VIENNA DEVELOPMENT METHOD

A fundamental step in the planning and design of nontrivial size software is the development and analysis of first a conceptual model and then a design model of the target system. The Vienna Development Method (VDM) is both a methodology for developing and refining models of software systems and a language for specifying such models. The distinctive feature of VDM models and development is the strong foundation on mathematics and logic and rigorous description of certain properties of the model. With such a mathematical foundation, VDM is particularly suited for use in development of safety-critical software systems.

Software engineers recognize why models of software are important. Fundamental errors of concept or design can be detected and easily corrected within a model. If such errors propagate into developed code, they can be severely costly and even catastrophic. In the early development of the software engineering field, there was a distinction made between models for requirements analysis, specification of target software, design, and even verification and/or testing. Now an understanding is emerging that a unified model of software can be used for each of these purposes with only refinement of the amount of abstraction and amount of detail within the model. VDM provides for this kind of unified model. VDM models are used to define requirements, specify processes and software architecture, and serve as the framework for refinement of target code.

The original VDM concept was developed at the IBM Vienna Research Laboratories during the 1970s (1). In the 1980s VDM took on an independent existence with strong support in England. During the same time period, the related concept of Z (pronounced zed) (2,3) was also developed in England. Z is similar in structure and role to VDM, but by name VDM seems to have a greater emphasis on the methodology component. In the 1980s, a procedural view of software was predominant; hence, both VDM and Z were developed without the now common class/object structure. Both are continuing in a process of refinement and standardization. In the 1990s, methods of object-oriented analysis and design (OOAD) evolved from the structured development methods of the 1980s (including VDM). OOAD is not a single methodology and notation, but it encompasses several similar methodologies and notations. Both VDM and Z can be used within an object-oriented framework. A comparison of VDM and OOAD vocabulary is shown in Table 1. An object-oriented form of VDM will be illustrated in this article. There are other object-oriented extensions of both VDM and Z (4,5). In the 1990s, a number of tools for development and processing of VDM models have been refined and are beginning to be used (6,7).

VDM and Z both have a core foundation of discrete mathematics (8). The mathematical structures of sets, sequences, and maps are used to model data, and mathematical logic is used to model operations and properties of data. These mathematics and logic concepts are not well rooted in computer science curricula in the United States; hence, neither VDM nor Z is widely promoted in the United States. Yet, the po-

Table 1. Relationship of VDM and OOAD Vocabulary

VDM	OOAD
context model	use-case model
process model	object model
data flow model	object model
abstract machine	class/object
data stores	data members of classes
data models	aggregation
processes	methods of classes
transactions	messages
invariant assertion	none (comments about data constraints)
pre-/postassertions	none (implicit in use case scenarios)

tential benefit of use of models based on mathematics is great. First, the models are suited for analysis and even verification of certain properties, and second, the models are suited for systematic refinement to executable form. That is, VDM models are suited for analysis of requirements and for specification of software functions, and they are also suited for refinement as a design model. Because the mathematical modeling aspect of VDM is usually difficult for persons learning the methodology, most books begin with a tutorial presentation of the discrete mathematics and logic concepts, even though that is not the beginning point of the whole methodology.

Because VDM is so based on mathematical expressions, VDM models are often incorrectly viewed as only formal specification models rather than as software development models. On the World Wide Web, one reference link to a VDM home page is in the area Computers, under sub-area Formal Methods. Within Usenet, VDM and Z are discussed within comp.specification. In the United States, neither formal methods nor formal specifications are widely accepted as necessary components for general software development. Instead, formal methods are viewed as appropriate for safety-critical software, for which there is some property, called a safety property, that must be established with great certainty. The process and arguments to convince reviewers that a safety property holds is called verification. That is distinguished from validation, which is concerned that software meets its general functional requirements. Verification can be done either informally, by careful reasoning about the interaction of the safety property and the mathematical models of operations, or formally by using tools to prove the safety property based on the mathematical specifications of the model operations. There are a number of tools that assist in verification of VDM models. It is widely noted (perhaps as a criticism) that such formal verification can only be done effectively by persons well experienced in mathematical logic. Even then, formal verification is tractable only for fairly simplified forms of safety properties.

Within the area of formal specifications, VDM is characterized as a model-based specification form, which is distinguished from algebraic specification forms and object models. The key aspects of the model-based part of VDM is that data components are modeled (represented) by using abstract mathematical types and the behavior of operations is specified by using assertions which constrain the state of data components before and after each operation. In contrast, in algebraic specifications, data components are not explicitly represented. Rather, the behavior of model operations are defined implicitly by equations which constrain their interactions. In object models, data components are represented by aggregation of component objects and the behavior of operations is described by case scenarios that show the propagation of messages.

Since object-oriented development is such a common foundation for software, we will illustrate VDM within an object-oriented framework. This form is modified in two ways. First, it uses an object model (classes and objects) to represent the architectural structure of a system, rather than using separate procedures as in the original VDM. Other aspects such as the models of data components and specification of operations and properties for verification follow the concepts of VDM. Second, it uses a textual representation of mathematical symbols of VDM.

Some further comparison of VDM and object models should be noted. Object models do not define a specific implementation structure for aggregates of data or specific semantics of messages. It is an implicit expectation for such models that developers can map structures of aggregation to some concrete implementation structure and that case scenarios sufficiently describe the behaviors of messages that developers can determine the implied implementation. Similarly, the abstract representation of data components in VDM do not determine a specific implementation. However, there are common mappings of the abstract types to concrete implementation. Once a specific mapping is determined, the operation specifications strongly constrain the refinement of assertions into executable code.

COMPONENTS OF VDM

The framework of VDM is the development methodology. Development begins with a high-level context model which identifies external actors, processes, and major data stores. This step is typically part of the analysis of requirements. In the object-oriented form, the top-level model is the use-case model. It is a top-level object model with identification of external operations. Visual tools can be used to construct the model diagram.

Within the development framework, the next component is data modeling, which consists of modeling of parameters and data stores using the VDM data types and defining invariant properties which constrain the data beyond the basic type definitions. Modeling data using the VDM types is similar in concept to representing data structures using programming languages, but data types are significantly different. The abstract VDM types are suited for very concise expressions, but they do not prescribe specific implementations. The VDM types are summarized in the Appendix. The concept of using predicates to define invariants about properties of data items also extends common programming style. The closest aspect in most programming languages is use of comments to explain properties of the data items. Some languages do support assertions (for example, Eiffel), but those assertions constrain the concrete data types of the language and are thus much less expressive than assertions about the abstract data types of VDM.

One common style for the VDM types and assertions is given within the specification language VDM-SL (9). VDM-

SL is a version of the specification language for the VDM methodology. There are also tools for checking the syntax of VDM-SL, a library of LaTex style macros for preparing VDM-SL for output in Postscript form, and translators for mapping VDM-SL to specific programming languages. VDM and VDM-SL use a number of mathematical symbols that are not directly available on most keyboards. The VDM-SL form in not used in the case example. Instead, a pure text form for VDM type and assertion symbols is used. Some VDM symbolic forms are shown in the Appendix.

The next component of the framework is process specifications. These are developed by specifying what is called the signature for each operation (parameters and access to external variables together with types and modes of parameters) and by defining assertions to express pre- and postconditions. The method of writing assertions is sometimes difficult to grasp, because assertions are essentially nonprocedural, whereas most programming languages are procedural. That is, assertions define the states of the data prerequisite to and resulting from each operation; they do not define the specific steps to compute the resultant state of the data. There is nothing directly equivalent to pre/postassertions in most programming languages. Again, Eiffel (and some other languages) supports a limited form of assertions, but those apply only to evaluation of the state of variables after a computation, whereas VDM process assertions define the computation to be done by the process. The specification language VDM-SL includes specification of processes, but in a nonobject-oriented style. In the nonobject-oriented style, each process declares which external data items are accessed. In the object-oriented form used in the case study shown later, each class method will access only the data components of the class, so it is not necessary to declare which external data items are accessed.

Another component is the mechanism for verification of invariant assertions. It must always be the case that claimed invariant assertions are consistent with the pre-/postassertions for each operation. Verification requires some demonstration, ranging from informal arguments to formal, mechanically checked proofs, that invariant assertions can be proved based on the process specifications. The verification component consists of the method of presenting such proofs and any tools used to develop proofs. One noted tool for verification of VDM-SL is the Mural system (10).

The final component of the development framework is the mechanism for refinement of the abstract model. The refinement includes, first, mapping abstract data types to concrete implementation structures and second, mapping process assertions to consistent procedural code. For rapid development of executable prototypes, some libraries of simple mappings of the abstract types and operations to concrete structures and operations have been defined. This type of development by refinement of specifications is one form of prototyping (11).

Even though various software tools are available to support each component of the VDM framework, none of the components can be effectively automated. They compose a methodology that can be used by software developers who have gained insight about the components.

A CASE STUDY

This section illustrates the steps of the VDM methodology by using an example of a small library system. The library example has been used in several papers about formal specifications. As explained earlier, the example is a modified form of VDM in that the architecture is defined by an object model, mathematical symbols are represented in textual form, and some shortcuts are used in writing predicates. Andrews and Ince (12) present a similar library example, using a pure VDM form. The case example is not complete, but it does illustrate each of the steps. The steps are presented in sequence, but in reality, software development more likely follows what is called a spiral or iterative process. That is, as issues are identified in each step of development, changes may be made to results of earlier steps.

Requirements

The process of elicitation of requirements is essentially the same for all software development methodologies. The primary requirement is the functions of the target system. Other requirements include major data storage, system and network architecture, performance, and security. For VDM, statements of requirements should be closely coupled with the context model shown next. The nature of requirements and the structure of the context model will depend greatly on the architectural pattern of the system, such as primarily a data base, a real-time controller, or a filter. The functional requirements should identify all agents that interact with the software system, all transactions that can occur, information about data items for all transactions, and all major data stores that are known by users. The statements of requirements for the simple library system are given in Table 2.

Context Model

The context model is the external view of actors, their interaction with the system, and major system architecture, including network structure and major data stores. The model consists of two parts, a graphical representation, as shown in Fig. 1, and representation of transactions, shown in the form of a grammar in Table 3. The context diagram shows multiple stations for checking books and information inquiry. It shows that the library is a single processor system that maintains data files for user information and books information. The specific kinds of book, information, and administrative transactions are defined in the transactions grammar. The identification of each transaction includes data items and their mode (input or output). Finally, the description of the data stores identifies data components and types.

In conventional design methods, the context model is a data flow model. In OOAD, the context model is the use-case model. In the original VDM, the transaction grammar merely identifies the transactions cases. In the example transaction grammar, we show not only the cases for different uses, but also sequential ordering and concurrent processing of various transactions. This grammar presents the same kind of temporal ordering information as would be shown in state diagrams that are used in other object-oriented design methodologies. The transactions, such as UserId, identify parameters, such as BkID, types of the parameters, such as “tBkID,” and the mode of each parameter, as “out.” The default mode is “in.” The types identified in the transactions are defined later in the conceptual model. For a real software development project, there would also be accompanying narrative to explain each transaction, but that is omitted for this example.

Table 2. Statement of Requirements for Library Example

1. General
   The library software system is intended to serve a single library. The system should keep record of books in the library inventory and of users who are registered to borrow books.
   The system should allow for one hundred thousand users and one million books.
2. Configuration
   The system should support multiple transaction stations (for check-in and check-out of books), multiple information stations (for queries about book information), and administration terminals (for administrative transactions).
   a. The transaction station will support a bar code reader, four keys (Clear, Enter User code, Enter code for book check out, and Enter code for book return), and a display status (null, OK, ReEnter, or an Error code number).
   b. The administration terminal should support text IO for inputting information about users and books. It should allow operations to add new books, remove books, add new users, and remove users.
   c. The information terminal should allow text report for searches by author, prefix of title, and it should report call number.
3. User information
   User information should include name, address, telephone number, unique user ID, and list and due dates of books checked out, and date of last check out.
4. Book information
   General information about books is available in a purchased database of books in print. It contains author and title and is indexed by both call number and ISBN.
   The library inventory, book information should include a unique book ID, a copy number, who has the book checked out, and date of last check out.
5. Transactions
   a. Transaction stations record borrowing a book and the returning of the book.
   b. Administration stations record addition of new books and removal of books from the library inventory and addition of new users and removal of users. Administration can query for books or users with no transactions past any specified date. Administration stations can record requests to recall a book.
   c. Information stations should support queries about books by author, by title, and by call number.
   d. Additional transactions may be added to facilitate the functioning of the library and the information system.
6. Constraints
   a. User cannot borrow any books if user has books overdue.
   b. If a user is removed, any checked-out books are also removed.
   c. Recalled books are held for the requesting user.

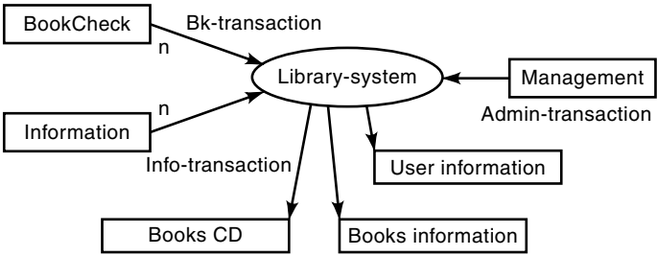

Figure 1. Context model for library example.

Conceptual Model

The context model is refined to define details of the primary VDM conceptual model. This consists of an architecture diagram and several text statements of component details. Typically, details of the structure would be expanded in a series of refinements. In the original VDM, the overall architecture is represented as an expanded process/data-flow model which shows the designer's understanding of internal data stores and structure. In the extended form, this visual component is shown as an object model in Fig. 2. The object model is similar to that in OOAD, but it differs in the modeling of data components. The data components are represented in terms of VDM abstract types, rather than as additional classes. This data modeling rather than expansion of classes allows the object model to be more concise than in OOAD. For the library, the design shows there are separate objects that encapsulate book information and user information.

Next, modeling of the data components of the two classes is shown in Table 4. As is typical for VDM, all data components of classes are modeled as sets, sequences, and maps. Once the class structure is defined, there is no further issue of efficiency of the data component models. The issue is to define what information is stored and what updates of the information are to be defined. Later, the abstract models can be implemented in different ways to meet particular needs for space and time efficiency. Thus, keeping information about books checked out in both the Books and Users objects was a design decision, likely based on encapsulation and access efficiency. (And, such redundancy of information allows discussion of the invariant property later.) In contrast, consider how to represent the maximum copy number for a particular ISBN:

$$\begin{aligned}\text{maxCopy}(\text{N}:\text{tISBN}) = \max(\,\text{set}(\text{i} \mid (\text{exist B} \bullet \text{Stock(B).ISBN} \\ = \text{N and Stock(B).Copy} = \text{i}))\end{aligned}$$

This defines information which needs to be used in adding a new copy of book, but it is clearly not the appropriate structure or method to be implemented. More strictly, the set in the previous expression should be written as:

$$\text{set}(\text{i}: \text{Natural} \mid (\text{exist B in Stock.domain} \bullet \ldots))$$

but we omit the set constraints for informal discussion.

A further step of data modeling is the development of invariant properties about data models and interrelationship of data models. Typically, such invariant assertions define constraints beyond those of the VDM data types. A typical constrain is that there are restrictions on fields of records within a collection of such records. An example safety property for the library is shown in Table 5. Both Stock and UsersData are restricted to not have 0 in their domain, because in the Stock structure, the 0 user indicates "no user." The fields Due and User of structure Stock are redundant, but it is desirable to use an explicit Boolean field. The predicate SameDue defines the condition that user U has book B due. The predicate AllSameDue defines the condition that for every book that is checked out, there is exactly one user that has the book. Finally, the invariant makes the claim that AllSameDue holds for the data models Stock and UsersData. This kind of constraint is often clear to designers, but easily lost in details of code implementation.

Table 3. Transaction Specifications

```
Library-transaction = BkTransaction* ; InfoRequest* ; AdminTransaction*

InfoRequest = pending

AdminRequest = pending

BkTransaction = CheckOutSeq | Return(BkId: tBkId; Status: out tStatus) | Clear( )

CheckOutSeq = CheckId( UserId: tUId; Status: out tStatus ), Book*, End( )

Book = CheckOut( BkID: tBkID; Status: out tStatus )

InfoTransaction =
  FindbyAuthor( Author: String; Bks: tBooks )
  | FindbyTitle( Title: String; Bks: tBooks )
  | Find( Isbn : tISBN ; Bk: tBookInfo )

AdminTransaction=
  Find( UID: tUID;  Uinfo: out tUserInfo )
  | Add( Uinfo: tUserInfo; UID: out tUID )
  | Delete( UID : tUID )
  | Update(  UID: tUID; Uinfo: inout tUserInfo )
  | InfoTransaction
  | Add( Book : tISBN ; BkID: out tBkID )
  | Delete( Book: tBkID )
  | Find( BkID: tBkID; Bk: tBookRec)

Legend:
  ;          concurrent transactions
  |          alternative transactions
  ,          sequential transactions
  *          0 or more instances of transaction
  [ ]        optional transaction
  ( )        encloses parameters for transaction
  pending    not defined yet
  out        return parameter
  inout      parameter in and return
```

VDM specification of operations requires identification of parameters, types, modes, and specifications for each operation. Thus, the conceptual model contains all the information of a data flow model. Example operations specifications are shown in Table 6. The preassertions, labeled "in," define the state of parameters and object data components before invocation of the operation. The postassertions, labeled "req" and "out," define the state of return parameters and object data components after execution of the operation. The "req" assertion defines a check of an input state and defines an exception or return parameter in case the check fails. Within the postassertions, final values of variables are denoted as "Var'out." Thus, the CheckId operation returns an appropriate Status'out value if any of three checked conditions fail. Otherwise, the Status'out is OK. The CheckOut operation knows that the CheckId is valid. It returns BookError value if the book is not indicated to be in stock. It invokes the Books.Take operation and updates the UsersData information. The "=+" notation is a short cut expression for the longer form:

$$\text{Var'out} = \text{Var} + \text{NewItem}$$

where the "+" operation may be set union, map overwrite, etc., depending on the structure of Var.

Often, the style of writing output assertions is difficult for programmers who are experienced using procedural languages. The output assertions intentionally suppress details of the sequential processing. However, the general concept (but not the legal form) of VDM can be used even if the pre- and postexpressions are written in a style of procedural coding.

Verification

The goal of verification is to show that claimed invariant properties are indeed true. Of course, the invariants about specific data stores may not hold during process of updating those data stores. Thus, the invariant for the library will not hold during execution of operations of Books and Users, but it should hold at any point of execution external to those two objects. The general structure of verification is to show, first, that the invariant holds for initial states of all objects, and then, show by induction that the invariant holds for each operation of related objects. Each step of induction assumes first that the invariant holds before execution of the operation and then uses the assertions of the operation to show that the invariant must hold after the operation. This must be done for every operation that can change the state of the related data objects. An informal outline of one step of verification of the library invariant is shown in Table 7.

A formal treatment of verification can involve a large number of component proofs, each with numerous steps with many details, which is a formidable task. Fortunately, several factors work to constrain the size of this work. First, invariants will be associated with specific objects rather than the whole system. Second, verification using abstract data models

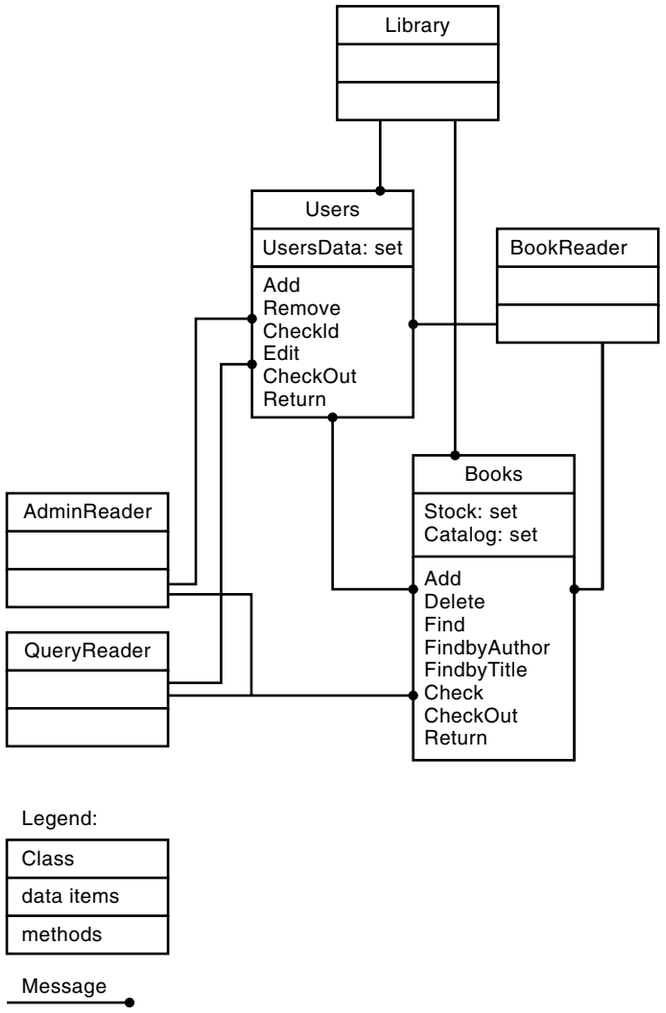

Figure 2. Object model component of the conceptual model.

Table 4. Type and Data Models

```
Library::

Types:
  tStatus = set(OK, BadID , PastDueBooks, MaxBooksOut, BookError );
  tUID = Natural;
  tBookId = Natural;

Books::

Types:
  tBooks = seq( tBookInfo);
  tISBN = Positive;
  tBookInfo = record(   Authors :   String;
    Title:      String;
    ISBN:       tISBN ;
    Call :      String
    Out:        Boolean );
  tAuthors = seq(  String) ;
  tBookRec = record(   ISBN:    tISBN;
    Copy:       Positive;
    Out:        Boolean;
    User:       tUID;
    Due:        tDate );

Data:
  Catalog: set(tBookInfo);
  Stock: map( tBkID, tBookRec);

Users::

Types:
  tUserInfo = record(   Name:     String;
    Address1: String;
    State:      String;
    ZIP:        Positive;
    Phone:      Positive;
    LastUsed: tDate;
    Due:        tDue );
  tDue = set( tDueItem);
  tDueItem = Record (   Book: tBkID;
    Date: tDate );

Data:
  Udata: map( tUID, tUserInfo) ;
```

requires much less detail than verification for concrete structures. Third, there has been continuing development of verification tools that assist in performing the mechanical details of verification. Still, such formal verification is suited for very special safety properties. Informal peer review of invariants is effective for most commercial development.

Reification

Reification is the refining of the conceptual model to concrete structures and code. The process consists first of selecting a particular implementation for each abstract type used to model the data stores. This decision is always a question of space and time performance and persistence of the concrete structures. For prototype development, there are libraries of common mapping of abstract structures to common implementations. This mapping to concrete structures will introduce additional classes and methods into the object model, precisely those classes and methods that were ignored in the conceptual model. Then, the "req" and "out" specifications for each operation are translated to procedural code using the appropriate operations for the implementation of each data component. This translation of assertions is usually an easy

Table 5. Safety Property

```
/* consistency of Books::Stock.Due and Users.UsersInfo.Due information:

Books::

  /* no 0 in domain
Inv: Stock.domain sub Positive;

  /* consistency of Due and User fields
Inv: (all B in Stock.domain • Stock(B).Due eqv Stock(B).User > 0 ) ;

Users::

  /* no 0 in domain
Inv: UsersData.domain sub Positive;

Library::

Define:
  SameDue( B: tBkId; U: tUID ) =
    Books.Stock(B).User = U
      eqv
    tDueItem(B,_) in Users.UsersData(U).Due ;

  AllSameDue =
    (all B in Books.Stock.domain •
    (all U in Users.UsersData.domain • SameDue(B, U) ) );

Inv:  AllSameDue ;
```

Table 6. Example Method Specifications

```
Users::

CheckId( U: tUID ; Status: out tStatus);
  req:  (U in UsersData.domain) else Status'out = NotValid;
  req:  (all i in UsersData(U).Due.domain •
    UsersData(U).Due(i).Date >Today) else Status'out = PastDue;
  req:  UsersData(U).Due.Size < MaxBooks else Status'out = MaxBooksOut;
  out:  Status'out = OK ;

CheckOut( U: tUID; BkID: tBkID; Status: tStatus );
  in:   CheckId( Uid, Valid );
  req:  Books.Check(BkID, In) else Status'out = BookError ;
  out:  Books.Take(U, BkId)
    and
    UsersData(U).Due'out =+ tDueItem( BkID, Today + OneMonth)
    and
    UsersData(U).LastUsed = Today
    and
    Status'out = OK ;

Remove( U: tUid );
  in:   U in Users'domain ;
  out:  (all x in Users(U).Due • Books.Remove( x.Book )  );
  out:  UsersData'out = { U } <\- UsersData  ;
```

Table 7. Outline of Two Verification Steps

```
For initial state:

Infer AllSameDue
 ------------------------------------------
1 ?     AllSameDue
2 =     (all B in Books.Stock.domain •
        (all U in Users.UserdData.domain • SameDue(B, U) ))
3 =     (all B in set() • ...)
4 =     true

Reasons:
1. statement of goal.
2. substitute definition of predicate.
3. substitute initial values for Stock and UserData (both empty
sets).
4. from property of the "all" predicate (true for empty sets).

For operation  Users.CheckOut( U, B):

From
A1    AllSameDue
A2    CheckId( Uid, Valid ),
A3    Books.Check(BkID, In),
A4    Books.Take(U, BkId),
A5    UsersData(UID).Due'out =+ DueItem( B, Today + OneMonth)
Infer AllSameDue'out
 ------------------------------------------
1       Stock'out = Stock (+) B
2       UserData'out = UserData (+) U
3 ?     AllSameDue'
4 =     AllSameDue and SameDue(B, U)
5 =     true

Reasons:
1. indicates that only book "B" is updated.
2. indicates that only user "U" is updated.
3. statement of the goal.
4. split the range of the quantifiers !! a common step which omits
   details.
5  the first term of 4 is true by assumption,
   second term is true from A4 and A5.
5b true from A4 and A5.
```

Table 8. Refinement of a Stock Data Component

```
Books::

Type:

  tBookRec2 = record(  ISBN:     tISBN;
    Copy:       integer;
    NextCopy: tBookId; /* index of next copy with same ISBN
    User:       tUID;     /* User > 0 imp Out
    Due:        tDate );
  tCopyRec = record(   Number:   integer;
    First:      tBookId);

Data:

  Stock : array( 1 .. MaxStock) of tBookRec;
  Copy:  hashTable( ISBN ) of tCopyRec;
```

coding task; it does not require new steps of design. The important point is that if the data models are refined correctly and the assertions are translated correctly, then the invariant properties are guaranteed to hold for concrete model. No further verification is required to determine that the invariant property holds for the implementation code level.

To illustrate these steps, a beginning refinement of the Stock structure is given in Table 8. The abstract map is reduced to just an array (which may be a very bad structure if book numbers are sparse). The conceptual field Out has been removed. Since it would be infeasible to search for copy numbers for duplicate books, we use a hash table to relate ISBN to a record structure of the highest copy number and the BookId for the first copy. This structure in turn would be refined to allow more efficient storage and retrieval. Given the particular implementation of the data components, refinement of the operation specifications should be straightforward.

FURTHER READING

To begin to study VDM, books on discrete mathematics (1) or tutorial books about VDM (9–12) are good beginnings. Several case studies using VDM have been collected in (13). To keep up with latest work on VDM, the Web links are useful (7,14–16). These contain links to general information, on-line technical articles, tools, case studies, and related topics such as Z, specifications, and verification, and extensions such as Z++.

APPENDIX. VDM TYPES AND EXPRESSIONS

Scalar Types

Name	Meaning
Boolean	{false,true}
Natural	{ 0, 1, 2, ...}
Positive	Natural − {0}
Integer	Natural + {−x | x: Positive }
Rational	{ n/d | n: Natural and d: Positive }
Real Character	{ a, b, c, ...}
String	seq(Character)
Atom	any identifier

Notes. The scalar types are taken as primitives. With the exception of type Real, the scalar types are defined in terms of sets and sequences.

Container Types

Name	Constructor	Meaning	Notes
Set	set(e1,e2,e3)	{e1,e2,e3}	(1)
PowerSet	set(T)	{x: T}	(2)
Sequence	seq(T)	map([0 .. n], T)	(3)
Map	map(D,R)	{ ⟨x: D, y : R⟩}	(4)
Map1	map1(D,R)	one-to-one map	(5)
Tuple	T1 × T2	(x : T1,y : T2)	(6)
Record	record(f1 : T1; f2 : T2;)		(7)

Notes. Tuples and records have a fixed size. The set, sequence, and map container types are dynamic. They do not have a fixed size. All the VDM dynamic containers are homogeneous (they have one defined type for all items in the container). Most interesting models consist of containers of other structures such as records, tuples, or other containers.

1. An enumerated set is a collection of the enumerated items. Sets cannot contain duplicate items. A variable of this type has a single value which is an item of the defined set. This type serves the same role as enumeration types in other languages. For example:

   type: tName = set(a, b, c);

   N: tName;

   out N'out = a ;
2. The value of a variable of a power set type is a set of values. That set of values is a member of the power set. For example:

   type: tNames = set(tName);

   Names: tNames;

   out Names'out = set(a, c) ;
3. Sequence is a list of items with index position from 0 to Size-1. For example:

   type: tWaiting = seq(tName);

   W1, W2: tWaiting;

   req W2.size < 10 ;

   req W2(3) = c ;

   out W1'out = seq(a, c, a, b);
4. Map is merely a finite table relating unique domain values to corresponding range values. For example:

   type: tAge = map(tName, Natural);

   Last, Class: tAge;

   req c in Last'domain and 10 in Last'range;

   out Last'out = map(a, 10) + map(b, 11);

   out Class'out = Last + map(a, 11) ;
5. Map1 is a one-to-one map, which requires that both the domain and range values must all be distinct. For example, map((a, 10), (b, 5), (c, 12)) is one-to-one.
6. Tuples are like records but with number fields rather than named fields.
7. Records are available in many programming languages. Now, they are entirely redundant with the class structure. All record structures could be replace with class declarations.

Common Set Operations

Name	Textual	Symbolic
Enumeration	set(a,b,c)	{a,b,c}
Member	x in S1	x ∈ S1
Union	S1 + S2	S1 ∪ S2
Intersection	S1 int S2	S1 ∩ S2
Difference	S1 − S2	S1 − S2
Constructor	set(x: T | p(x))	{x : T | p(x) }
Size	S1.size	|S|

Common Sequence Operations

Append	S1 + x
Index	S(i)
Concatenate	S1 + S2
Domain	S1.domain
Range	S2.range
Size	S1.size

Common Map Operations

Declaration	map(D,R)	D → R
	map1(D,R)	D ↔ R
Enumeration	map(d,r)	{d ↦ r}
	map((d1, r1), (d2, r2))	{d1 ↦ r1, d2 ↦ r2}
Evaluation	M(x)	M(x)
Domain	M.domain	dom M
Range	M.range	rng M
Overwrite	M1 + M2	
Restrict range	M /> S1	M ▷ S1
Restrict domain	S1 <\ M	S1 ◁ M
Subtract range	M1-/> S1	M ⩥ S1
Subtract domain	S1 <\- M	S1 ⩤ M

Common Predicate Expressions

Boolean Operators

and, or, imp, =, not

p else q = if not p then q

Quantifier	Textual	Symbolic
Universal	(all x in X · P(x))	(∀ x ∈ X · P(x))
Existential	(exist x in X · P(x))	(∃ x ∈ X · P(x))
Unique	(exist! x in X · P(x))	(∃! x ∈ X · P(x))

BIBLIOGRAPHY

1. D. Bjorner and C. B. Jones, The Vienna development method: The MetaLanguage, in *Lecture Notes Comp. Sci.,* **61**: New York: Springer-Verlag, 1974.
2. M. Spivey, *The Z Notation: A Reference Manual,* Englewood Cliffs, NJ: Prentice-Hall, 1989.
3. I. J. Hayes, C. B. Jones, and J. Nicholls, Understanding the differences between VDM and Z [Online], 1993, Available ftp: ftp.cs.man.ac.uk in file pub/TR/UMCS-93-8-1.ps.Z

4. E. H. Durr and J. van Kawtijk, VDM++: A formal specification language for object-oriented designs, in *Computer Systems and Software Engineering,* Proc. CompEuro'92, IEEE Comput. Soc., 1992, pp. 214–219.
5. D. Carrington et al., Object-Z: An object-oriented extension to Z, in S. Vuong (ed.), *Formal Description Techniques II,* Amsterdam, The Netherlands: Elsevier, 1990, pp. 281–296.
6. VDM tools Available: ftp://chowell.ncl.ac.uk/pub/fu-tools-db.
7. VDM-ST Tools, Available: [online] http://www.ifad.dk/vdm/vdm.html
8. D. Ince, *Introduction to Discrete Mathematics and Formal System Specifications,* London: Oxford Univ. Press, 1988.
9. D. Bjorner and C. B. Jones, *Formal Specifications and Software Development,* Englewood Cliffs, NJ: Prentice-Hall, 1982.
10. B. Cohen, W. Harwood, and M. Jackson, *The Specification of Complex Systems,* Reading, MA: Addison-Wesley, 1986.
11. D. Andrews and D. Ince, *Practical Formal Methods with VDM,* New York: McGraw-Hill, 1991.
12. F. D. Rolland, *Programming with VDM,* Macmillan, 1992, pp. 122.
13. C. B. Jones and R. Shaw (eds.), *Case Studies in Systematic Software Development,* Englewood Cliffs, NJ: Prentice-Hall, 1990.
14. VDM Forum; Available: send mail to "mailbase@mailbase.ac.uk," with the message body "join vdm-forum First Last-Name"
15. Formal Methods, Available: http://www.comlab.ox.ac.uk/archive/formal-methods/pubs.html#intro (contains links to several VDM and Z references and bibliographies)
16. Formal Methods Europe, Periodic conference on formal methods including VDM and VDM++ and tools. Available: http://www.csr.ncl.ac.uk:80/projects/FME/

WILLIAM HANKLEY
Kansas State University