SmallVDM: An Environment for Formal Specification and Prototyping in Smalltalk

Silvio Lemos Meira
Cássio Souza dos Santos

Universidade Federal de Pernambuco
Departamento de Informática
PO Box 7851
50.739 Recife-PE BR

Abstract

We present an environment for specification and prototyping of object-oriented systems in Smalltalk along with a style of specification, which allows the definition of some object-oriented characteristics in VDM. A set of auxiliary tools stimulates the development of specifications on-line and fast prototyping. The presentation is mostly informal and the mechanism is exemplified by giving an object-based specification of a simple process scheduler. We discuss the gains of using formal methods, coupled with a good programming environment, in the implementation of object-oriented software systems.

1 Introduction

The problems raised by the development and maintenance of large and complex software systems in a competitive and dynamic environment have been studied under various perspectives. The term *"software crisis"* stands for this problem and many efforts have been made to overcome it, what has given rise to different paradigms and methodologies of software development. Among many others, formal specification and object-orientation address that problem.

Formal specifications allow the construction of more reliable and maintainable systems, as they provide the basis for logical reasoning about design in all stages of the software life cycle. The correctness of the proposed solution may be attested by means of rigorous or formal proofs. Verification and validation of the design are both assisted by formal methods.

The use of formal methods in practice, despite all claims about their advantages, is far behind potential due to the the lack of interesting tools to support formal development. In addition, most programmers have had only practical education so far and are not qualified to use formal methods.

Nevertheless, the scenery is changing, as foreseen in [Hoa84]. Disciplines on formal specification have started to be normal offerings in computing courses, as the interest in the subject increases. The need to use formal mathematical methods in industrial applications has driven the development of more and more interesting tools.

Certainly not a panacea, as many want to (make) believe, object orientation emerges as an extremely interesting approach to software development. The emphasis on reusability, along with the unification of many phases of development, has to do with many problems related to the *software crisis*.

The marriage of formal specification with object orientation is not a novelty, but it is known to be unstable. Object-Z [Duk90], MooZ [Mei90b] and Abel [Dah87] are some attempts to formalize object-oriented software design. The approaches invariably define a new notation, although some happen to be extensions of existing ones, in order to capture basic concepts such as class and inheritance.

We take a different approach. Starting from VDM, a notation which has achieved a good level of maturity and acceptance, we propose a style of specification supported by a development environment, supporting a development method. The method determines some criteria for the design of object-oriented systems with a fair degree of formalism. The environment, named SmallVDM, has been developed in Smalltalk and supports both the specification and prototyping phases.

SmallVDM is an extension of a previous system [Tep90], where prototypes of model-based specifications were derived in a Smalltalk environment. This work presents a more elaborate and suitable environment, which extends the original Smalltalk environment and makes the specification and prototyping of object-oriented systems an easier task to accomplish. What's more, the original work dealt with model-based specifications, whilst the current one imposes a style which allows the definition of object-based specifications. As a result of this, the resulting prototype is more object-oriented and consequently easier derived.

2 VDM

VDM was originally developed in the IBM Vienna Research Laboratories, starting back in 1973 and it has evolved to be one of the most well established methods for rigourous software development. A complete account of the language is given in [Jon86].

VDM is based in first-order logic and set theory. The method suggests the development of software through successive refinements. From a very abstract initial specification, others are derived until, after data and operation refinements, the final implementation is constructed. The consistency, that is to say, the semantic equivalence between the levels of refinement must be rigorously verified.

The type system is based on set theory. Types are sets of values. In addition to basic types, such as Integer and Boolean, VDM offers some type constructs and related operations. These types, which include sets, mappings and sequences, are used for data structuring.

A mathematical model is constructed for the software, defining the state of the application and operations over it. Auxiliary functions may also be defined.

An operation is defined by specifying its pre- and post-conditions. Furthermore, operations must indicate which components of the state they read and/or write. Functions do not affect the state.

There is no data encapsulation. A model defines a concrete object upon which the operations actuate, not a data type nor a class that has instances. There are no modularization mechanisms and reusability is difficult. One can only reuse functions and pre- and post-conditions in a specification.

Further, we present a partial specification of a process scheduler of an operating system, which is in accordance with the style of specification supported by SmallVDM. The specification incorporates some object-oriented characteristics and is said to be object-based.

3 Smalltalk

Smalltalk is considered one of the languages most true to the object-oriented paradigm [Pin88]. Its philosophy and implementation are extensively discussed in [Gol83].

In Smalltalk, every object is an instance of a class, including the classes themselves. An executing program consists of a set of objects that communicate sequentially through message exchange. Some other object-oriented languages introduce new concepts to the paradigm, such as that of multiple-inheritance, but the basic concepts are all supported by Smalltalk.

Smalltalk is more than a programming language. It is a programming environment, composed of a sophisticated interface, a set of tools such as editors and browsers, and an extensive library of pre-defined classes from which the programmer can derive his own.

Although not very efficient because interpreted, that is fundamental to provide the exploratory style of programming embodied in the Smalltalk philosophy, through fast environment interaction. The language is also typeless and some errors may occur during execution.

Smalltalk is also considered a prototyping language. Some of the characteristics mentioned above reinforce that character, although many applications developped for production seem to contradict it.

Some pieces of code generated for the prototype of the process scheduler mentioned in the previous section will be listed in this work.

4 SmallVDM

SmallVDM tries to conciliate VDM and Smalltalk in a sole framework, making good use of the best they happen to offer.

VDM makes possible the formalization of the design and contributes with its simplicity and model-orientation. Smalltalk, in turn, apart from being purely object-oriented, presents a sophisticated development environment passive to extension and most adequate to prototyping.

In a few words, we can define SmallVDM as an interpretation of VDM in a Smalltalk environment that allows rapid construction of prototypes from specifications. This definition suggests two views of the tool: specification animation and formal object-oriented development.

Specification animation was the initial motivation which led us to build a VDM environment in Smalltalk, allowing direct transformation of VDM specifications into Smalltalk code, in spite of the style (paradigm) adopted in the construction of the model and respective operations.

Some specifications, such as the semantics of SQL presented in [Mei90a], and that of an object-oriented database model presented in [Mot90], were animated in this environment, allowing the validation of specifications in a semi-automatic fashion, as discussed in [Mot90].

Changing the point of view leads to a second interpretation of SmallVDM, this time as a tool for formal object-oriented programming. Now the goal is the construction of an object-oriented system, and the paradigm must be taken into account during the process of formal specification.

As discussed before, VDM was not designed for the specification of object-oriented systems. We believe, however, that it can be used to express many of the characteristics of such systems if some criteria are adopted for the construction of the specification. These criteria constitute a style of specification, which can be enforced by the environment where the specification is developed. On-line development of specifications is the key for assuring the object-orientation and providing reusability and extensionability.

SmallVDM not only provides the animation of object-based specifications but it also assists the specification process, ensuring that no violation of the object-oriented design philosophy occurs.

5 Object-Oriented Design

Research into object-oriented design has increased as its acceptance grows wider. Many attempts have been made to find new rules for good design and to build tools to support them. Some of these are discussed in [Wir90].

In what follows, we assume the division of software development in three basic phases: analysis, design and implementation. The objects of discussion during analysis are just the same as during design and implementation. That makes the transition from one phase to the next one more natural and straightforward, as the domain of discussion is uniform from the client to the programmer. Further considerations can be found in [Kor90].

The integration of formal design and implementation in the same framework is more directly achieved and that is the approach undertaken by Meyer in the Eiffel programming language, which allows the incorporation of formal requirements in the program code. The contracts established between objects are explicitly expressed as part of the implementation of classes. Eiffel and its philosophy are described in [Mey88].

Objects identified during the analysis of the problem must be defined in more detail in the design phase, which must be language-independent, but not paradigm-independent, so as to provide the best utilization of its characteristics.

One can implement an object-oriented design in a traditional procedural language, by doing away with inheritance. The implementation of a procedural design in an object-oriented language, however, results in waste.

In the light of those considerations, we conclude that the specification of an object-oriented system must come after high-level design. By high-level design we call the task of defining in detail the objects of the domain, their interfaces, the classes and dependence relations among them. The formal specification of operations and auxiliary functions will be called low-level design.

SmallVDM does not support the analysis phase. That preceeds the design and comprehends the identification of the objects in the domain of application, as well as the classes from which these objects will be instances and the hierarchy structure they compound. Details of functionality and interface are postponed to the next phase.

5.1 High-Level Design

The high-level design phase is supported by SmallVDM in many ways. When working in the environment, one is always in the context of one application. Each application has its own dictionary of classes and classes may be shared by different applications.

The type definitions in the specification are recorded in the environment. Each type gives origin to a class, which is created automatically. A new class is either origin of a new hierarchy or the extension or specialization of an existing class. The definition of a type may generate new type definitions. When defining a type, the designer is asked to insert the type in the system's hierarchy structure defined up to that moment. If no insertion is possible he is asked to check the global hierarchy of classes, to see whether an insertion is still possible or not. A graphical representation of the hierarchy is available, from which the designer may extract all necessary information about the classes and the hierarchy structure itself. The graph of the hierarchy is not necessarily connected, and types can be extensions of types defined in other applications.

The system stimulates the definition of abstract classes (classes having no instances), used to gather characteristics that are common to its subclasses, in order to promote reusability.

A non-empty set of general concepts must be associated to each new type (class) being defined, helping to identify abstract classes and to position a class in the system's hierarchy.

Informal description of types and their attributes must be given. A type is associated to each attribute of a type (field types in a composite type definition, domain and co-domain types in a map definition, etc.). The environment associates to each class a list of all its clients, i.e. all classes whose instance components may be instances of it. Such list is very useful when maintaining a class.

In opposition to Smalltalk philosophy, the type definition of attributes is a VDM requirement that is essential for the control of the specification. During the execution of the derived prototype, the environment performs parameter type checking in order to detect possible parameter passing errors in the specification. A final version of the implementation, when errors are no longer expected to occur, may discard type checking, adopting the Smalltalk philosophy again. By then, there should be no possibility of run time typing errors to occur.

Each class resulting of a type definition is a subclass of one of the following pre-defined classes of the environment, which implement the types available in VDM for the construction of data structures:

- Mapping
- Set
- Record
- Sequence
- TypeUnion
- GivenSet
- Enumeration

Record implements composite types. The names and types of its attributes are informed to the environment and access methods are automatically generated. In Smalltalk, by default, methods are public and components are private. SmallVDM constrains the access according to the specification (*ext* clauses, specifying which components of the state are used for reading and writing in an operation). Any attempt to disobey the specification is precluded by the environment.

TypeUnion is used to specify polymorphic structures. Whenever such structures happen to be necessary, the set of all possible types in a polymorphic ocurrence is used to define a union of types and simulate the (constrained) polymorphism.

GivenSet is used for the representation of basic types, such as Integer and Boolean. The interpretation of these types as given sets makes specifications more natural and homogeneous in the environment, but the original Smalltalk implementation is kept (classes Integer and Boolean for the types mentioned above).

The class *Set* used by the environment is not that in the original Smalltalk image, but a new one incorporating some of the old characteristics, as expected, but extending them with quantification, parameter type checking, etc.

Enumeration is used in the implementation of types with finite domains. These types are defined by the enumeration of the elements in their domains.

The usual VDM operations are defined for each of these classes. The correspondence between some VDM constructions and the environment created in Smalltalk for their implementation is shown below.

	VDM	Smalltalk
LOGIC:		
	$\forall e \in s \cdot < exp >$	`s forAll:[:e|<exp>]`
	$a \Rightarrow b$	`a implies: [b]`
	$a \vee b$	`a or: [b]`
SET:		
	$\mathrm{card}\, s$	`s card`
	$s \subseteq t$	`s isSubsetOf: t`
	$a \notin t$	`(t includes: a) not`
MAP:		
	$\mathrm{rng}\, s$	`s rng`
	$s \lhd m$	`s domainRestriction: m`
	$s ⩤ m$	`s domainDeletion: m`
COMPOSITE OBJECTS:		
	$s(n)$	`n s`
	$\mu(n, s \mapsto t)$	`n s:t`
SEQ:		
	$\mathrm{len}\, s$	`s len`
	$s \frown [t]$	`s add:t`
	$\mathrm{tl}\, s$	`s tl`

5.2 Low-level design

Although classes can be expressed as abstract data types, VDM doesn't provide mechanisms for defining such abstract types. A VDM model can be seen as an abstract data type, but there is no encapsulation, with models representing concrete objects and not classes which can have instances.

In spite of that, an object-based style of specification can be adopted in order to structure the specification and to create a smooth path to an object-oriented implementation.

The specification is constructed on-line in an interactive environment. Facilities including a syntatic editor and procedures for type and operation definition direct the structure of the specification.

As mentioned before, a type defined in the specification corresponds to a class in the environment. The methods defining its functionality are specified in VDM notation as *associated functions* of that type. This association is provided by the environment but it is not (formally) reflected in text of the specification. Internally, however, the environment maintains these associations, as well as the inheritance hierarchy structure and encapsulation requirements. The text of the specification, which is generated by the environment in LaTeX format, reflects informally, i.e. by means of comments, the characteristics which could not be formally expressed.

Associated functions are specified as functions whose first parameter and the value returned have the same type and are viewed internally as operations. *Auxiliary functions*, which are not viewed as operations, must also be associated to a type, which in this case is the type of their first parameter. A third kind of function, which has no parameter, is associated to the type of the value that it returns and is called a *constructor function*. Examples of these functions will be presented later.

The specification obtained is object-based. The inheritance structure is not formally specified but, as mentioned above, internally maintained by the environment. The specification of a class as a subclass of another one is an extension of the superclass specification and the characteristics of the subclasses are attached to their superclasses as if a new and unique class was being defined. The environment provides the separation between the two definitions, keeping reusability and imposing encapsulation pragmatically.

When defining an operation, the designer must fill a form, where all information necessary for maintenance by the environment is given. That includes the definition of the operation's and arguments' names and respective types, *ext rd* and *ext wr* clauses and the pre- and post-conditions. The environment checks the consistency of the given data with respect to the rest of the system.

The integration of tools such as a syntatic editor and a pretty-printer, which formats the final text of the specification, is intended to stimulate the practice of formal specification by releasing the designer of manual work.

6 Prototyping

The prototyping phase corresponds to that of implementation in the analysis-design-implementation schema. Some aspects concerned with implementation, such as efficiency and the definition of sophisticated interfaces for the user of the system are not taken into account when constructing a prototype.

The generation of Smalltalk code from specification is semi-automatic. The pre-defined types of VDM are implemented as classes in SmallVDM. These classes are the basis for the implementation. The environment generates most of the code, making short work of the implementation.

Pre- and post-conditions are automatically translated and when operations are executed,
pre-conditions are tested first. After the execution, post-conditions can also be tested, thus verifying the consistency of the specification with respect to the implementation.

Translation of post-conditions and *associated functions* into methods may involve some transformations which must be carried out by the designer in order to reduce the level of abstration and to allow automatic coding. These transformations, however, are easily carried out and may be avoided by adopting a style where more concrete (direct) definitions of functions and post-condictions are built from the very beginning.

The verification of the semantic equivalence between different levels of abstraction is not imposed by the environment. It's up to the designer to verify the consistency of transformations but the system may assure consistency by testing pre- and post-conditions, as well as type invariants, during execution.

A generic interface is used to test the prototype and there is no need to write code for input and output, which is automatically provided by the environment according to parameter and result types. The internal state of an application can be checked at any time and many versions of the state may be stored to allow different kinds of test to be carried out simultaneously.

7 An Object-Based Specification

We now present a partial specification of a process scheduler, which is the portion of an operating system that organizes the access of processes to processors. The scheduler presented here is based on that of the Minix operating system [Tan87].

Each process has an associated identification and a process table stores information about them. A process can be in one of the following states: *Ready*, *Executing*- and *Blocked*. The possible state transitions are those shown in Figure 1.

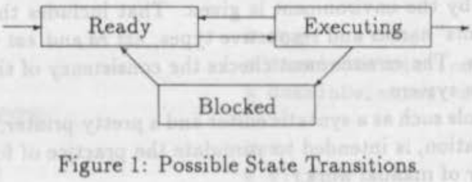

Figure 1: Possible State Transitions

There is only one processor and access to it is granted according to priority levels and, for processes with the same priority, according to the order in which they get ready for execution. The scheduler can be modelled in many different ways. Our model is based in two elements, a table of processes and a queue of priorities, as shown below.

7.1 Data Type Specification

The first phase of the specification consists in the definition of the type of the objects identified during analysis. Classes are generated automatically for each defined type.

PRIORITY and *STATUS* are defined as enumerations. The type *PRIORITY* is the set of the natural numbers from 1 to n. The constant n must be given a concrete value for the environment cannot operate on undefined values which are, however, preserved in the text of the specification.

$PRIORITY = \{1, \ldots, n\}$

The type *STATUS* is the set of the three possible states of a process, as specified below.

$STATUS = \{Ready, Executing, Blocked\}$

IDENTIFIER and *OTHER* are defined as given sets. They may be specified as undefined types but the environment demands their definition. As for constants, the indefinition is preserved in the text of the specification.

IDENTIFIER = Integer

$OTHER = String$

OTHER represents the type of the other components of a process which are not relevant to the scheduler specification. It could be discarded not only in the specification, but also in the implementation.

The definition of a type to represent a queue of process identifiers is given below. The operations on objects of that type assure their behaviour as a queue, where a FIFO (First In First Out) policy is assumed. Its invariant determines that no identifier may occur more than once in a queue of that type.

$IDQUEUE = \textsf{seq of}\ IDENTIFIER$

where

$inv\text{-}IDQUEUE() \ \triangleq\ \forall f \in IDQUEUE \cdot \textsf{len}\, f = \textsf{card}\,\textsf{rng}\, f$

PROCESS is a composite type. The *pri* component represents the priority level of the process, *st* is the status of the process and *other* represents other data not relevant to the specification.

$$\begin{aligned} PROCESS :: \quad pri &: PRIORITY \\ st &: STATUS \\ other &: OTHER \end{aligned}$$

The definition of a type to represent a table of processes suggests a generalization. The type TABLE is defined as:

$TABLE(X, Y) = \textsf{map}\ OBJECT\ \textsf{to}\ OBJECT$

OBJECT is a pre-defined type in the environment. It is the type union of all the types defined in the environment and is used in generic definitions. Other type unions may be defined for that purpose.

The type *TABLEOFPROCESSES* is an instance of *TABLE*:

$TABLEOFPROCESSES = TABLE(IDENTIFIER, PROCESS)$

It is important to notice that *TABLEOFPROCESSES* could have been defined as a mapping, without resorting to the definition of *TABLE*. This intermediate type, however, is intended to define an abstract class where many general operations on tables could be defined and reused later on. These operations include insertion and remotion of entities, table look-up, etc. With the exception of inclusion and updating, such operations will not be shown here and their existence is taken for granted.

We are aware that inheritance can not be simulated by type parameterization [Mey88], but reusability is achieved and the environment is able to reflect the inheritance structure in the implementation.

There is a queue associated to each level of priority, so there are n queues of process identifiers, with the processes associated to identifiers in those queues all in the *Ready* state. The n queues are grouped in a table which associates priorities to them. The invariant of *TABLEOFQUEUES* asserts that a process identifier cannot occur in distinct queues.

$TABLEOFQUEUES = TABLE(PRIORITY, IDQUEUE)$

where

$$\begin{array}{l} inv\text{-}TABLEOFQUEUES() \quad \triangleq \\ \quad \forall p \in IDENTIFIER \cdot \forall f, g \in IDQUEUE \cdot p \in \mathsf{rng} f \wedge p \in \mathsf{rng}\, g \;\Rightarrow\; f = g \end{array}$$

The state is composed of a table of processes and a table of queues. Its invariant determines that every identifier ocurring in a queue will have an entry in the table of processes and that the associated process is *Ready*. It also asserts that there is at most one process in execution at any time.

$$\begin{array}{lll} SCHEDULER :: & table & : TABLEOFPROCESSES \\ & queues & : TABLEOFQUEUES \end{array}$$

where

$$\begin{array}{l} inv\text{-}SCHEDULER(mk\text{-}SCHEDULER(table, queues)) \quad \triangleq \\ \quad \mathsf{card}\,\{p \in \mathsf{rng}\, table \mid st(p) = Executing\} < 2 \wedge \\ \quad \forall p \in PRIORITY \cdot \\ \qquad \forall id \in queues(p) \cdot \\ \qquad\qquad id \in \mathsf{dom}\, table \wedge st(tabela(id)) = Ready \wedge pri(table(id)) = p \end{array}$$

As types are defined interactively, the environment processes the information being given and is able to present them in the VDM format shown above, along with the informal comments added to each type and its components.

7.2 Specification of Operations

The limitations of VDM with respect to modularization lead to monolithic specifications. SmallVDM provides modularization internally, by associating operations, which are defined as *associated functions*: when defining an operation, the designer must first identify the type to which it will be associated.

This is a partial specification, so not every operation of the scheduler will be shown, but the operations depicted illustrate many of the concepts and ideas discussed up to this point.

7.2.1 TABLE

As mentioned, many reusable operations can be defined over objects of type *TABLE* but only those for insertion, update and creation will be shown.

$$\begin{array}{l} InsertInTable\ \ (table\colon TABLE(OBJECT, OBJECT) \\ \qquad\qquad\qquad key\colon OBJECT \\ \qquad\qquad\qquad value\colon OBJECT)\ newTable\colon TABLE(OBJECT, OBJECT) \\ \mathsf{pre}\ key \notin \mathsf{dom}\, table \\ \mathsf{post}\ table \cup \{key \mapsto value\} \end{array}$$

InsertInTable inserts an element in the table. The associated key cannot already have an entry in the table. In *UpdateTable* the key must be in the table so that the associated object can be updated.

UpdateTable (*table*: *TABLE*(*OBJECT*, *OBJECT*)
key: *OBJECT*
value: *OBJECT*) *newTable*: *TABLE*(*OBJECT*, *OBJECT*)

pre $key \in \text{dom}\ table$

post $table \dagger \{key \mapsto value\}$

InsertInTable and *UpdateTable* are *associated functions*, i.e. functions that represent operations over a type. The next function is an example of a *constructor function*, which has no parameter and is associated to the type of the value it returns.

EmptyTable *t*: *TABLE*(*OBJECT*, *OBJECT*)

post $t = \{\}$

7.2.2 TABLEOFPROCESSES

The operation that inserts a process in the table of processes and sets its state to *Ready* is specified by

InsertProcessInTable (*table*: *TABLEOFPROCESSES*
identifier: *IDENTIFIER*
process: *PROCESS*) *newTable*: *TABLEOFPROCESSES*

pre $identifier \notin \text{dom}\ table$

post let $p = SetStatus(process, Ready)$ in
$newTable = InsertInTable(table, identifier, p)$

SetStatus is associated to type *PROCESS* and will be defined soon. The next operation "executes" the process associated to the identifier passed as parameter.

ExecuteProcess (*table*: *TABLEOFPROCESSES*
identifier: *IDENTIFIER*) *newTable*: *TABLEOFPROCESSES*

pre $identifier \in \text{dom}\ table$

post $newTable = table \dagger \{identifier \mapsto SetStatus(table(identifier), Executing)\}$

7.2.3 PROCESS

SetStatus was used above and defines an operation that changes the *st* component of a process to be that passed as parameter to the function.

SetStatus (*process*: *PROCESS*
newStatus: *STATUS*) *newProcess*: *PROCESS*

post $\mu(process, st \mapsto newStatus)$

7.2.4 TABLEOFQUEUES

Constructor functions sometimes need information in order to initialize the object they create. That is the case of *EmptyTableOfQueues*. There is no point in associating it to the type of its first parameter, as if it was an auxiliary function. The designer must be aware of the semantics behind the definitions so as to make the proper associations.

EmptyTableOfQueues (n) *table*: *TABLEOFQUEUES*

post $\forall i \in PRIORITY \cdot table(i) = [\,] \wedge \text{card dom } table = \text{card } PRIORITY$

InsertIdInQueue inserts a process identifier in the queue associated to the priority of the process the identifier relates to.

InsertIdInQueue (*table*: *TABLEOFQUEUES*
identifier: *IDENTIFIER*
priority: *PRIORITY*) *newTable*: *TABLEOFQUEUES*

pre $identifier \notin \text{rng } table(priority)$

post $newTable = UpdateTable(table, priority, table(priority) \frown [identifier])$

The next two functions are *auxiliary*, being associated to the type of their first parameter. When many parameters are passed to such functions, some considerations must be made in order to determine which of them best fits in the first position.

PriorityOfNextToExecute (*table*: *TABLEOFQUEUES*) *p*: *PRIORITY*

pre $\exists i \in PRIORITY \cdot \text{len } table(i) > 0$

post $p = min(\{i \in PRIORITY \mid \text{len } table(i) > 0\})$

The function *IdentifierOfNextToExecute* returns the identifier associated to the next process to be executed, whilst the previous one returned its priority. Both functions use another auxiliary function (*min*) which is associated to the type *PRIORITY*. It returns the smallest integer in the set passed as parameter. As types are sets of values, there is no problem in associating this function to the type *Priority*.

IdentifierOfNextToExecute (*table*: *TABLEOFQUEUES*) *id*: *IDENTIFIER*

pre $\exists i \in PRIORITY \cdot \text{len } table(i) > 0$

post $\text{hd } table(min(\{i \in PRIORITY \mid \text{len } table(i) > 0\}))$

The function *RemoveFromQueue* removes from the table of queues the identifier of the next process to be executed.

RemoveFromQueue (*table*: *TABLEOQUEUES*) *newTable*: *TABLEOFQUEUES*

post let $p = PriorityOfNexToExecute(table)$ in
$newTable = UpdateTable(table, p, \text{tl } table(p))$

7.2.5 Scheduler Operations

The definition of operations on the state of the application makes use of the operations defined as functions associated to the types of its components and sub-components.

The operation *InitialState* initializes the internal state of the scheduler.

InitialState

ext wr *table* : *TABLEOFPROCESSES*
  wr *queues* : *TABLEOFQUEUES*

post $table = EmptyTable() \wedge queues = EmptyTableOfQueues$

The entry of a new process to be scheduled comprehends its insertion in the table of processes and of its identifier in the queue associated to its priority in the table of queues.

InsertProcess (identifier: IDENTIFIERprocess: PROCESS)

ext wr *table* : *TABLEOFPROCESSES*
  wr *queues* : *TABLEOFQUEUES*

post $table = InsertProcessInTable(\overleftarrow{table}, identifier, process) \wedge$
  $queues = InsertIdInQueue(\overleftarrow{queues}, identifier, pri(process))$

A process is *Executing* when it has access to the processor, passing from the *Ready* to the *Executing* state.

FromReadyToExecuting (identifier: IDENTIFIERprocess: PROCESS)

ext wr *table* : *TABLEOFPROCESSES*
  wr *queues* : *TABLEOFQUEUES*

pre $\nexists p \in \mathrm{dom}\, table \cdot st(table(p)) = Executing$

post let $id = IdentifierOfNextToExecute(\overleftarrow{queues})$ in
  $table = ExecuteProcess(\overleftarrow{table}, id) \wedge queues = RemoveFromQueue(\overleftarrow{queues})$

The other state transitions are specified in the same way as *FromReadyToExecuting*, by making use of the functions previously defined.

8 Generating the Prototype

Once the system is completely specified, the designer can construct its prototype semi-automatically. Type invariants, pre- and post-condititons are automatically translated to Smalltalk and checked during execution. As discussed before, some post-conditions must be transformed into more directly implementable definitions but these changes are easy to perform.

The code generated for some operations specified in the previous section for the process scheduler is now presented.

The operation *InsertProcessInTable*, which was specified as a function, is implemented as a method of class TableOfProcesses. The code derived for the function definition is presented below.

```
InsertProcessInTable:parameters
 | identifier process p |
 identifier := parameters at:1. process := parameters at:2.
 (self dom includes:identifier) not
      ifFalse:[self error:'Pre-condition not satisfied
                          for InsertProcessInTable'].
 p := process changeStatus:(Array with:'Ready').
 self InsertInTable:(Array with:identifier with:p)
```

Pre- and post-condition testing is normally implemented as separate methods as to make them reusable. The inclusion of the test of the pre-condition in the method above was intended to ease the presentation.

The first parameter in the specification is removed in the implementation and references to it, whose type determines the class where the method is to be implemented, are transformed in *self* references in Smalltalk (a reference to the receiver of the message itself). The assignment of the value returned by the function in the specification to the object passed as first parameter is substituted by a message sent to that object, as we can verify in the implementation of *InsertProcess*:

```
InsertProcess:parameters
  | identifier process |
  identifier := parameters at:1.
  process := parameters at:2.
  table  InsertProcessInTable:(Array with:identifier with:process).
  queues InsertIdInQueue:(Array with:identifier with:process pri)
```

The methods *InsertIdInQueue*, from class TableOfQueues and *UpdateTable*, from Table, are listed below.

```
InsertIdInQueue:parameters
 | identifiers priority  |
 identifier := parameters at:1.
 priority := parameters at:2.
 ((table at:priority) rng includes:identifier) not
    ifFalse:[self error:'Pre-condition not satisfied
                        for InsertIdInQueue'].
 self UpdateTable:(Array with:priority
                        with:((self at:priority) add:identifier))
```

```
UpdateTable:parameters
 | key value  |
 key := parameters at:1.
 value := parameters at:2.
 (self dom includes:key)
    ifFalse:[self error:'Pre-condition not satisfied
                        for UpdateTable'].
 self at:key put:value
```

9 Conclusions

SmallVDM as an animation tool has been successfully used for verification and validation of specifications [Mot90]. Executable programs have been easily derived and tested with the help of a set of tools put together to support a development methodology.

No modification was proposed in the specification language, contrary to many other approaches on formalism in object-oriented design. This possibility, however, is not completely discarded and some attempts to give modularization mechanisms to VDM are described in [Mid90].

Our approach concentrates initially in the definition of a development methodology based in a established language: VDM offers the language and the development method from specification to implementation through successive refinements.

Programming environments for supporting the programming activity, such as Smalltalk, can be extended to assist formal design and carefully chosen tools can reduce the amount of manual work. The paradigm can be enforced from the specification phase, with software reusability coming as consequence of specification reusability. Classes as modules must be properly managed to facilitate reuse and a good class manager, at specification level, must provide most reusability by assisting the definition of new class hierarchies and compartmentalising specifications.

The limitations of VDM for expressing object-oriented characteristics led to the definition of a style of specification supported and imposed by the SmallVDM environment. The experience has proved to be worthwhile and has indicated some directions towards the definition of a proper formalism for object-oriented systems. The MooZ group, which is working in the definition of an object-oriented extension of Z, as described in [Mei90b], has based some of its decisions in the results obtained from SmallVDM. The final goal is the definition of an environment for supporting object-oriented development starting from MooZ specifications, and many features of SmallVDM are to be reused. Some of these, such as the syntactic editor for the VDM notation, are under implementation. Most of the environment is already operational and has been used to support the method exposed herein.

References

[Dah87] O. J. Dahl: "Object-Oriented Specifications". Research Directions in Object-Oriented Programming - MIT Press Series in Computer Systems, 1987.

[Duk90] D. Duke and R. Duke: "Towards a Semantics for Object-Z". Proc. VDM'90, Springer-Verlag, Kiel-FRG, April 1990.

[Gol83] A. Goldberg and D. Robson: "Smalltalk-80: The Language and its Implementation". Addison-Wesley, 1983.

[Hoa84] C. A. R. Hoare: "Programming: Sorcery or Science". IEEE Software, April 1984.

[Jon86] C. B. Jones: "Systematic Software Development Using VDM". Prentice-Hall International, 1986.

[Kor90] T. Korson and J. D. McGregor: "Understanding Object-Oriented: A Unifying Paradigm". Communications of the ACM, September 1990.

[Mei90a] S. R. L. Meira, R. Motz and J. F. Tepedino: "A Formal Semantics for SQL". Intern. J. Computer Math.,Vol 34, pp. 43-63, 1990.

[Mei90b] S. R. L. Meira and A. L. C. Cavalcanti: "Modular Object-Oriented Z Specifications". Z Technology and Users Meeting, Springer Verlag Workshopes in Computing, May 1991.

[Mey88] B. Meyer: "Object-Oriented Software Construction". Prentice-Hall International, 1988.

[Mid90] C. A. Middelburg: "Syntax and Semantics of VVSL". Ph D Thesis - University of Amsterdam, September 1990.

[Mot90] R. Motz: "Formal Analysis of an Object-Oriented Data Model". Master's Thesis - Departamento de Informática - UFPE. (In Portuguese)

[Pin88] L. J. Pinson and R. S. Wiener: "An Introduction to Object-Oriented Programming and Smalltalk". Addison-Wesley, 1988.

[Tan87] A. S. Tannenbaum: "Operating Systems: Design and Implementation". Prentice-Hall International, 1987.

[Tep90] J. F. Tepedino, R. Motz and S. R. L. Meira: "From Model-Based Specifications to Object-Oriented Prototypes - A Method". X Congresso da SBC, Vitória, Brazil, July 1990.

[Wir90] R. J. Wirfs-Brock and R. E. Johnson: "Surveying Current Research in Object-Oriented Design". Communications of the ACM, September 1990.