (3).
   1. Define an appropriate type (TREE-FUNC).

2. Define an invariant (function) expressing the fact that a given binary tree is indeed ordered and represents a function.
3. Define a (retrieve-) function that, given a TREE-FUNC object, yields the corresponding map object.
4. Define operations corresponding to: application, domain, and overriding with *one* new integer-to-integer association (type: simple-override: TREE-FUNC INTG INTG → TREE-FUNC).
5. Define operations corresponding to general overriding (type: override: TREE-FUNC TREE-FUNC → TREE-FUNC) and restriction (type: restrict: TREE-FUNC INTG-set → TREE-FUNC).

Suggested Reading Lists

The following table categorizes items in the bibliography by applicability. References enclosed in parentheses refer to parts of other references (see the bibliography for further details). *VDM Textbooks* provide material from which a course can be taught. The *Recommended* readings consist of a number short papers on VDM and related material. *Detailed* readings discuss technical details of VDM or are examples of major applications of VDM. The *Background* readings cover general non-VDM material that is useful, but not essential, to understanding and applying VDM.

VDM Textbooks	Recommended	Detailed	Background
Bjørner78	(Andrews88)	Astesiano87	Bachman69
Bjørner82b	Berztiss87	(Bear88)	Bauer82
(Bjørner82c)	Bjørner77	Bekic74	Burstall81
(Bjørner82d)	Bjørner79	Bjørner80c	CIP85
(Bjørner82e)	Bjørner80b	(Bjørner80a)	Futatsugi85
Cohen86	Bjørner82a	Bjørner87b	Horowitz76
Jones80	Bjørner86	Bloomfield88	Horowitz78
(Jones82)	Bjørner87a	Botta87	Plotkin81
Jones86	Burstall77	Bundgaard81	Stanat77
	(Crispin87)	Haff80	Stoy77
	Darlington76	Hansen85	
	Oest86	(Jones78)	
	(Prehn87)	(Jones87)	
	(Scullard88)	(Letschert87)	
		(Pedersen87)	
		Pedersen88	
		(Ruggles88)	

Bibliography

Andrews88

Andrews, D. J., *et al*. "The Formal Definition of Modula-2 and Its Associated Interpreter." In *VDM '88: VDM—The Way Ahead*, R. Bloomfield, *et al.*, eds. Lecture Notes in Computer Science, vol. 328. Berlin: Springer-Verlag, 1988, 167-177.

> ***Abstract:*** *A three year research project is currently being undertaken at Leicester Universtiy, The National Physical Laboratory (NPL) and The British Standards Institute (BSI). The project aims to produce a formal definition of the syntax and semantics of the programming language Modula-2, written in VDM Meta IV, together with a rigorously verified interpreter derived directly from the definition. In the process of producing a good quality document of the formal definition of Modula-2, two by-products will also be developed and applied. They are a VDM structure editor and an enviornment to generate LaTeX files from the VDM structure editor.*

Astesiano87

Astesiano, E., *et al*. *The Draft Formal Definition of Ada: The Dynamic Semantics Definition.* Dansk Datamatik Center/CRAI/IEI/University of Genoa, Lyngby, Denmark, January 1987.

> This is a five-volume description of the dynamic (runtime) behavior of Ada programs. It is part of the results of a project sponsored by the Commission of the European Communities (CEC) Multi-Annual Programme: "The Draft Formal Definition of Ada."

Bachman69

Bachman, C. W. "Data Structure Diagrams." *Data Base 1*, 2 (Summer 1969), 4-10.

> This is the original paper on data structure diagrams and their application to data models and databases.

Bauer82

Bauer, F. L., and H. Woessner. *Algorithmic Language and Program Development*. Berlin: Springer-Verlag, 1982.

> This includes a discussion of the use of wide-spectrum languages in program development.

Bear88

Bear, S. "Structuring for the VDM Specification Language." In *VDM '88: VDM—The Way Ahead*, R. Bloomfield, *et al.*, eds. Lecture Notes in Computer Science, vol. 328. Berlin: Springer-Verlag, 1988, 2-25.

> ***Abstract:*** *This paper describes a stucturing scheme for the VDM Specification Language. A VDM* document *may be split into a number of* modules *which may be parameterised. Modules may* import *and* export *constructs. A parameterised module may be* instantiated *by another module. We define an abstract syntax and give a compositional denotational semantics. Context Conditions are discussed informally, but are not set out in any detail.*

Bekic74

Bekic, H., *et al*. *A Formal Specification of a PL/I Subset.* TR 25.139, IBM Vienna Laboratory, Vienna, 1974.

> The first major application of VDM to the description of programming language semantics. It is the application for which Meta-IV was invented.

Berztiss87

Berztiss, A. *Formal Specification of Software.* Curriculum Module SEI-CM-8-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., October 1987.

> ***Capsule Description:*** *This module introduces methods for the formal specification of programs and large software systems, and reviews the domains of application of these methods. Its emphasis is on the functional properties of software. It does not deal with the specification of programming languages, the specification of user-computer interfaces, or the verification of programs. Neither does it attempt to cover the specification of distributed systems.*

Bjørner77

Bjørner, D. "Programming Languages: Formal Development of Interpreters and Compilers." *5th Intl. Comp. Symp.* Amsterdam: North-Holland, April 1977, 1-21.

> This paper demonstrates seven stages of systematic VDM development, and specifies and transforms a formal definition of a simple, applicative language with recursion into an attribute grammar specification for a compiling algorithm.

Bjørner78

Bjørner, D., and C. B. Jones, eds. *The Vienna Development Method: The Meta-Language*. Lecture

Notes in Computer Science, vol. 61. Berlin: Springer-Verlag, 1978.

This book includes a tutorial on Meta-IV, a reference manual for Meta-IV, and a formal definition of Algol 60.

Bjørner79

Bjørner, D. "The Vienna Development Method (VDM): Software Specification & Program Synthesis." In *Mathematical Studies of Information Processing*, E. K. Blum, M. Paul, and S. Takasu, eds. Lecture Notes in Computer Science, vol. 75. Berlin: Springer-Verlag, 1979, 326-359.

Abstract: *A capsule view is given of the VDM specification language and the associated specification techniques for defining software, respectively the systematic derivation techniques for synthesizing and proving correct program realizations from such abstract software architectures.*

The paper exhibits examples illustrating abstract syntax specifications of both abstract and derived concrete syntactic and semantic domains, and denotational and derived operational elaboration function definitions mapping syntactic domain objects into their semantic domain object denotations, respectively into operations on these. In deriving the concrete programs from abstract definitions, and in proving correctness, extensive use is made of invariant (-preserving) static and dynamic well-formedness predicates and retrieval (or: abstraction) functions bringing concrete, realization-oriented objects "back" into their defining abstract objects. Such uses are likewise illustrated. Examples of proofs based on the idea of commuting diagrams follow. These make use of a number of data structure lemmas: properties of the abstract and concrete objects chosen to represent, respectively realize, the specified software concepts. We finally exemplify the beginnings of such a catalogue of auxiliary lemmas.

Bjørner80a

Bjørner, D., and O. N. Oest. "The DDC Ada Compiler Project." In *Towards a Formal Description of Ada*, D. Bjørner and O. N. Oest, eds. Lecture Notes in Computer Science, vol. 98. Berlin: Springer-Verlag, 1980, 1-20.

The paper describes how VDM was expected to be applied in the development of the DDC Ada compiler (see also [Oest86]).

Bjørner80b

Bjørner, D. "Experiments in Block-Structured GOTO-Modelling: Exits vs. Continuations." In *Abstract Software Specifications*, D. Bjørner, ed. Lecture Notes in Computer Science, vol. 86. Berlin: Springer-Verlag, 1980, 216-247.

The paper illustrates a spectrum of so-called **exit** and continuation semantics definitions, including combinations of both.

Bjørner80c

Bjørner, D., and O. N. Oest, eds. *Towards a Formal Description of Ada*. Lecture Notes in Computer Science, vol. 98. Berlin: Springer-Verlag, 1980.

This book contains the following:

- a description of the DDC Ada compiler project (see [Bjørner80a])
- a denotational definition of the static semantics (context conditions) of Ada
- a formal description of the dynamic semantics (runtime behavior) of the sequential (non-parallel) parts of Ada
- a formal definition of parallelism in Ada
- a definition of an abstract machine for executing Ada programs

Bjørner82a

Bjørner, D., and H. H. Løvengreen. "Formal Semantics of Data Bases." *8th Intl. Conf. on Very Large Data Bases.* New York: ACM, September 1982.

The paper advocates the use of formal specification in the design of new data models and in the recording of old ones. It contains an extensive model of the IMS database management system.

Bjørner82b

Bjørner, D., and C. B. Jones. *Formal Specification and Software Development*. Englewood Cliffs, N. J.: Prentice/Hall International, 1982.

This book contains a number of papers covering:

- the meta-language of VDM
- application of VDM to programming language semantics (see [Jones82])
- application of VDM to compiler development (see [Bjørner82e])
- stepwise development using VDM
- formalization of database models (see [Bjørner82c])
- development of database management systems (see [Bjørner82d])

Bjørner82c

Bjørner, D., and H. H. Løvengreen. "Formalization of Data Models." In *Formal Specification and Soft-*

ware Development, D. Bjørner and C. B. Jones, eds. Englewood Cliffs, N. J.: Prentice/Hall International, 1982, 379-442.

This paper includes VDM formalizations of the relational data model, the hierarchical data model, and the network data model.

Bjørner82d

Bjørner, D. "Realization of Database Management Systems." In *Formal Specification and Software Development*, D. Bjørner and C. B. Jones, eds. Englewood Cliffs, N. J.: Prentice/Hall International, 1982, 443-458.

Just as VDM can be used to justify the design of an interpreter or compiler with respect to the definition of the language, this paper illustrates how a database management system implementation can be related to the specification of a data model. Implementations of both hierarchical and network architectures are discussed.

Bjørner82e

Bjørner, D. "Rigorous Development of Interpreters and Compilers." In *Formal Specification and Software Development*, D. Bjørner and C. B. Jones, eds. Englewood Cliffs, N. J.: Prentice/Hall International, 1982, 271-320.

This paper shows the development of compilers and interpreters for a simple, applicative language that includes functions as values. The intermediate steps use imperative constructs of the meta-language (*e.g.*, declarations). The example is used to illustrate the step from language definitions to compiling algorithms.

Bjørner86

Bjørner, D. "Software Development Graphs—A Unifying Concept for Software Development?" In *Foundations of Software Technology and Theoretical Computer Science, 6th Conf., New Delhi, India, December 18-20, 1986*, K. V. Nori, ed. Lecture Notes in Computer Science, vol. 241. Berlin: Springer-Verlag, 1986, 1-19.

Abstract: *Software Development, as a concept, is seen as composed from aspects of Theoretical Computer Science, Programming Methodology, Software "Engineering" and Management. We define all of these concepts. We then define the notion of Software Development Graphs. Syntactically, Software Development Graphs are cycle-free, directed, finite graphs. Semantically, Software Development Graphs can be given four distinct kinds of semantics: one for each of the four major components of Software Development. The presentation alternates between serving some technical ideas and postulating some "philosophical" frame of reference for that larger concept: Software Development.*

Bjørner87a

Bjørner, D., and A. Rasmussen. *An Annotated VDM Bibliography.* Dansk Datamatik Center and Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, 1987.

The bibliography contains some 230 references to literature related to VDM. The literature spans from monographs and textbooks, via Ph.D. and Master's theses, papers in refereed scientific journals, articles in semi-refereed conference proceedings, to workshop and symposia contributions, and student term project, technical, and scientific reports.

Bjørner87b

Bjørner, D., *et al.*, eds. *VDM '87: VDM—A Formal Method at Work.* Lecture Notes in Computer Science, vol. 252. Berlin: Springer-Verlag, 1987.

The proceedings from the first VDM-Europe Symposium include papers on these subjects:

- experience gained from using VDM
- development methods (how to use VDM)
- VDM environments
- theoretical foundation of VDM
- standardization of the specification language
- tutorials on VDM

Bloomfield88

R. Bloomfield, *et al.*, eds. *VDM '88: VDM—The Way Ahead.* Lecture Notes in Computer Science, vol. 328. Berlin: Springer-Verlag, 1988.

The proceedings from the second VDM-Europe Symposium contain papers on a variety of subjects including:

- hardware test-case selection
- specification of Chinese characters
- compiler specification and development
- formal definition of standards
- using VDM to develop Ada programs
- VDM tools

The references [Andrews88], [Bear88], [Ruggles88], and [Scullard88] can be found here.

Botta87

Botta, N., and J. Storbank Pedersen. *The Draft Formal Definition of Ada: The Static Semantics Defi-*

nition. Dansk Datamatik Center, Lyngby, Denmark, January 1987.

This is a five-volume description of the static semantics (context conditions) of Ada. It is part of the results of a project sponsored by the CEC (Commission of the European Communities) Multi-Annual Programme: "The Draft Formal Definition of Ada."

Bundgaard81

Bundgaard, J., and J. Storbank Pedersen. *Kontor-Automations-Systemer (KAS), En formel model af et generisk KAS.* DDC004/1981-06-24/B, Dansk Datamatik Center, Lyngby, Denmark, 1981.

The report, which is written in Danish, contains a formal model of a generic office system expressed using Meta-IV and CSP.

Burstall77

Burstall, R. M., and J. Darlington. "A Transformation System for Developing Recursive Programs." *J. ACM 24*, 1 (Jan. 1977), 44-67.

Abstract: *A system of rules for transforming programs is described, with the programs in the form of recursion equations. An initially very simple, lucid, and hopefully correct program is transformed into a more efficient one by altering the recursion structure. Illustrative examples of program transformations are given, and a tentative implementation is described. Alternative structures for programs are shown, and a possible initial phase for an automatic or semiautomatic program manipulation system is indicated.*

Burstall81

Burstall, R. M., and J. Goguen. "An Informal Introduction to Specifications using Clear." In *The Correctness Problem in Computer Science*, R. Boyer and J. Moore, eds. London: Academic Press, 1981, 185-213.

This paper describes the algebraic specification language Clear.

CIP85

CIP Language Group. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*. Lecture Notes in Computer Science, vol. 183. Berlin: Springer-Verlag, 1985.

The report describes the specification language used in the Computer-Aided Intuition-Guided Programming (CIP) project.

Cohen86

Cohen, B., *et al*. *The Specification of Complex Systems*. Wokingham, England: Addison-Wesley, 1986.

This book presents different specification methods: algebraic, VDM, and Milner's CCS (for specifying concurrent systems). Each method is applied to a case study to illustrate its application.

Crispin87

Crispin, R. J. "Experience Using VDM in STC." In *VDM '87: VDM—A Formal Method at Work*, D. Bjørner, *et al.*, eds. Lecture Notes in Computer Science, vol. 252. Berlin: Springer-Verlag, 1987, 19-32.

Abstract: *Introducing any new technology involves organizational, skill, method and tool changes, which require a commitment from the industry concerned. The introduction of formal methods into system and software design is no exception. Before the widespread use of formal methods can be achieved, it will be necessary for the IT [information technology] industry to convince itself that the methods are genuinely usable in an industrial context, can be made to fit within the market and technical environment, and yield significant improvements over conventional methods. This paper describes some of the ways STC has used VDM to develop real systems and the benefits which we feel have been achieved. At the same time, some limitations of the existing methods have been noted, giving pointers for further development of the technology.*

The paper discusses three specific projects within STC that have applied VDM, the development of a toolset for VDM users, of control software for a monitoring station, and of an electronic mail system for telex users.

Darlington76

Darlington, J., and R. M. Burstall. "A System which Automatically Improves Programs." *Acta Informatica 6* (1976), 41-60.

Abstract: *Here we give methods of mechanically converting programs that are easy to understand into more efficient ones, converting recursion equations using high level operations into lower level flowchart programs.*

The main transformations involved are (i) recursion removal (ii) eliminating common subexpressions and combining loops (iii) replacing procedure calls by their bodies (iv) introducing assignments which overwrite list cells no longer in use (compile time garbage collection).

Futatsugi85

Futatsugi, K., *et al*. "Principles of OBJ-2." *Conf. Record of the Twelfth Ann. ACM Symp. on Principles of Prog. Lang.* New York: ACM, January 1985, 52-66.

OBJ-2 is a functional programming language with an underlying formal semantics that is based upon equational logic, and an operational semantics that is based on rewrite rules. The paper deals with issues of modularization and parameterization as well as implementation techniques.

Haff80

Haff, P., and D. Bjørner, eds. *A Formal Definition of CHILL: A Supplement to the CCITT Recommendation Z.200*. Lyngby, Denmark: Dansk Datamatik Center, 1980.

This report contains a formal definition of the programming language CHILL using Meta-IV and adopting Hoare's CSP for describing parallelism.

Hansen85

Hansen, I. Ø., and N. Bleech. *Meta-IV Tool-set, Functional Specification.* DDC165/RPT/2, Dansk Datamatik Center, Lyngby, Denmark, 1985.

A description of the DDC Meta-IV toolset including an editor, a syntax analyzer, a type checker, a database for formal specifications, and output tools.

Horowitz76

Horowitz, E., and S. Sahni. *Fundamentals of Data Structures*. Woodland Hills, Calif.: Computer Science Press, 1976.

This is not a VDM book, but it provides useful background information on data structures. This information can guide the stepwise development of data structures in VDM specifications.

Horowitz78

Horowitz, E., and S. Sahni. *Fundamentals of Computer Algorithms*. Potomac, Md.: Computer Science Press, 1978.

This is not a VDM book, but it includes algorithms that are useful in constructing VDM specifications at lower levels of abstraction.

Jones78

Jones, C. B. "Denotational Semantics of GOTO: An Exit Formulation and its Relation to Continuations." In *The Vienna Development Method: The Meta-Language*, D. Bjørner and C. B. Jones, eds. Lecture Notes in Computer Science, vol. 61. Berlin: Springer-Verlag, 1978, 278-304.

A proof of equivalence of two GOTO definitions is given. One definition uses the so-called **exit** style of definition; the other, the so-called continuation style.

Jones80

Jones, C. B. *Software Development: A Rigorous Approach*. Englewood Cliffs, N. J.: Prentice/Hall International, 1980.

This textbook on VDM focuses on the use of mathematical data types in program development. It contains a number of exercises.

Jones82

Jones, C. B. "Modelling Concepts of Programming Languages." In *Formal Specification and Software Development*, D. Bjørner and C. B. Jones, eds. Englewood Cliffs, N. J.: Prentice/Hall International, 1982, 85-123.

This paper describes the VDM approach to the modeling of central programming language concepts.

Jones86

Jones, C. B. *Systematic Software Development Using VDM*. Englewood Cliffs, N. J.: Prentice/Hall International, 1986.

This is the most recent textbook on VDM. It emphasizes proofs and proof techniques in the context of VDM and includes a number of exercises.

Jones87

Jones, K. D. "Support Environments for VDM." In *VDM '87: VDM—A Formal Method at Work*, D. Bjørner, *et al.*, eds. Lecture Notes in Computer Science, vol. 252. Berlin: Springer-Verlag, 1987, 110-117.

Abstract: *This paper discusses the experiences and issues of building two different levels of system to support the use of VDM. The MULE system is an example of an environment giving support in the syntactic generation of formal objects, such as specifications. The IPSE 2.5 system is an attempt to produce an industrial scale system to support the use of formal methods over the whole of a software development life cycle.*

Letschert87

Letschert, T. "VDM as a Specification Method for Telecommunications Software." In *VDM '87: VDM —A Formal Method at Work*, D. Bjørner, *et al.*, eds. Lecture Notes in Computer Science, vol. 252. Berlin: Springer-Verlag, 1987, 106-109.

This paper gives a short overview of VDM applications within Philips Kommunikations Industrie in West Germany.

Nielsen88

Nielsen, M., and S. Lynenskjold. *RAISE Project Overview.* RAISE/DDC/MN/19/V3, Dansk Datamatik Center, Lyngby, Denmark, 1988.

This report describes the RAISE (Rigorous Approach to Industrial Software Engineering) method and specification language, RSL (RAISE Specification Language). Also, an overview of the RAISE tools is given.

Oest86

Oest, O. N. "VDM from Research to Practice." *Information Processing 86: Proc. IFIP 10th World Comp. Congress.* Amsterdam: North-Holland, September 1986, 527-533.

Abstract: *The Vienna Development Method (VDM) is one of the few mathematically based methods for software development which has been successfully transferred from the protected world of the research laboratories into industrial use.*

After a brief description of VDM—its contents and the ideas behind it—the paper continues by outlining the history of VDM from its foundation in the early seventies through its evolvement through research and application into its current form. In this period VDM has been changed, extended, and variants have emerged. VDM in all its variants is now forming the major basis for what is known as the ESPRIT RAISE project supported by the CEC (Commission of the European Communities).

The paper ends by describing the largest application of VDM to date, the design and development of the DDC Ada Compiler, an effort which took place at Dansk Datamatik Center, Denmark, from 1981 to 1984. This development was partly funded by the CEC under its Multi Annual Programme within the field of Data Processing.

Pedersen87

Pedersen, J. Storbank. "VDM in Three Generations of Ada Formal Descriptions." In *VDM '87: VDM—A Formal Method at Work*, D. Bjørner, *et al.*, eds. Lecture Notes in Computer Science, vol. 252. Berlin: Springer-Verlag, 1987, 33-48.

Abstract: *Since 1980, three different formal descriptions of the Ada programming language have been developed, based on the principles of the Vienna Development Method (VDM). This paper characterizes each of the three descriptions and explains some of the differences.*

Pedersen88

Pedersen, J. Storbank, and M. H. Klein. *Using the Vienna Development Method (VDM) to Formalize a Communication Protocol.* SEI-88-TR-26, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., December 1988.

This report contains a formal description in Meta-IV of a communication protocol used by the U.S. Navy.

Plotkin81

Plotkin, G. D. *A Structural Approach to Operational Semantics.* DAIMIFN-19, Aarhus University, Denmark, Aarhus, Denmark, September 1981.

This book describes Plotkin's Structural Operational Semantics, often referred to as "SOS."

Prehn87

Prehn, S. "From VDM to RAISE." In *VDM '87: VDM—A Formal Method at Work*, D. Bjørner, *et al.*, eds. Lecture Notes in Computer Science, vol. 252. Berlin: Springer-Verlag, 1987, 141-150.

Abstract: *Although VDM—the Vienna Development Method—has probably been the most widespread and popular so-called formal method for software development in use so far, it is clear that VDM suffers from a number of deficiencies. In this paper, the transition from VDM to a new "second generation" formal method—RAISE—is discussed. Problems with VDM are discussed, and their solutions within RAISE are outlined. The reader is assumed to be familiar with VDM.*

Ruggles88

Ruggles, C. "Towards a Formal Definition of GKS and Other Graphics Standards." In *VDM '88: VDM—The Way Ahead*, R. Bloomfield, *et al.*, eds. Lecture Notes in Computer Science, vol. 328. Berlin: Springer-Verlag, 1988, 64-73.

The paper reports on work done at the University of Leicester on formalizing the ISO Graphical Kernel System (GKS) standard using Meta-IV.

Scullard88

Scullard, G. T. "Test Case Selection Using VDM." In *VDM '88: VDM—The Way Ahead*, R. Bloomfield, *et al.*, eds. Lecture Notes in Computer Science, vol. 328. Berlin: Springer-Verlag, 1988, 178-186.

Abstract: *This paper describes the design verification process adopted by the VLSI Distributed Array Processor (VDAP) Project. In this project structured, informal design techniques were used in*

the hardware design process, but the validation team used some of the tools and methods of VDM as a means of defining a testing strategy.

Stanat77
Stanat, D. F., and D. F. McAllister. *Discrete Mathematics in Computer Science*. Englewood Cliffs, N. J.: Prentice-Hall, 1977.

This book contains the discrete mathematics background material needed for this module.

Stoy77
Stoy, J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, Mass.: MIT Press, 1977.

This is a classical textbook on denotational semantics.