# 1

# Distributed System Specification in VDM$^{++}$

*Kevin Lano*
*Dept. of Computing, Imperial College*
*180 Queens Gate, London SW7 2BZ. email:* kcl@doc.ic.ac.uk

**Abstract**

This paper describes an application of VDM$^{++}$ to the specification and design of a simple communication system, based on requirements for an advanced network service specified by the author.

It is shown how VDM$^{++}$ may be combined with diagrammatic methods (OMT and Fusion) in order to enhance the precision of the latter. We also discuss the issue of feature interaction, and the use of object-oriented specification to address this problem.

**Keywords**

Object-oriented specification, communications systems, feature interaction, methods integration

## 1 INTRODUCTION

VDM$^{++}$ (Dürr et al, 1995) was developed as a concurrent object-oriented extension of the widely used VDM (Vienna Development Method) language (Jones, 1990). 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 (Lano, 1995).

A toolset, *Venus*, exists for VDM$^{++}$, and includes syntax and type-checking facilities, and code generation in C++. The toolset is integrated with a CASE tool for OMT (Rumbaugh et al, 1991), to allow translations of OMT into VDM$^{++}$, and graphical presentations of VDM$^{++}$ specifications. Translation tools targeted at Ada95 and the hardware description languages ELLA and VHDL are also being developed (Moore, 1995).

VDM$^{++}$ is suitable as a language for the specification of distributed systems because of its combination of object-orientation and concurrency: objects may represent modules or 'partitions' which are executable on distinct processors. In VDM$^{++}$ it is additionally possible to specify internal concurrency within an object, whereby distinct methods of the object may be simultaneously executing, on different physical processors.

Object-oriented specification languages have already been used in the communications field: the ZEST language, an object-oriented extension of Z, has been used within British Telecom (BT) to specify network services, where it was considered to have a number of advantages over more traditional languages such

G. V. Bochmann et al. (eds.), *Formal Description Techniques VIII*

as SDL or LOTOS (Cusack et al, 1992). However, ZEST and other Z-based object-oriented specification languages do not have direct support for concurrency or real-time, unlike VDM$^{++}$, nor are industrially usable tools available. In addition, design and refinement techniques are available for VDM$^{++}$.

Fusion (Coleman et al, 1994) provides semi-formal notations (structured English) for operation pre-conditions, post-conditions and invariants, and uses variants of the OMT object-model notation and Booch object interaction diagram notation, together with other notations, to provide a rigorous method for sequential systems. It is now used extensively within Hewlett Packard, across 15–20 divisions and in areas such as printer technology, network management software and test software. Other companies have taken up Fusion in the USA and Europe.

## 1.1 Method

The following steps are taken as part of a process of formal object-oriented development using OMT or Fusion notations and VDM$^{++}$.

- *Requirements Analysis*: identify the top-level operations of the system (user services) and scenarios of use of these operations, using Fusion operation models. Identify the data of the system, using OMT object models. Identify the dynamic behaviour of the system, using statecharts (Harel, 1987) or (in Fusion) lifecycle expressions.
- *Requirements Formalisation*: each entity type in the object model of the system becomes a VDM$^{++}$ class, with OMT attributes and associations becoming VDM$^{++}$ instance variables. This translation is automatically performed by the Venus toolset. Statecharts or lifecycles can be systematically formalised as method definitions, permission guards and thread definitions in the VDM$^{++}$ classes. Object interaction graphs are constructed to identify the external services of each class and the pattern of communication between objects (i.e., which parts of the overall services are to be the responsibility of particular objects). They provide guidance on the structure of method definitions.
- *Refinement and Design*: a series of (possibly fully formal and proved) refinement steps are carried out from the initial VDM$^{++}$ classes into classes which are immediately translatable into C++ or Ada95.
- *Implementation*: translation into the target programming language. At present a C++ translator is available (Voss, 1995), and an Ada95 translator is under development.

## 1.2 Syntax

A VDM$^{++}$ specification consists of a collection of *class definitions*, of the form:

```
class C
types
  T = TDef
values
  const : T = val
functions
  f : A → B
  f(a) == Defn_f(a)
time variables
       u_C : P_C
```

```
  input wC : SC
instance variables
  vC : TC;
inv objectstate ==  InvC;
init objectstate ==  InitC
methods
  m(x : X)  value y : Y
    pre Pre_m,C(x, vC) ==
        Defn_m,C;
  ...
sync ...
thread ...
aux reasoning ...
end C
```

The static data aspects of a system component are described in the **types**, **values**, **functions**, **instance variables**, **inv** and **init** clauses. These components are similar to corresponding state definition components in VDM, except that class names **D** may be used as types in the form **@D** "the type of references to objects of **D**" in declarations.

The **time variables** clause lists quantities which are continuously varying inputs or outputs of the system (for example, a component describing a temperature monitor would have an input time variable corresponding to the measured temperature). An **output** modality is the default for variables in this clause. The effects of transitions and state activities are described in the **methods** clause.

Methods can be defined in an abstract declarative way, using *specification statements*, or by using a hybrid of specification statements, method calls and procedural constructs such as sequential composition and conditional statements.

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 writable variables
     rd read-only variables
 pre precondition
 post postcondition]
```

In the **postcondition** the value of an attribute **att** at initiation of execution of the specification statement is denoted by $\overleftarrow{\mathbf{att}}$. **att** itself denotes the value at termination of execution, i.e., the value which is established by the statement.

The dynamic behaviour of instances of the class **C** is specified in the **sync**, **thread** and **aux reasoning** sections.

The **sync** and **thread** clauses 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 operations on sequences of method names, in a more restricted version of the regular expression language of Fusion life histories, or a set of *permission* statements of the form:

```
per Method ⇒ Cond
```

are given. This statement prevents **Method** initiating execution unless **Cond** holds.

Threads describe the behaviour of active objects, and can involve general statements, including a `sel` 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.

In the following sections we will show how VDM$^{++}$ may be used to specify a simple communications system. Only extracts will be given, for reasons of space.

# 2 REQUIREMENTS

The system to be specified supports 'intelligent' management of communication between phones and other nodes in a telecommunications network, which can be used to transmit conversations and other forms of data, such as computer files, provided that the source and target nodes of a communication can deal with this data.

Agents (people, generally) can request communications with other nodes in the network, receive calls from other nodes, and can disconnect, hold, suspend or switch calls, and can access information on the history of calls that they have made or received. They can set their own nodes to accept, reject or send messages back to the caller in response to incoming calls. Agents can assume various roles (i.e., their job role versus their personal role) and communication can be directed at or from an agent with respect to these roles.

# 3 ANALYSIS

From the above description we can identify the following entities:

- Agent – representing a person. Each agent will be assumed to be associated with several nodes at any time, and they can change nodes. Subtypes of this type will represent the various roles that an agent can play (so, for instance, the role that a person plays as an employee will be counted as a distinct agent from the one that corresponds to the role they play as a private individual). An agent has operations of **request_call**, **terminate_call**, and so forth, all of which are delegated via its local nodes as its interface to the network.
- Node – representing a node in the communication network. This services at most one agent, and provides operations to initiate and respond to calls, and attributes which store the current activities of the node and past activities. Subtypes of **Node** could include Telephone, Fax or PC, but are not detailed here.
- Network – representing the set of nodes and their interconnections which form the network. This data can be dynamically modified (new nodes and connections can be added), and this entity determines which nodes are able to communicate with others. A network could also associate actual nodes (object identifiers) to names for nodes (such as telephone numbers), but this is a level of detail that we will avoid at the abstract specification level.
- Call – this represents all information about a specific communication: what its source and target nodes are, what data is being transmitted, its status (active, awaiting connection, suspended, held, etc.), and its duration, starting and termination times, and so forth.

- Data – representing the data that can be transmitted in a call. Its subtypes include Agent (a person can be data when in interactive communication) and File (computer files).
- Activity – representing the forms of activity which an agent (via a node) may be engaged in presently or in the past. Types of activity include receiving a call and sending a call.
- Option – a list of options which an agent may select from. Such objects are user-definable menus whose choices correspond to methods that delegate suitable actions to the local or target nodes when selected (e.g.: to leave a message for the agent at the target node if they are busy or absent).

A partial OMT object model of the system is shown in Figure 1. An unfilled triangle denotes inheritance in which the listed subtypes are pairwise disjoint (this is the OMT, rather than Fusion, convention).

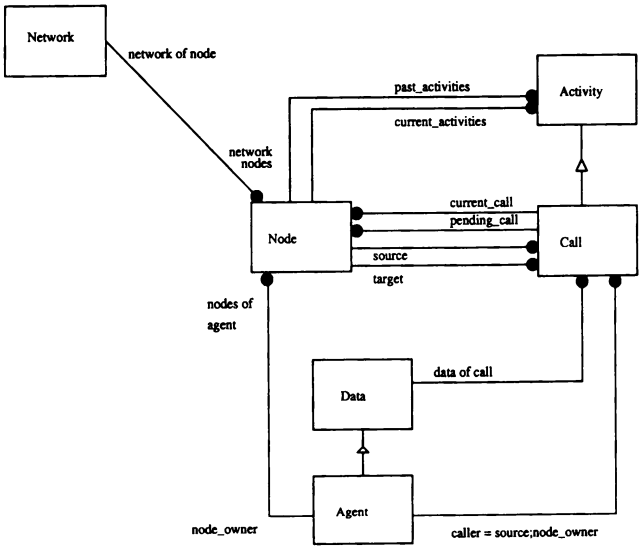



**Figure 1** Object model of communication system

Some example system operations or scenarios of use are as follows:

- *An agent requests a connection to a specific target node.* This has the preconditions that a path exists from the source to the target node, and that the agent is authorised to make this connection. Its effects are to create a new **Call** instance, to set its source and target appropriately, to set its data, and to modify the target node to set its pending call to be the current call.
- *A node receives a request to establish communication.* The response of the node depends on its status. If it is 'busy' or 'absent' then a suitable notification of unavailability, together with a list of options, should be sent back to the caller. Otherwise the communication is opened and confirmed to the caller (a more sophisticated approach could ask the recipient agent if they wanted to accept the call first).

Once the communication has been established the state of the node should be changed to 'busy' and an appropriate **Call** activity added to the list of current activities.

An abstract formal specification of the first operation can be given in a **Workspace** class which lists all the operations which an external user expects to achieve using the system:

```
class Workspace
instance variables
  network : ©Network;
  nodes : 𝔽(©Node);
  calls : 𝔽(©Call)
methods
 request_call(agent : ©Agent, node : ©Node, target : ©Node)
     pre network.authorised_to_connect(node, target)  ∧
         node ∈ agent.local_nodes ∧
         target ∈ nodes ∧
         target ≠ node   ==
     [ext wr calls, target
      post ∃ cl : ©Call · cl ∉ ←calls ∧
                  calls = ←calls ∪ { cl }  ∧
                  cl.source = node ∧
                  cl.target = target ∧
                  cl.data = agent ∧
                  cl.caller = agent ∧
                  target.pending_call = cl ]
end Workspace
```

The specification of **request_call** uses the same general structure as a Fusion operation specification, describing the pre-states in which the operation should be executed, the desired post-states, and the objects which are created by the operation, but uses precise mathematical notation rather than structured English. This is a key advantage if proofs about the effect of the operation are required. At this initial abstraction level (Model 0) class types can be used just as if they were record types in VDM, with direct access to their attributes and the ability to use non-side-effecting methods as expressions (e.g. **authorised_to_connect** in the above definition). The call type here has been assumed to be interactive, so that the agent 'sends itself as data'.

$\mathbb{F}(\mathbf{T})$ is the type of finite sets of elements of **T**.

# 4 SPECIFICATION

The highly abstract analysis model and its operations can be refined via a process of 'annealing' (Dürr et al, 1994), in which new classes are defined and the effects specified in the abstract operations are achieved by invoking methods on supplier objects. In the following specification we will not detail how arguments of operations (e.g., dialled numbers) are assembled from input streams of low-level signals, or how the actual user interface (e.g., buttons, fax sensors or phone hook) corresponds to the conceptual interface

given by the **Workspace**. Feature interaction is not an explicit aspect of this system, however this issue should be addressed in order that the specification can be reused for any future extensions of the system. We identify ways in which this can be achieved in Section 6.

## 4.1 Call class

This class has non-trivial dynamic behaviour, described by the OMT statechart in Figure 2.

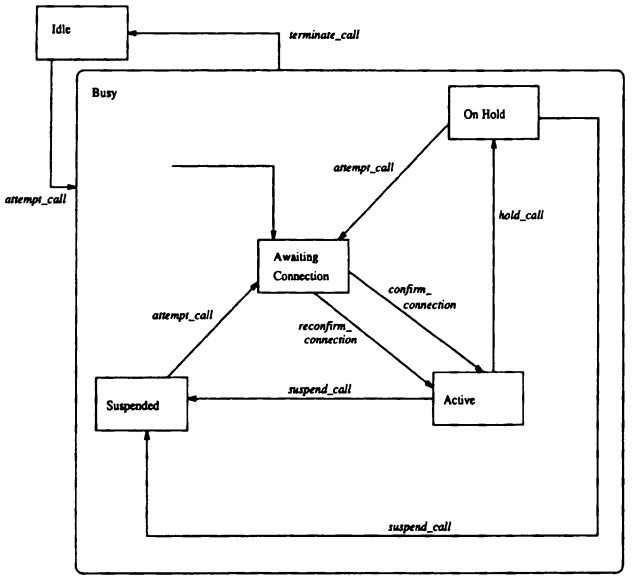



**Figure 2** Statechart of Call entity

In outline, its definition is therefore as follows:

```
class Call
  is subclass of GlobalTypes, Activity
types
  CState = < idle > | < awaiting_connection > | < active > |
           < suspended > | < on_hold >
    /* Cstate is an enumerated type representing the
       current state of the call. */
instance variables
  cstate : CState;
  call_type : CType;
```

```
  target, source : @Node;
  data : @Data;
  caller : @Agent;
  start_time, termination_time : TIME;
    /* TIME is the type representing time – it is usually
       defined as the non-negative real numbers. */
inv objectstate ==
  source ≠ target;
init objectstate ==
  cstate = < idle >
methods
  attempt_call() ==
       cstate := < awaiting_connection >;

  confirm_connection() ==
       [ext wr cstate, start_time
        post cstate = < active > ∧ start_time = now];

  reconfirm_connection() ==
       cstate := < active >;

  suspend_call() ==
       cstate := < suspended >;

  hold_call() ==
       cstate := < on_hold >;

  terminate_call() ==
       [ext wr cstate, termination_time
        post cstate = < idle > ∧ termination_time = now]
sync
  per attempt_call ⇒ cstate ≠ < active > ∧ ¬ void(target);
  per suspend_call ⇒ cstate = < active > ∨ cstate = < on_hold >;

  per hold_call ⇒ cstate = < active >;
  per terminate_call ⇒ cstate ≠ < idle >;
  per confirm_connection ⇒ cstate = < awaiting_connection >;
  per reconfirm_connection ⇒ cstate = < awaiting_connection >
thread
  while true do
    sel
      answer attempt_call ->
               target!answer_call(self),
      answer confirm_connection ->
               source!confirm_connection(),
      answer reconfirm_connection ->
               source!confirm_connection(),
```

```
        answer suspend_call ->
                target!be_suspended(),
        answer hold_call,
        answer terminate_call ->
                target!release_connection(self) ||
                source!release_connection(self)
end Call
```

The expression **now** refers to the current time within a VDM$^{++}$ specification. In a specification statement postcondition it refers to the time that this statement terminates.

The inheritance and attributes clauses of this class are derived from the description of the Call entity in the object model. The states of the class, and its operations, are derived from the statechart of the Call entity. The thread definition is derived from Fusion object interaction graphs.

The **void(target)** predicate is true exactly when **target** does not refer to an existing object of the **Node** class. Accesses to an object state, and invocations of methods on an object are only valid if the object reference concerned is non-void.

The || operator is an extension to VDM$^{++}$ as defined in (Dürr et al, 1995). It is defined as the concurrent execution of two state transitions where these transitions update disjoint data. It enables abstraction from the order of such transformations, and corresponds to the $\wedge$ operator in VDM and Z, and to || in B (Abrial, 1995).

The thread tests successively if there are any outstanding **attempt_call** requests, then if there are any outstanding **confirm_connection** requests, and so forth. The first operation with an outstanding request is then answered, and the method body executed. Then the statement following the answer statement for this select clause is executed and the select loop is restarted at the beginning. A thread may include an arbitrary statement, although the usual style of thread specification is an unbounded loop over a select statement, as above.

**Call** as specified above could be expressed as a formal subtype of a general call concept which allows a set **call_participants** of nodes to be involved in the communication. The relevant data refinement is:

**call_participants** = {**source**, **target**}

Desirable properties expected for all forms of call can be expressed in the general call class, whilst subclasses express the additional behaviour resulting from new features (in the above case, a restriction that all calls have at most two participants).

## 4.2 Node class

The most complex class in the system is the **Node** class, which manages both the reception and transmission of calls between agents. Nodes present a certain image of their associated agent to the rest of the world. In particular a node may have status **busy** which need not reflect the true status of the agent. For the purposes of communication however, it is the state of the nodes attached to an agent which is significant. It may be preferable to create separate types of a **Recieving_node** and **Transmitting_node** and use multiple inheritance to define types of nodes which can exhibit both sets of behaviours.

```
class Node
  is subclass of GlobalTypes
```

```
types
  NState = < busy > | < absent > | < other >
instance variables
  previous_activities : 𝔽(@Activity);
  current_activities : 𝔽(@Activity);
  part_of_network : @Network;
  nstate : NState;
  options : NState ⟶ᵐ @Option;
    /* options is declared to be of a "map" type, and therefore
       associates exactly one Option to each possible state. */
  agent : @Agent;
  pending_call, current_call : @Call;
init objectstate ==
  previous_activities = { } ∧
  current_activities = { } ∧
  nstate = < other >
methods
    /* Methods to set options for a status, to assign an agent
       and network, etc. */

  answer_call(call : @Call) ==
          pending_call := call;

  confirm_connection() ==
          (current_activities := current_activities ∪ { current_call } ||
           nstate := < busy >);

  initiate_call(target : @Node, caller : @Agent,
                call_type : CType, data : @Data)
      pre self ≠ target ==
          (dcl call : @Call := Call!new;
             current_call := call;
             topology [post call.source = self ∧
                            call.target = target ∧
                            call.data = data ∧
                            call.call_type = call_type ∧
                            call.caller = caller]);

    /* Plus methods to hold, suspend,
       reconnect, switch, terminate and release calls */

sync
  per initiate_call ⇒ nstate = < other >;
  ...
thread
  while true
  do
```

```
    sel
      nstate = < busy > ∧ pending_call = current_call
           answer answer_call ->
                    pending_call!reconfirm_connection(),

      nstate = < other > answer answer_call ->
            (pending_call!confirm_connection() ||
             nstate := < busy > ||
             current_call := pending_call ||
             current_activities := current_activities ∪ { pending_call }),

      nstate ≠ < other > answer answer_call ->
            (pending_call.caller)!notify_unavailable(options(nstate)),

      nstate = < other > answer initiate_call ->
                    current_call!attempt_call(),
      ...

end Node
```

The definition of **initiate_call** is still quite abstract, as it involves direct updates to attributes of a supplier object (**call**, via a `topology` statement) which is not allowed in a design or implementation-level class in VDM$^{++}$. Instead, these updates must be achieved by a suitable initialisation operation of the **Call** class.

An important invariant of **Node** is that if its state is **busy** then the **current_call** is not void, and its status is not **idle** or **suspended**. Such an invariant cannot be written in the `inv` clause because it refers to supplier attributes. Instead it is placed in the `aux reasoning` clause.

The **Network** class manages the set of existing nodes, and allowed connections between them. The abstract query **authorised_to_connect** is now implemented by maintaining a list of allowed connections as a relation.

The **Agent** class defines the operations which an agent may perform in the system.

A Fusion interaction graph for the 'request call' system operation is shown in Figure 3. This shows how the original abstract operation has been localised as an operation of **Agent**, and decomposed into a series of calls on individual objects. There is in addition a check on the validity of the requested connection, using the **Network** class. This is performed at the **Workspace** level.

This operation can be seen to refine the initial highly abstract specification in **Workspace** since it has corresponding preconditions, and results in the creation of a new **Call** instance whose attributes are set as specified in the abstract specification. Moreover **current_call!attempt_call()**, if it succeeds, will lead to **target.pending_call** being set to be this new call instance, as required in the initial specification.

Finally, the class **GlobalTypes** is defined by:

```
class GlobalTypes
types
  CType = < interactive > | < urgent > | < normal > | < text >
end GlobalTypes
```

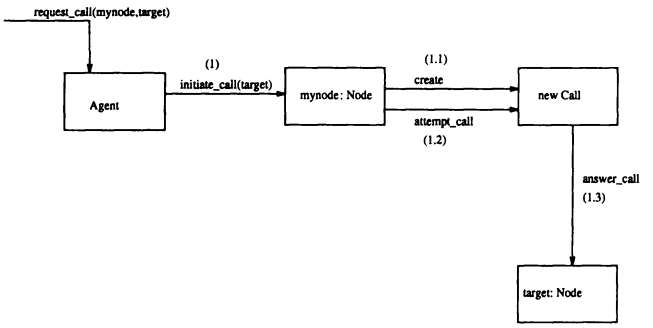



**Figure 3** Interaction graph for 'request call'

Both the **Data** and **Activity** classes are empty.

# 5 REFINEMENT AND IMPLEMENTATION

For the most part, the above classes are already quite close to an implementation in an object-oriented language. The || operator needs to be systematically replaced by suitable sequential compositions, and any direct access to attributes of supplier objects needs to be replaced by method invocations on these objects. The C++ (Stroustrup, 1994) code generator can be applied to such refinements. An example of generated code is the following for a fragment of the **Node** class containing just the **confirm_connection** method:

```
/* Node.cc: */
#include "Node.h"

void vdm_Node::vdm_confirm__connection() {
  Set rhs_3;
  {
    Set var1_4;
    var1_4 = vdm_current__activities;
    Set var2_5;
    {
      Set res_s_6;
      {
        Class tmpVar_7(VDM_Call);
        tmpVar_7 = vdm_current__call;
        res_s_6.Insert(tmpVar_7);
      }
      var2_5 = res_s_6;
```

```
    }
    Set tmpUnionSet_8;
    tmpUnionSet_8.ImpUnion(var1_4);
    tmpUnionSet_8.ImpUnion(var2_5);
    rhs_3 = tmpUnionSet_8;
  }
  vdm_current__activities = rhs_3;
  {
    Quote rhs_9;
    rhs_9 = (Quote) "busy";
    vdm_nstate = rhs_9;
  }
}
```

Extensive use of library classes is made in the generated code. These library classes provide implementations of sets, sequences, maps and other abstract data types. In the above code for example, the generic **Set** class is used to provide a type for the **current_activities** variable.

Code generation can also be used at an early development stage in order to provide animation facilities, although for efficient production code it is necessary to carry out several refinement stages and to produce classes which are within a subset of the language (avoiding specification statements or other forms of non-determinism and declarative specification).

# 6 DISCUSSION

The specification of the communication system is based on a close interaction between the **Call**, **Agent** and **Node** classes. Nevertheless, each of these classes has a coherent and relatively self-contained functionality. In particular, agents do not need to know anything about calls or to access call identifiers, since all management of the past and present calls is the responsibility of the nodes participating in the call.

Mathematical notation has been used to give a complete and precise description of the functionality of methods at a high level of abstraction. Neither natural language, diagrammatic notations or pseudocode would provide such an ability. A large number of validation properties (intuitive expectations about the specification which should hold if it correctly expresses the requirements) can be proved. For example, it is direct to show that all elements of **past_activities** which are calls must have terminated status, and that they can never have their status subsequently modified.

Extensive use has been made of threads, and of interaction between class threads and methods. The reason for adopting this style of specification is explained in Section 7.

The feature interaction problem can be addressed in part by using the object-oriented mechanism of subtyping. We take the feature interaction problem not as being simply the detection of inconsistencies between separate feature specifications, but rather the need to find an incremental way of specifying new features without needing to specify how they interact with existing features. This problem is somewhat similar to that addressed by the concept of subtyping in object-orientation. Subtyping (i.e., semantically meaningful inheritance) of classes ensures that all the laws and behaviour of the old version of a class (the supertype) must remain valid in a new version (the subtype), but new specialisations of these laws and behaviour can be added.

The principle is to specify the overall functions of a communication system (such as a *routing function*,

that defines how a **initiate_call** event at one node is translated into an appropriate set of **request_connection** events at other nodes) at a suitable level of abstraction, together with all desired characteristics of these functions. Class subtypes may then be introduced which redefine the way these functions are performed, but always in a way which is consistent with the existing specification.

Therefore, when a new feature is introduced, it is ideally not necessary to specify how it modifies pre-existing features, but only how it redefines the small number of top-level functions. As an example, a potential problem arising from the addition of new features (incoming caller identification and last caller callback) to the BT network in the UK is the inadvertent display or printing of ex-directory numbers. However, if a number display or bill printing module was specified using an invariant such as

$$\{\mathbf{p.number} \mid \mathbf{p} \in \mathbf{elems(printed)}\} \cap \mathbf{ex\_directory} = \varnothing$$

"No ex-directory number can be printed", where **printed** : seq(**@Printable**) records the billable items printed so far by the bill printing module, then introducing the above features does not require a change to the specification of this function. Instead two new operations are added to the users interface.

Verification of subtyping is, however, required in such a formal framework: here this will include verification that the new operations do not violate the above invariant. The benefits of the object-oriented modelling approach are clear in the case of communication systems, where a system may consist of a heterogenous and dynamically varying collection of user devices and network switches, with different sets of features. Object creation and deletion provides a means of representing the dynamic nature of the network, whilst subtyping enables common aspects of different forms of device to be expressed and reasoned about.

# 7 REMOVING CALLBACKS VIA THREADS

A common mechanism in object-oriented design is the *callback*, whereby two objects can access each other, and where a method of one object may invoke a method on the other even if it was originally itself invoked by that object. Whilst overlapping execution of methods of the same object is allowed in VDM$^{++}$, it should be avoided unless it is essential for improved responsiveness to external events.

Consider the following two classes, which represent a highly simplified communication system:

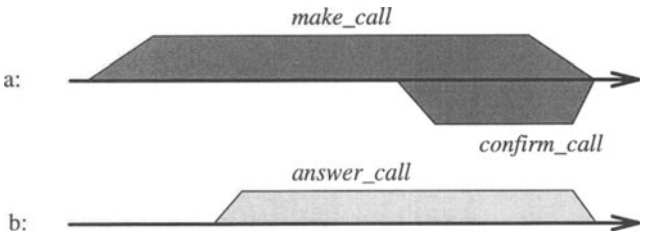
```
class A
instance variables
  x : ℕ;
  b : @B
methods
  make_call() ==
       (x := x + 1;
        b!answer_call(x));

  confirm_call() ==
       x := 0
end A
```

```
class B
instance variables
  y : ℕ;
  a : @A
methods
  answer_call(x : ℕ) =
       (y := x;
        a!confirm_call())
end B
```

Assume that we have two objects **a**, of **A** and **b** of **B** with **a.b** = **b** and **b.a** = **a**.

Then an invocation of **make_call** on **a** leads to the three overlapping method calls shown on Figure 4.

It is possible to reason about the overall effect of the methods of **a** because the execution of **make_call**




can be decomposed into two parts, the first (assignment) sets **x** to $\overleftarrow{\mathbf{x}} + 1$ where $\overleftarrow{\mathbf{x}}$ is the value of **x** at initiation of the method, and the second part sets **x** to 0. Nevertheless this is unnecessarily complex, and can be improved by removing all method calls in **A** and **B** into the threads of these classes:

```
class A'
instance variables
  x : ℕ;
  b : ⌀B'
methods
  make_call() =
       x := x + 1;

  confirm_call() =
       x := 0
thread
  while true do
    sel
      answer make_call ->
                    b!answer_call(x),
      answer confirm_call
end A'
```

```
class B'
instance variables
  y : ℕ;
  a : ⌀A'
methods
  answer_call(x : ℕ) =
                        y := x
thread
  while true do
    sel
      answer answer_call ->
                        a!confirm_call()
end B'
```

The pattern of interaction between these objects is now purely sequential (Figure 5). At any point in time there is only one active object.

# 8 CONCLUSION

This paper has demonstrated the suitability of VDM$^{++}$ for specifying and developing communication systems. Extensive commercial tool support for the language is available, and a number of publications on the language exist. Tutorials and courses on the language have also been given. Because of its basis in VDM-SL, it is expected that the language will have a wide take-up within Europe, and provide a viable