Antony Bryant and Lesley Semmens (Eds)

Methods Integration

Proceedings of the Methods Integration Workshop, Leeds, 25-26 March 1996

Paper:

Integrated Formal and Object-Oriented Methods: The VDM ++ Approach

K. Lano and S. Goldsack

Published in collaboration with the
British Computer Society

INTEGRATED FORMAL and OBJECT-ORIENTED METHODS: The VDM⁺⁺ Approach

K. Lano

Dept. of Computing, Imperial College, 180 Queens Gate,
London, SW7 2BZ, UK

S. Goldsack

Dept. of Computing, Imperial College, 180 Queens Gate,
London SW7 2BZ

Abstract

This paper describes methods integration techniques and tools developed for the VDM⁺⁺ formal specification language. Case studies which have been carried out for the language using tool support for the integrated development method are also described. The original features of the method are its use of highly abstract initial specifications to avoid the design bias of object-oriented structuring, and the formal treatment of structural transformations.

1 Introduction

Object-oriented specification languages are an area of significant potential within software engineering, and aim to combine the benefits of precise mathematical notations with the advantages of object-oriented structuring mechanisms. Languages in this field include Object-Z [6], ZEST [4], VDM⁺⁺ [7], Z⁺⁺ [16] and MooZ [23]. In a separate line of development are object-oriented analysis and design methods which incorporate some elements of formality. Fusion [2] and Syntropy [3] are the two main methods of this form.

Object-orientation and formal techniques are highly compatible in a number of ways:

- object-orientation encourages the creation of abstractions, and formal techniques provide the means to precisely describe such abstractions;
- object-orientation provides the structuring mechanisms and development disciplines needed to scale up formal techniques to large systems;
- formal techniques allow a precise meaning to be given to complex object-oriented mechanisms such as aggregation or statecharts.

There are two alternative ways of integrating diagrammatic and formal methods which have been proposed and used in practice by groups and companies working in this area:

- *covert formality*, in which mathematical notation and an explicitly formal specification language may be avoided, but with a mathematical semantics being provided for structured English or diagrammatic specification forms;
- *overt formality*, involving translation processes from a diagrammatic notation into a formal notation, and sometimes in the reverse direction as well, and the use of development steps within the formal specification and design language.

The Fusion method of Hewlett-Packard epitomises the former approach, whilst the VDM⁺⁺ method described in this paper adopts the latter.

The covert approach, whilst more immediately acceptable and usable within industrial software development, may not be as effective as the overt approach when formal reasoning about systems is required.

Section 2 summarises the method adopted for VDM⁺⁺, Section 3 introduces the language and its notation, and Section 4 describes the translation process between OMT [27] and VDM⁺⁺ notations. Section 5 gives an example of the process, and Section 6 describes an industrial case study.

2 Development Process

The VDM⁺⁺ development process, termed IFDM or “iterative formal development method” is as follows:

1. Requirements analysis, using the object models and dynamic models of OMT [27], and other notations, such as the object interaction diagrams of Booch [1];
2. Translation of object models into initial VDM⁺⁺ class specifications (this step is automated by the *Venus* toolset for the language). The initial (Model_0) specification can consist of a single class which describes the required operations with no assumptions about the classes that will exist in the design or the decompositions of operations into method calls;
3. Identification of VDM⁺⁺ classes and their dynamic behaviour specifications and method definitions. This step involves *localisation* of system operations into a collection of methods located in particular classes – a Model_1 specification;
4. Refinement and design in VDM⁺⁺. This step uses both conventional VDM-SL data refinement concepts and new concepts such as *annealing* – refining one class by a collaborating collection of several classes. It yields a series of models related by refinement;
5. Code generation in C++ or other OO programming languages. At present a C++ code generator exists [29], and an Ada95 generator is under development. Code generation can either be used to support animation via an executable prototype at the specification stage, or to produce production quality code from specifications which have been refined into a fully procedural form.

Internal consistency proof obligations can be generated and checked at specification and refinement stages, to ensure that formal classes actually have models and can be implemented. A description of these conditions is given in [15]. Proof obligations for subtyping and for refinement steps can also be generated, based on those for conventional VDM-SL refinement. These obligations ensure that a development step does represent progress towards an executable implementation. They are described in [15].

Diagrammatically, the process and tool-supported actions are as shown in Figure 1.

As with Fusion and Syntropy, IFDM emphasises the need to separate models of real-world situations from models of the required software. It extends these methods with the capability to define mathematical relationships between such models.

The key aspect of VDM⁺⁺ is that it allows all the separate requirements upon an entity to be combined in a single textual class description. Thus the static data requirements, described in OMT object models, yield a specification of the data elements of a VDM⁺⁺ class, whilst the timing and state-transition behavior requirements, described in timed statecharts, yield definitions of the synchronisation and thread parts of a VDM⁺⁺ class. Operation (functional) models of OMT, and object interaction graphs, which express processing requirements and initial designs, can be used to define methods and threads of a class. This integration of the separate aspects of an entity is a natural step towards implementation in a programming language (such as Ada) where these aspects are combined in the description of single modules. It also enhances the capability for consistency checking between these aspects.

This paper will consider the formalisation of the object model and dynamic models of OMT. Formalisation of object interaction diagrams is covered in [18].

Figure 1: Development Process

2.1 Tool Architecture

The *Venus* toolset combines an established CASE tool for OMT, *LOV*, produced by Verilog, with a toolset for VDM⁺⁺ notation. The latter, known as *VPP*, has been developed by IFAD and is based on their established VDM-SL toolset. The two tools are linked by a *coupling module* which provides translations between the different internal representations used by the tools. The level of integration is quite high, in that the translation process covers most of the OMT object model notation and the VDM⁺⁺ notations, but is not complete, because of the use of two different representations. Both tools use a Motif or Motif-like interface, and provide similar ways of working, so that it is not difficult to combine their use.

Constructs in VDM⁺⁺ which have no counterpart in OMT (such as the detailed definitions of methods) are translated into special purpose “notes” in the *LOV* representations. This permits them to be viewed and edited via the *LOV* tool and to be translated back into the appropriate VDM⁺⁺ constructs. Certain constructs in OMT have no counterpart in VDM⁺⁺, and are not translated. For instance, the distinction between inclusive and exclusive subtyping is not made in VDM⁺⁺. In addition, only the strongest form of aggregation (bound, unreplaceable, unsharable containment) is represented in VDM⁺⁺ [15].

3 VDM⁺⁺: Syntax and Semantics

VDM⁺⁺ is a concurrent object-oriented extension of the VDM specification language [8, 7]. The aim of defining the language was to use the concepts of object-orientation to provide structuring facilities for VDM specifications. In addition it was intended to extend the capabilities of VDM to a wider range of systems, including concurrent, reactive and real-time systems. VDM⁺⁺ is one of the most advanced of the present generation of object-oriented formal languages [14].

VDM⁺⁺ has been adopted by a major European software firm, CAP, and has also been industrially trialed in the AFRODITE ESPRIT project. Industrial case studies of the language include a minimum safe altitude warning component for an air traffic control system, ship load planning and monitoring, and elements of the next generation of particle

detector devices at CERN.

Applications of the language have made essential use of the extended capabilities of VDM⁺⁺. In particular, the CERN case study has extremely stringent timing requirements, whilst the use of inheritance and subtyping in the air traffic control system was believed to have improved the ability to validate the system [17].

3.1 VDM⁺⁺ Syntax

A VDM⁺⁺ specification consists of a set of *class* definitions, where these have the general form:

```

class C
types
  T = TDef
values
  const: T = val
functions
  f: A → B
  f(a) == Defnf(a)
instance variables
  vC: TC;
inv objectstate == InvC;
init objectstate == InitC
methods
  m(x: Xm,C) value y: Ym,C
    pre Prem,C(x, vC) == Defnm,C;
  ...
sync ...
thread ...
aux reasoning ...
end C
    
```

A class is a generalisation of the VDM module concept (eg, as implemented in the IFAD VDM toolbox). It uses methods in place of operations, and method invocation in place of “quoting” operation pre and post conditions. As such, it has a more systematic basis for its structuring mechanisms, and one that is more familiar to industrial practitioners.

The **types**, **values** and **functions** components define types, constants and functions as in conventional VDM (although class reference sets @D for class names D can be used as types in these items – such classes D are termed *suppliers* to C, as are instances of these classes. C is then a *client* of D). The **instance variables** component defines the attributes of the class, and the **inv** defines an invariant over a list of these variables: **objectstate** is used to include all the attributes. The **init** statement defines a set of initial states in which an object of the class may be at object creation. Object creation is achieved via an invocation of the operation C!new, which returns a reference to a new object of type C as its result.

The methods of C are listed in the **methods** clause. The list of method names of C, including inherited methods, will be referred to as methods(C) in the following. The subset of methods(C) which may be used by external clients of C is extmethods(C), whilst the subset that can be only used internally is intmethods(C). The union of extmethods(C) and intmethods(C) is methods(C). Internal methods arise as a result of inheritance steps (*controlled inheritance*) that remove ancestor methods from the external interface of the descendent.

Methods can be defined in an abstract declarative way, using *specification statements*, or by using a hybrid of specification statements, method calls and procedural code. Input parameters are indicated within the brackets of the method header, and results after a **value** keyword. Preconditions of a method are given in the **pre** clause. The form of a specification statement is:

```

[ext wr write frame
  rd read frame
  pre precondition
    
```

post precondition]

In the **postcondition** the value of an attribute **att** at initiation of execution of the specification statement is denoted by $\overline{\text{att}}$.

Other clauses of a class definition control how **C** inherits from other classes: the optional **is subclass of** clause in the class header lists classes which are being extended by the present class – that is, all their methods become exportable facilities of **C**. An **inherits** clause in **C** lists classes which are being included via *controlled inheritance*: whereby some inherited methods are made internal to **C**.

Dynamic behaviour of objects of **C** is specified in the **sync** and **thread** clauses, which must not conflict. In the **sync** clause, which describes the behaviour of *passive* objects, either an explicit history of an object can be given, as a *trace* expression involving regular expressions in terms of method names, or as a set of *permission* statements of the form:

per Method $\Rightarrow$ Cond

restricting the conditions under which methods can initiate execution.

Threads describe the behaviour of active objects, and can involve general statements, including a **select** statement construct allowing execution paths to be chosen on the basis of which messages are received first by the object, similar to the **select** of Ada or **ALT** of OCCAM. A typical thread using a select statement is:

```
thread
  while true do
    sel
      x ≥ 10  answer m1  →  a!m(x - 10),
      x ≤ 10  answer m2  →  a!m(x),
      x = 10   →  self!m3()
```

An object whose behaviour is specified by this thread waits for the first request of **m1** or **m2** to be recieved, unless **x** = 10, when it may directly proceed to execute the **m3** method on itself, provided there are no outstanding requests for **m1** or **m2**. If **x** > 10 then it must wait for a request for **m1**, and will then execute **a!m(x - 10)** on the supplier object **a**. If **x** < 10 then it must wait for a request for **m2**, and will then execute **a!m(x)** on the supplier object **a**. Having completed the execution of the statement associated with the select clause, the thread of control will return to the loop and this behaviour will be repeated until the object is destroyed.

A set of internal consistency requirements are associated with a class, which assert that its state space is non-empty, and that the definition of each method maintains the invariant of the class, and that the initialisation establishes the invariant. The syntax of traces and threads is given in [8].

The **aux reasoning** section can contain additional logical specifications, for example, of temporal relationships. Examples are given in the following sections.

A simple example of a VDM⁺⁺ class is the following, describing the concept of files which reside on filestores and partitions:

```
class File
instance variables
  owner : @Directory;
  filestore : N;
  partition : N;
init filestore, partition ==
  (filestore := 0; partition := 0)
methods
  set_owner(s : @Directory) ==
    [ ext wr owner
      post owner = s];

  move(xin, yin : N) value error : N ==
    ([ ext wr filestore, partition, error
```

```

        post filestore = xin  ∧  partition = yin ∧  error = 0  ∨
          error = 1];
    return error)
sync
    per move  ⇒  #fin(set_owner) ≥ 1
end File

```

The method definition of `move` asserts that the value of `error` at termination of `move` is either 0, and that in this case the values of `partition` and `filestore` are set to the input values `yin` and `xin` respectively, or it is 1, and the values of `partition` and `filestore` are non-deterministically set to elements of $\mathbb{N}$.

The **sync** constraint indicates that a `move` invocation can only begin execution if at least one `set_owner` invocation has already completed. ($\#fin(m)$ counts the number of terminations of `m` to date.) This is distinct from the concept of a precondition of an operation, which only indicates under what conditions the postcondition of the operation can be assumed at its termination.

3.2 Semantics

There are two levels of semantics for VDM⁺⁺ [13]: the object-oriented extension of VDM-SL can be described in a modal action logic extending the logic of partial functions (LPF) that underlies VDM-SL. The real-time and concurrent aspects require a further extension, to an RTL-based formalism [16] which uses concepts of events and event times to define the meaning of the duration and delay constraints that can be asserted in VDM⁺⁺.

An example axiom, describing the state-transformation effect of a method on the local state of an object, is:

$$\forall i \in \mathbb{N}_1 \cdot (\text{self} \in \overline{C} \wedge \text{Pre}_{m,C}) \odot \uparrow(m, i) \Rightarrow (\neg [\text{Code}_{m,C}] \neg (v_C = v'_C)) [\text{v}_C \otimes \downarrow(m, i) / v'_C] \odot \uparrow(m, i)$$

The $\neg [\] \neg$ construct is used to convert the program statement $\text{Code}_{m,C}$ into a predicate on a set of prestate variables v_C and poststate variables v'_C . $[\text{Code}_{m,C}] \chi$ is the usual weakest precondition of $\text{Code}_{m,C}$ with respect to χ . We write $\text{Post}_{m,C}$ for $\neg [\text{Code}_{m,C}] \neg (v_C = v'_C)$.

The method definition of `m` is of the form:

```

m(x : Xm,C) value y : Ym,C
pre Prem,C ==
    Codem,C;

```

The axiom asserts that the values $v_C \otimes \downarrow(m, i)$ of the object attributes at the time $\downarrow(m, i)$ that the *i*-th invocation of `m` terminates, and the values $v_C \otimes \uparrow(m, i)$ of the object attributes at initiation of the invocation, are related by the postcondition at termination, provided that the object exists at initiation: $\text{self} \in \overline{C}$ and the method precondition holds then.

The theory of a class `C` is then a collection of axioms in a language $\mathcal{L}_C$ in extended RTL. Subtyping between classes is defined in terms of theory extension of these theories: Class `C` is a supertype of class `D` via a retrieve function $R : T_D \rightarrow T_C$ between the respective states, and a renaming ϕ of methods of `C` to those of `D`, if, for every $\varphi \in \mathcal{L}_C$, $\Gamma_C \vdash \varphi$ implies that $\Gamma_D \vdash \phi(\varphi[R(v)/u])$ where v is the tuple of attributes of `D`, u of `C`.

ϕ must map internal methods of `C` to internal methods of `D`, and external methods to external methods. The notation $C \sqsubseteq_{\phi, R} D$ is used to denote this relation.

`D` is a refinement of `C` if it is a subtype of `C` and the retrieve function R satisfies the condition of *adequacy*:

$$\forall u \in T_C \cdot \text{Inv}_C(u) \Rightarrow \exists v \in T_D \cdot \text{Inv}_D(v) \wedge R(v) = u$$

That is, R is onto. In addition, no new external methods can be introduced in `D`, so that ϕ is onto when restricted to $\text{extmethods}(C)$.

Problems arise from the interaction of subtyping and frame rules such as the *locality principle*: that the state of an object can only change during the execution of an invocation of a method of its class. Alternatives to this principle are discussed in [20].

4 Formalisation of Analysis Models

The OMT notation of [27] will be used here, with the dynamic models being extended to the full timed statechart language of [11].

4.1 Formalisation of Object Models

The static data model of OMT, called the Object Model, specifies classes together with their attributes and operation signatures, and associations or relationships between classes. Inheritance and aggregation can be specified in addition.

The formalisation process for the object model is largely a mechanical process. In order to generate the Model_0 specification a single VDM⁺⁺ class for the complete data model is produced, in which the entities E of the object model are expressed using sets $\mathbf{EID}$ of tokens for object identities, and attributes and associations are expressed as maps based on these sets. If we know that only a fixed set of objects are needed, then we can represent their data in a static manner as a set of variables, without the need to introduce a type of identities for these objects (the traffic-light system of Section 5 is an example of this case).

Production of the Model_1 specification consists of defining VDM⁺⁺ classes C for each entity C in the object model, with attributes as specified in the object model (with VDM⁺⁺ types being used). Associations from C become attributes in C of type @D , $\mathbb{F}(\text{@D})$ or $\text{seq}(\text{@D})$, depending on the cardinality of the association with respect to the entity D at the other end. It may not be necessary to formalise both directions of every association: the *Venus* toolset provides configuration facilities which let the user identify which parts of an association or object model they wish to be formalised for a particular application.

Inheritance is represented by **is subclass of** clauses. Aggregation with a fixed multiplicity is represented by indexed inheritance in VDM⁺⁺.

4.2 Formalisation of Dynamic Models

Again, two levels of translation are used. For the Model_0 specification the (statechart) state of an object is simply represented as an additional attribute of an object, of an enumerated type.

The interpretation of transitions uses the **whenever** construct of VDM⁺⁺:

whenever χ
also from $\delta \implies \varphi$

This asserts that the predicate φ must be true at some time point in each interval of the form $[t, t + \delta]$, where t is a time where the predicate χ becomes true (ie., changes from being false or undefined to true).

Thus in case that we have a statechart state $S1$ as the source of an automatic transition τ with guard condition E and time specification $[l, u]$, the appropriate **whenever** statement is:

whenever $\text{state} = S1 \wedge \text{residence}(S1) \leq u \wedge$
 $\text{residence}(S1) \geq l \wedge E$
also from $\delta_\tau \implies \text{state} = S2$

where $S2$ is the target state of τ , and δ_τ is the duration of τ (negligible but non-zero). $\text{residence}(S)$ denotes the amount of time that $\text{state} = S$ has held from the most recent time that this became true. It is zero if the state is not currently S .

For a transition triggered by an external event t , we have:

whenever $\text{state} = S1 \wedge \text{residence}(S1) \leq u \wedge$
 $\text{residence}(S1) \geq l \wedge E \wedge \# \text{waiting}(t) > 0$
also from $\delta_t \implies \text{state} = S3$

The actions which take place as a result of the transitions can be defined separately. If there are several transitions away from the same state, then care should be taken to ensure that only one **whenever** clause for this state can have a true premise at any time: ie, transitions should be prioritised.

For the translation to Model_1 assume that statecharts are given by pairs (S, T) of sets S of states and T of transitions. Then the translation steps for the Model_1 level are:

1. For an entity C with statechart (S, T) the VDM⁺⁺ class C is extended with methods for each transition t in T , parameterised with formal parameters taken from those of t , with types added (input from the user may be needed to identify these types);
2. Create an enumerated type SE with elements for each state of S , and a variable $svar : SE$ representing the current state of an object from the class C ;
3. Each method m representing a transition with source s_1 and destination s_2 has the outline specification

$$m(p : PType) == \begin{array}{l} \text{[ext wr svar} \\ \text{post svar = < s_2 >]} \end{array};$$

(p would be used explicitly in a refinement of this specification) and corresponding permission statement

$$\text{per } m \Rightarrow \text{cond}(m, s_1) \wedge svar = < s_1 >;$$

where $\text{cond}(m, s_1)$ is the condition (if any) guarding this transition in the chart. If m has several possible sources and destinations, then the guard is the disjunction

$$(svar = < s_1 > \wedge \text{cond}(m, s_1)) \vee \dots \vee (svar = < s_n > \wedge \text{cond}(m, s_n))$$

of all the possible source states $s_1, \dots, s_n$ and enabling conditions $\text{cond}(m, s_1), \dots, \text{cond}(m, s_n)$. The method postcondition is a conditional statement selecting the appropriate poststate on the basis of the prestate;

4. The initial state ini is represented by the initialisation

$$\text{init svar} == svar := < ini >;$$

5. The chart is represented by a thread specification:

$$\begin{array}{l} \text{while true do} \\ \quad \text{sel } (svar = < s_1 > \wedge \text{cond}(m, s_1)) \text{ answer } m \rightarrow \text{action}(m, s_1), \\ \quad \dots; \end{array}$$

where $\text{action}(m, s_1)$ expresses any remote transition invocations associated with transition m from s_1 , on the appropriate supplier objects: if the chart for C invokes operations on supplier chart instances $g_1 : D_1, \dots, g_m : D_m$, then these suppliers need to be declared as instance variables $g_i : @D_i$. Automatic transitions are invoked (on *self*) in the statement part of a guarded command, and do not appear in the answer statement part of a select clause. Notice that transition actions take place in the *target* state of the transition, as in the interpretation of [3].

The above approach omits consideration of time-dependency in enabling conditions, or time bounds on transitions (or state residence). The general form of a timed transition is given in Figure 2. E represents an additional enabling condition which must hold before t can fire, l is a lower bound on the time for which t can be enabled before it fires, whilst u is an upper time bound (which might be infinity ∞).

We assume the existence of the global time-valued variable now , which gives the current time, and a time-valued variable $lastentered$ local to each class, both initialised to 0.

Constraints $[l, u]$ on the delay of a method m from the time that its enabling condition holds until it fires become expressed as inequalities:

$$(now \leq lastentered + u) \wedge (lastentered + l \leq now)$$

Figure 2: General form of Timed Transition

in the condition of the guarded command controlling the firing of the method. This condition prevents execution of **m** until at least **l** time units from the time **lastentered** that the source state of the transition was entered. Again, there may be several (**source**, **destination**) cases to be accounted for in this specification, and each may have a timing constraint.

The equality **lastentered** = **now** is added to the postcondition of every method specification. Constraints on the duration of activities can be expressed using the value of **now** – **now**.

Other statechart constructs, such as nesting and AND composition, can also be translated into VDM⁺⁺ class constructs. Details are in [15]. A logical interpretation of statecharts, suitable for proof, is given in [16].

5 Example Development: A Traffic Control System

In this section we will examine a small real-time system specification case study, of a traffic control system, to illustrate the above development process on a non-trivial example.

5.1 Requirements Identification

The components of the controlled system are:

- two pairs of traffic lights, one controlling a farm road, and one controlling a main road, where these roads intersect;
- a monitor for the presence or absence of traffic on the farm road;
- the software controller.

The natural language requirements are:

1. at least one set of traffic lights should be red at every time (invariant);
2. if traffic is absent on the farm road then the farm road lights should be red and the main road lights green (a desired property/invariant);
3. the main road lights should remain red for at most **maxmrred** seconds (timing);
4. if there is not already traffic on the farm road, the system should respond to traffic appearing on the farm road by setting the main road lights red and then the farm road lights green (procedure);
5. the farm road lights should remain red for at most **maxfrred** seconds, while traffic is present on the farm road – it may be red for an unbounded time if there is no such traffic (timing).

maxfrred is greater than **maxmrred**.

Since this system is primarily a reactive event-based system, rather than being concerned with data, operations are identified in connection with events on the dynamic model, rather than on the object model. These operations are: to react to the monitor reporting that the traffic on the farm road has **cleared**, and to react to the monitor reporting that traffic on the farm road has **approached**.

The initial Model₀ specification is expressed in terms of the states of the monitor and traffic lights without factorisation into separate objects (cf. Fusion operation schemas or Syntropy essential models). In particular, we do not yet define a controller object to carry out the required reactions or execute a state machine devised to implement the desired behaviour, but only specify the desired reactions by using **whenever** statements:

```

class Model0
types
  MState = < no_traffic > | < traffic_moving > | < traffic_blocked >;
  TLState = < red > | < green > | < amber_red > | < amber_green >
values
  inamber : ℕ;
  maxmrred, maxfrred : ℕ
instance variables
  mstate : MState;
  mrlightstate, frlightstate : TLState;
init objectstate ==
  [ext wr mstate, frlightstate, mrlightstate
   post
     mstate = < no_traffic > ∧
     frlightstate = < red > ∧
     mrlightstate = < green >]
methods
  approach() ==
    [ext wr mstate, mrlightstate, frlightstate
     post mstate = < traffic_moving > ∧
          frlightstate = < green > ∧
          mrlightstate = < red >];

  clear() ==
    [ext wr mstate, mrlightstate, frlightstate
     post mstate = < no_traffic > ∧
          frlightstate = < red > ∧
          mrlightstate = < green >]

aux reasoning
  always frlightstate = < red > ∨ mrlightstate = < red >;

  whenever mstate = < traffic_moving > ∧ #waiting(clear) > 0
  also from 2 * inamber + δclear ==> (mstate = < no_traffic >;

  whenever mstate = < traffic_blocked > ∧ #waiting(clear) > 0
  also from δclear ==> mstate = < no_traffic >;

  whenever mstate = < traffic_moving > ∧
           residence(< traffic_moving >) = maxmrred
  also from 2 * inamber + δ3 ==> mstate = < traffic_blocked >;

  whenever mstate = < no_traffic > ∧ #waiting(approach) > 0
  also from 2 * inamber + δapproach ==> mstate = < traffic_moving >;

  whenever mstate = < traffic_blocked > ∧
           residence(< traffic_blocked >) = maxfrred
  also from 2 * inamber + δ4 ==> mstate = < traffic_moving >

end Model0
    
```

The first formula in the **aux reasoning** section expresses the first requirement that *at all times* one or other traffic light set is red. This is a stronger assertion than simply placing the formula $\text{frlightstate} = \langle \text{red} \rangle \vee \text{mrlightstate} = \langle \text{red} \rangle$ in a class invariant, since class invariants may fail to be true during the execution of a method, ie, in this case, during a transition of the controller. The second constraint is expressed by the first two **whenever** statements and by the definition of **clear**. The δ values should be derived as maximum allowable delays from the requirements.

Requirements three to five are expressed by the corresponding **whenever** statements and by the definition of **approach**.

5.2 Requirements Formalisation

In the initial design, we invent a controller object to link the traffic light and monitor objects in accordance with the required behaviour constraints. The object model of this design is shown in Figure 3. All of the associations are 1-1, that is, there can be no sharing of controllers between different monitors, etc. The directions of the associations which we wish to formalise are named. VDM constants are represented by *class* attributes (prefixed by a \$) in OMT object

Figure 3: Object Model of Traffic Light System

models.

The outline `Model1` classes are simply:

```

class Monitor
time variables
    input traffic_level: N
instance variables
    controller: @Controller
end Monitor
    
```

A *time variable* is an attribute which may vary continuously even if no method of the object owning it is currently executing. They are used to describe interfaces between the software and an external world.

```

class Controller
values
    maxmrrred, maxfred: N
instance variables
    frlights, mrlights: @TrafficLights
end Controller
    
```

The `TrafficLights` class is empty except for a declaration of `inamber: N` as a constant.

The statechart of the traffic lights is shown in Figure 4, and that of the monitor in Figure 5.

The complete `Model1` formalisation of the traffic lights is therefore:

```

class TrafficLights
types
    TLState = < red > | < green > | < amber_red > | < amber_green >
values
    inamber: N
instance variables
    
```

Figure 4: Statechart of Traffic Lights

Figure 5: Statechart of Monitor

```

    tlstate : TLState;
init objectstate ==
    [ext wr tlstate
      post tlstate = < red > ∨ tlstate = < green >]
methods
    initialise_red() ==
      [ext wr tlstate
        post tlstate = < red >];

    initialise_green() ==
      [ext wr tlstate
        post tlstate = < green >];

    go_red() ==
      [ext wr tlstate
        post tlstate = < amber_red >];

    go_green() ==
      [ext wr tlstate
        post tlstate = < amber_green >];

    tau1() ==
      [ext wr tlstate
        post tlstate = < red >];

    tau2() ==
      [ext wr tlstate
        post tlstate = < green >]
sync
    per go_red ⇒
      tlstate = < green >;
    per go_green ⇒
      tlstate = < red >;
    per tau1 ⇒
      tlstate = < amber_red >;
    per tau2 ⇒
      tlstate = < amber_green >
thread
    (sel answer initialise_green,
      answer initialise_red;
      while true do
        sel
          tlstate = < green > answer go_red →
            (delay(inamber);
              self!tau1());
          tlstate = < red > answer go_green →
            (delay(inamber);
              self!tau2()))
end TrafficLights
    
```

The delay statements are an alternative way of expressing time lags between the enabling of a transition (here, `tau1` and `tau2`) and their execution. This could also be specified using a `lastentered` variable and two additional select clauses, as described in Section 4.2 above.

The thread describes a process which allows the object to be either initialised to the `red` state or to the `green` state, and then continually loops, testing each select condition (to the left of the $\rightarrow$ symbol), until a true condition is found.

An **answer m** with a true condition is selected if there are outstanding requests to execute **m**. The corresponding select clause is then executed by executing **m** and then the statement to the right of the $\rightarrow$.

Verification of the internal consistency of this class will include proving that the thread ensures that the methods are always invoked within their permission guards. This is a liveness requirement: since we know that a traffic light object cannot be shared between two clients in the system, any invocation of a method outside its guard condition will result in deadlock.

The monitor is formalised as a client of the controller, since it only sends events to the controller, and does not receive data or events from it. Similarly the traffic lights are formalised as suppliers of the controller. The **traffic_level** variable of **Monitor** indicates the number of cars on a suitable section of the farm road. When this changes from being zero to being non-zero an **approach** signal should be sent to the controller. When it changes in the opposite direction a **clear** signal should be sent:

```

thread
  while true
  do
    sel
      traffic_level > 0  $\wedge$  old_tl = 0  $\rightarrow$ 
        (old_tl := traffic_level;
         controller!approach());
      traffic_level = 0  $\wedge$  old_tl > 0  $\rightarrow$ 
        (old_tl := traffic_level;
         controller!clear())
    
```

where **old_tl** is set to **traffic_level** initially.

The **Model₁** specification of the controller, using a thread to implement the **whenever** constraints, is then:

```

class Controller
types
  CState = < no_traffic > | < traffic_moving > | < traffic_blocked >
instance variables
  frlights, mrlights: @TrafficLights;
  cstate: CState;
  lastentered:  $\mathbb{R}$ ;
init objectstate ==
  (cstate := < no_traffic >; lastentered := 0)
methods
  approach() ==
    [ext wr cstate, lastentered
     post lastentered = now  $\wedge$  cstate = < traffic_moving >];

  clear() ==
    [ext wr cstate, lastentered
     post lastentered = now  $\wedge$  cstate = < no_traffic >];

  traffic_moving_entry() ==
    (mrlights!go_red();
     frlights!go_green());

  traffic_moving_exit() ==
    (frlights!go_red();
     mrlights!go_green());

  tau3() ==
    [ext wr cstate, lastentered
     post lastentered = now  $\wedge$  cstate = < traffic_blocked >];
    
```

```

tau4() ==
    [ext wr cstate, lastentered
    post  lastentered = now    ^    cstate = < traffic_moving >]

sync
    per tau3  =>    cstate = < traffic_moving >;
    per tau4  =>    cstate = < traffic_blocked >
thread
    while true do
        sel
            cstate = < no_traffic >
                answer approach → self!traffic_moving_entry(),

            cstate = < traffic_moving >
                answer clear → self!traffic_moving_exit(),

            cstate = < traffic_blocked > answer clear,

            (cstate = < traffic_moving > ^
             lastentered + maxmred ≤ now) →
                (self!tau3();
                 self!traffic_moving_exit()),

            (cstate = < traffic_blocked > ^
             lastentered + maxfred ≤ now) →
                (self!tau4()
                 self!traffic_moving_entry())
    end
end Controller

```

The abstract state is closely related to this initial design. For instance, `mstate` is implemented by `cstate`, and `mrlightstate` is implemented by `mrlights.tlstate`.

Since `TrafficLights` is a class, objects of the class are declared by using the type `@TrafficLights` of *references* to this class. The **init objectstate** clause defines the initial state of the component – in this case the `< no_traffic >` state.

The statechart expressing the design of the system is shown in Figure 6.

The use of a translation process has helped to establish a good traceability between the requirements and the implementation: that is, it is direct to relate the requirements through the intermediate stages of their realisation to features of the eventual implementation. This therefore improves the maintainability, assessability and the capability for validating the developed system.

6 Industrial Case Studies

A number of case studies have been carried out using the methods integration process described here, and using versions of the *Venus* toolset. These include:

- Ship-loading systems for bulk and chemical carriers;
- A minimum safe altitude warning system for an air traffic control network;
- Data acquisition from particle accelerators;
- Translation into a hardware description language.

The second case study will be described here.

Figure 6: Statechart of Controlled System

6.1 Minimum Safe Altitude Warning (MSAW)

This system is part of a projected revision of the French air-traffic control network. VDM⁺⁺ has been trialed on this component of the system in order to evaluate its possible wider application within the ATC application. The prime function of the MSAW subsystem is to extrapolate the current trajectory of a plane, and to predict potential collisions with known static obstacles (eg, mountains or tall buildings). A warning signal to an ATC operator is given in the case that a high probability of collision is identified.

VDM⁺⁺ was used at several different levels. An initial formal model provided an abstract and concise description of the main entities of the system (formalised as classes `Obstacle`, `Radar`, etc.) and the required functions, formalised as methods of these classes. This model was then elaborated towards a logical design, incorporating implementation details such as the need to delay the giving of an alarm as long as possible (to avoid spurious warnings). A succession of models of increasing refinement was developed. An advantage of the object-oriented structuring facilities was that the specification could be refined by refining each of the abstract classes separately, and by adding in new classes: it was not necessary to eliminate the conceptually clear entities defined in the initial model. Inheritance was also used to specify natural hierarchies such as the range of possible shapes (cones, cylinders, etc) for obstacles.

The top-level processing of the system is defined in the `Workspace` class, parts of the `Model1` specification of this class are shown as follows:

```
class Workspace is subclass of MSAWTypes
functions
  OrderOfTreatment :
     $\mathbb{F}(\text{@Terrain} \mid \text{@Obstacle}) \rightarrow \text{seq}(\text{@Terrain} \mid \text{@Obstacle})$ 
  OrderOfTreatment(obstacles) ==
    let s : seq( @Terrain | @Obstacle )
    be st
      elems(s) = obstacles  $\wedge$ 
      len(s) = card(obstacles)
    in s

    /* This function defines an ordering on the set of obstacles */

instance variables
  obstacles :  $\mathbb{F}$  @Obstacle;
  terrain : @Terrain;
  ...
  ground_radar : @Radar;
init obstacles == obstacles := {}

methods

Init( ) ==
  ( ground_radar := Radar!new;
    terrain := Terrain!new;
    ...
    topology [ post_detector.ground_radar = ground_radar ]
  );

SetObstacle(obstacle : @Obstacle ) ==
  obstacles := obstacles  $\cup$  {obstacle};

CalculateCollisions( ) ==
  def modeAs = ground_radar!GetModeAs() in
  let obs = OrderOfTreatment({terrain}  $\cup$  obstacles) in
  for all modeA  $\in$  modeAs do
    def prediction_segment = PredictionSegment!new;
```

```

        positions = ground_radar!GetPositions(modeA) in
        if card dom(positions) ≥ 2 then
        ( prediction_segment!Set(modeA, positions,
                                bulletin_meteo);
          for all i ∈ inds obs do
            detector!CalculeRisque(predication_segment, obs(i))
          )
        )
    /* This method is the main operation of the system – it calculates
       the risk of intersection of the prediction segment of each plane
       with each obstacle. The following thread asserts that it is performed
       repeatedly with period RadarInterval. */

thread
    periodic(RadarInterval)(CalculateCollisions)
end WorkSpace

```

A problem related to this case study which can be addressed using VDM⁺⁺ is the need to avoid multiple hierarchies of classes. This arises from the use of particular shapes to represent the layout of different forms of obstacle. An obstacle can be of one of several classes (eg, a military zone, a building or tower, an airport zone, etc), and each kind of obstacle may be further subdivided into types depending on its shape (cylinder, cone, inverted cone, etc). Rather than creating a hierarchy formed out of all combinations of obstacle type and shape (thus leading to classes `conicalbuilding`, `cylindricalbuilding`, etc), the shape(s) of which an obstacle are constructed can be defined as attributes of the obstacle, and operations dependent on both the type of obstacle and the shape performed as methods of the obstacle class, using delegation. For instance a method `intersects` could be defined as follows in the class `MilitaryObstacle`:

```

class MilitaryObstacle
    is subclass of Obstacle
instance variables
    active : ℬ;
    ...
methods
    intersects(seg : @PredictionSegment) value ℬ ==
        if active
        then
            (dcl bb : ℬ := false;
             bb := primary_shape!intersects(seg);
             return bb)
        else
            return false
        end
    end

```

Where

```
primary_shape : @Shape
```

is defined as an attribute of `Obstacle`, and the `intersects` method is inherited and redefined from `Obstacle`.

In classes representing other types of obstacle a different algorithm could be used to calculate the `intersects` method, utilising the polymorphic `intersects` method of `Shape`. This permits dynamic binding to be used on both the `Obstacle` and `Shape` hierarchies, without introducing a combinatorial explosion of classes. This is an example of a structural design pattern [9].

The case study was perceived as being a success, in that a description of the system at both an abstract level, comprehensible to the domain experts, and at implementation level, close to executable code, was produced. The *Venus* toolset was used extensively to keep the formal specifications and diagrammatic models mutually consistent.

The main problem concerned the absence of tool support for the real-time extensions of the language, which are now in place. Training took approximately 10% of the development time, however this should be reduced for future applications using the language. The language is now being further considered for use within larger elements of the ATC application.

Some statistics from the development are as follows:

- Analysis model size: 934 lines of *LOV* class description files;
- Specification size: 18 classes comprising 530 lines of VDM⁺⁺ text files;
- Prototype implementation size: 1776 lines of “.cc” files, and 639 lines of “.h” files.

The growth of the system from specification the prototype is directly due to the use of library classes by the code generator. A similar growth occurs for the implemented code, although in this case additional complexity is introduced during the VDM⁺⁺ refinement and design process.

7 Comparison with other approaches

Methods integration approaches involving formal and diagrammatic methods have ranged from a highly focused use of formal techniques to verify a security critical or safety critical kernel of a system (eg., [5]), to extensive combination at several life cycle stages [12, 26]. In addition the translation between diagrammatic and formal notations ranges from being precisely defined and automatable to being intuitive and dependent on user understanding. *Venus* does not mandate either a localised or broad use of formal notations throughout a development: it is possible to use the formal notation and tool support for a small part of a system, which is then integrated via the *LOV* tool into the remainder of a conventional OO development. Alternatively the formal notations can be used to catch semantic errors that are not detectable via the OMT notation (such as a claimed subtyping relationship not being mathematically valid), as part of a rigorous review process. *Venus* provides an automated but configurable translation from object models to formal specifications, thus combining the benefits of precise and user-controllable translation processes. The translation process for statecharts is at present manual, although it is systematically defined and automation is envisaged in future versions of the toolset. No automated translation for functional models or Booch interaction diagrams is envisaged, as these do not have a precise semantics.

The benefits of the high degree of integration provided by *Venus* include being able to switch between formal or diagrammatic views of a system as required, in order to gain greater understanding of a system description, or to conduct specific forms of analysis. Some forms of transformation are more easily performed using the diagrammatic notations (for example, restructuring a complex data model by using a diagram editor) than by using formal textual equivalents, and a close integration allows appropriate tools to be used at each stage.

In contrast to Syntropy and Fusion, the IFDM method of VDM⁺⁺ does not add formal notations directly to diagrams, but instead combines diagrams and formal text via translation steps. Placing formal text directly on diagrams can lead to excessive complexity in these diagrams, so hindering their usefulness in giving an abstract presentation of requirements.

Tool support exists for the rigorous review technique at BT [28] and for the Fusion method [2]. *Venus* provides comparable support for diagrammatic methods to these tools, and more extensive support for formal notations. The main omission from *Venus* at present are tools for proof, although the axiomatic semantics needed to support such tools has been established [15].

Conclusions

The VDM⁺⁺ language and the combination of VDM⁺⁺ with OMT via the *Venus* toolset provides a pragmatic integration of formal and object-oriented techniques. Experience in industrial trials of the toolkit has suggested that it may be well-suited for the introduction of formality into existing object-oriented development.

The language and method permits existing expertise and techniques for object-oriented software development to be reused (for example, the use of delegation in the MSAW case study to avoid combining multiple hierarchies of

classes), whilst providing a more rigorous basis for verifying development steps and the internal consistency of classes. Specialisations of the IFDM presented here have been given for process control systems [19] and distributed systems [18].

Training courses have been given for the language and tools to industrial clients, with a high degree of success resulting directly from the use of a CASE tool “front end” for formal specifications. This meant that excessive emphasis on formal text and mathematical notation could be avoided at the beginning of the course, with general abstraction and modelling principles being emphasised instead.

Acknowledgement

The authors acknowledge the financial support of the CEC funded Esprit project AFRODITE, involving the following companies and organisations: Cap Volmac, CERN, DRA, Imperial College, Helintec S.A., IFAD Denmark, Utrecht University and Verilog. The air-traffic control case study was carried out by STNA-7 and CAP.

References

- [1] Booch G. **Object-Oriented Analysis and Design with Applications**, Benjamin/Cummings, 1994.
- [2] Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F., and Jeremaes P. *Object-oriented Development: The FU-SION Method*. Prentice Hall Object-oriented Series, 1994.
- [3] Cook S. and Daniels J. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice Hall, Sept 1994.
- [4] Cusack E. and Wezeman C. Deriving tests for objects specified in Z. In J Nicholls, editor, *Z User Meeting 1992*, Workshops in Computing. Springer-Verlag, 1993.
- [5] Draper C. Practical Experiences of Z and SSADM. In J Bowen, editor, *Proceedings of 1992 Z User Meeting*, Workshops in Computing. Springer-Verlag, 1993.
- [6] Duke R., King P., Smith G. Formalising Behavioural Compatibility for Reactive Object-oriented Systems, *Proc 14th Australian Compt. Sci. Conf. (ACSC-14)*, 1991.
- [7] Durr E. and Dusink E. The role of VDM⁺⁺ in the development of a real-time tracking and tracing system. In *Proceedings of FME '93*, eds. J. Woodcock and P. Larsen, LNCS, 1993, pp. 64–72.
- [8] Durr E., Duursma A., Plat N. (Eds). VDM⁺⁺ Language Reference Manual, AFRODITE project document AFRO/CG/ED/LRM/V9.1, CAP Gemini Innovation, May 1994.
- [9] E Gamma, R Helm, R Johnson and J Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1994.
- [10] Goldsack S., Lano K. *Theory of Annealing*, AFRODITE project technical note, Imperial College, 1994.
- [11] Harel D. *On Visual Formalisms*, Communications of the ACM 31, 5, pp 514–530, May 1988.
- [12] Hill J. V. Software development methods in practice. In *Proceedings 6th Annual Conference on Computer Assurance (COM-PASS)*, 1991.
- [13] Kent S., Lano K. *Axiomatic Semantics for VDM⁺⁺*, AFRODITE project document AFRO/IC/SKKL/SEM/V1, August 1995.
- [14] Lano K., Haughton H. *Object-oriented Specification Case Studies*, Prentice Hall Object-oriented Series, 1993.
- [15] Lano K. *Formal Object-oriented Development*, FACIT series, Springer-Verlag, 1995.
- [16] Lano K. *Reactive System Specification and Refinement*, Proceedings of TAPSOFT '95, Springer-Verlag LNCS 915, 1995.
- [17] Lano K. Applications of Formal Methods to Safety-Critical Transport Systems: the AFRODITE Project, *Safety Critical Systems Club Newsletter*, Vol. 4, No. 2, January 1995.
- [18] Lano K. *Distributed System Specification in VDM⁺⁺*, FORTE '95 Proceedings, Chapman and Hall, 1995.
- [19] Lano K., Goldsack S. *Discrete Event Process Controller Synthesis Using VDM⁺⁺*, First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '95), IEEE Press, 1995.
- [20] Lano K., Fiadeiro J., Goldsack S. *Categorical Semantics for Object-oriented Specification*, BCS FACS Xmas workshop, 1995.

- [21] Liskov B. *Data Abstraction and Hierarchy*, OOPSLA '87 Conference Proceedings, 1987.
- [22] McHale C., Baker S., Walsh B. and Donnelly A. Synchronisation Variables, Amadeus Project report TCD-CS-94-01, University of Dublin, 1994.
- [23] Meira S. and Cavalcanti A. Modular object-oriented Z specifications. In *Z User Meeting 1990*, Workshops in Computing, pages 173–192. Springer-Verlag, 1991.
- [24] Meyer B. **Object-oriented Software Construction**, Prentice Hall, 1988.
- [25] Ostroff J. **Temporal Logic for Real-time Systems**, Wiley, 1989.
- [26] Polack F. and Whiston M. Formal methods and system analysis. In *Methods Integration Conference*, Workshops in Computing, Springer-Verlag, 1991.
- [27] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W. **Object-Oriented Modelling and Design**. Prentice-Hall International, 1991.
- [28] Semmens L., France R., and Docker T.: Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6), 1992.
- [29] Voss, H. User Manual for the VDM⁺⁺ to C++ Code Generator, AFRODITE project document AFRO/IFAD/HV/CGMAN/V3, IFAD, Odense, Denmark, 1995.