# Proceedings of the 8th Overture Workshop
# 13th September 2010
# London

**Editors: Ken Pierce and Nico Plat and Sune Wolff**

# Introduction

VDM (The Vienna Development Method[1]) is a well-established formal method, which has seen widespread use in both academia and industry. In recent years, extensions have been defined that introduce notions of object-orientation and real-time control to VDM. These extensions bring with them many interesting semantic issues that are yet to be resolved.

This one-day workshop, jointly organized by the VDM community and BCS-FACS, addressed semantic issues relating to the formal specification language of VDM and its derivatives, together referred to as "VDM-10" and to the "Overture" open tools project for VDM-10[2].

The workshop was divided into two themes, each covering a different area of VDM-10: object-orientation and real-time/co-simulation. Each area was presented and discussed in a block of roughly three hours during the day. Each block was introduced by one or more speakers, followed by plenary or group discussions of the identified semantic issues for the remainder of the block. In addition, a time slot was added to give PhD students the opportunity to present their work in the semantics area.

The workshop was followed by a BCS-FACS Evening Seminar, in which Jan Broenink gave a presentation entitled "Embedded Control Software Design with Formal Methods and Engineering Models".

Before you lie the proceedings of the workshop, consisting of all the papers that were presented during the day and summaries of the discussions that took place. We hope you enjoy reading and that this material will be a next step in the advancement of VDM semantics!

The workshop organisers and editors of these proceedings were:

- Ken Pierce (Newcastle University, UK)
- Nico Plat (West Consulting BV, The Netherlands)
- Sune Wolff (Aarhus School of Engineering, Denmark)

[1] See: `www.vdmportal.org`

[2] See: `www.overturetool.org`

# List of Participants

**Nick Battle** Fujitsu, UK.
**Jan Broenink** University of Twente, The Netherlands.
**Joey Coleman** Newcastle University, UK.
**John Cooke** Loughborough University, UK.
**Erik Ernst** Aarhus University, Denmark.
**John Fitzgerald** Newcastle University, UK.
**Leo Freitas** Newcastle University, UK.
**Gudmund Grov** University of Edinburgh, UK.
**Cliff Jones** Newcastle University, UK.
**Peter Gorm Larsen** Aarhus School of Engineering, Denmark.
**Kenneth Lausdahl** Aarhus School of Engineering, Denmark.
**Matthew Lovert** Newcastle University, UK.
**Claus Nielsen** Aarhus School of Engineering, Denmark.
**Ken Pierce** Newcastle University, UK.
**Nico Plat** West Consulting BV, The Netherlands.
**Shin Sahara** CSK Systems Corporation, Japan.
**Marcel Verhoef** Chess, The Netherlands.
**Sune Wolff** Aarhus School of Engineering and Terma A/S, Denmark.

# Table of Contents

# Object Oriented Issues in VDM++

Nick Battle

Fujitsu UK
nick.battle@uk.fujitsu.com

**Abstract.** The semantics of the VDM-SL dialect of VDM is formally defined in an ISO standard. However, the object oriented dialect, VDM++, has only an informally defined semantics and this causes problems both in the development of VDM++ tool support, and with the use of the dialect for formal verification. This paper summarises the semantic issues encountered while developing VDM++ support in the VDMJ tool, and outlines the informal proposals for how to address them in VDM-10.

## 1 Introduction

VDMJ [2] is the open source language tool built into Overture [5]. It includes support for all three VDM dialects. The VDM-SL implementation in VDMJ encountered no ambiguities in the semantic definition of the language, as one would expect given its formal specification [6]. However, although VDM++ has a well defined syntax, it only has an informally defined semantics [3], [4]. This caused several problems during the development of VDM++ support in VDMJ, and led to subtle differences between VDMJ and VDMTools.

Full details of the problems encountered, together with informal proposals to address them are given in [1]. This paper summarizes the most significant issues and their proposed solutions. First, we look at the differences between VDM-SL and VDM++. Then Section 3 considers the initialization of VDM++ specifications and object construction. Section 4 considers inheritance, overloading, overriding and binding. Section 5 covers the semantics of operation pre- and postconditions. The final section identifies some areas that are undefined in VDM++ but have yet to have proposals.

## 2 VDM-SL and VDM++ Compared

VDM-SL specifications have a simple structure, even with the addition of modules (which are not included in the ISO standard). A module comprises a set of types, constants, functions, operations and state. Everything except state can be exported or imported, allowing modules to reference each other, though state can only be directly referenced by the owning module's operations. The initialization of a VDM-SL specification evaluates a single record expression to initialize each module's state values atomically.

VDM++ specifications have a class based structure. A class comprises a set of types, constants, functions, operations and state. A class may define static variables, and object instances of a class may define instance variables. A class may also define a thread

operation and synchronization conditions. Classes can be derived from each other in hierarchies with multiple inheritance; functions and operations can be overloaded, as well as being inherited or overridden. The initialization of the state variables in a VDM++ specification requires the evaluation of a separate expression for each variable, plus an optional constructor operation for initializing object instances.

The extra richness in the VDM++ specification structure means that there are many issues that simply do not have an equivalent in VDM-SL, and this, combined with its informal semantics, is the source of most of the problems that this paper describes.

## 3 Initialization and Construction

**1. When are static instance variables initialized?**

Static instance variables in VDM++ are roughly equivalent to state variables in VDM-SL, except that they can be referenced directly from outside the class, depending on their access qualifiers (public, protected or private), and they can be separately initialized, rather than the whole class being initialized by a single record expression.

This causes problems for tools because it introduces a non-trivial dependency ordering between the set of initializers in the specification. Consider an initializer `var := new X()`. This has an implicit dependency on all the state variables of the class X (including instance variables), and any state in other classes that their initializers access, and any state that the constructor operation accesses, either directly or via operations that it calls, including any superclass constructors, and superclass state initializers. Note that it is not generally possible to statically determine the variable dependency of an operation body.

Even without this complication, VDM++ does not define whether variables are initialized in declaration order —perhaps with undefined forward references— or whether forward references are an error. There is similarly no defined order for the initialization of separate classes in a specification. The goal is to define VDM++ initialization sufficiently clearly that a specification will always initialize into the same state.

**Proposal** *Static variables are initialized by computing the dependency graph of static variables and constants in the specification, including references made via operations and constructors. The order of initialization of independent sub-graphs is not defined. A circular dependency is an error. Initialization cannot create threads, or use loose or non-deterministic statements*.

This proposal gives the maximum flexibility while still giving well defined results. It fits with the philosophy of VDM-SL where there is no definition of initializer ordering, apart from the implicit dependency order between expressions.

Unfortunately, while this does clarify the initialization order, it is still not sufficient. Consider a public static "next value" operation which increments a static variable and returns its new value. If that operation is used to initialize two or more static instance variables (in arbitrary classes), the proposal above tells us that the incremented variable must be initialized first, but it does not define the order of initialization of the two dependent variables — and yet they will have different values.

*This is an outstanding issue*. There is no obvious natural ordering between classes that happen to call the "next value" operation described above to initialize their statics[1]. Both VDMJ and VDMTools do a certain amount of dependency ordering during initialization, but this proposal would require changes to both tools.

### 2. What is the order of initialization of instance variables?

Following from the initialization of the static variables in a class, there is a similar problem with the instance variables in an object. These may have initializers which reference other instance variables (including inherited ones, or those via operations), or visible static variables in other classes, as well as visible constants.

**Proposal** *Instance variables are evaluated using a dependency graph of the variables they depend on, including references made via operations and constructors. Independent sub-graphs can be initialized in any order, and circular dependencies are an error. Variables which have common dependencies are initialized in declaration order*.

This is similar to the proposal above for static variables, though in the case of the instance variables there is a natural declaration ordering which can be used to define the order of initialization of variables which depend on the same variable. As with static variables, both VDMJ and VDMTools do a certain amount of dependency ordering, but this proposal would require changes to both tools.

### 3. How are multiple explicit superclass constructors called?

VDM++ has multiple inheritance, but the syntax does not define a way to explicitly call multiple superclass constructors. Rather, superclass constructors are called as operations directly from the subclass constructor body — in fact they can be called from anywhere. Because all constructors return a value, and because there is no restriction on where such superclass constructors calls have to be made, they tend to be made as the last operation call in a constructor body, constructing the hierarchy "backwards". It is possible to make the calls earlier, and make multiple calls for multiple inheritance, but only with statements that do not return the value, such as `let - = A(x), - = B(y) in ...` or `dcl a:A := A(x), b:B := B(y); ...`.

**Proposal** *VDM++ should have an explicit superclass construction syntax, similar to C++ and C#. Superclasses are constructed before subclasses. It would then be an error to try to call a constructor operation explicitly*.

This makes things clear, familiar and constructs hierarchies in a "natural" top-down order. This would require significant changes to both VDMJ and VDMTools, in the parser, type checker and runtime systems.

[1] This same problem applies to VDM-SL specifications with modules. The module initialization order is not defined by the ISO standard.

**4. If there are multiple superclasses, what is the definition of the order of their (deep) construction? How does that relate to the initialization of their instance variables?**

This isn't defined in VDM++, probably as a result of the relaxed nature of superclass constructor calling (above).

**Proposal** *VDM++ should make a depth first construction of superclasses, in the order of the classes in the "is subclass of" clause. Instance variable initializers are evaluated before the constructor at each object level*.

Again, this makes things clear, familiar and most important of all, it provides a definition of the order of superclass construction. This does require changes to VDMJ and VDMTools, though these would be done anyway as part of (3) above.

**5. Does calling an explicit constructor suppress calls to the default constructor? When should default constructors be called?**

Currently, since a constructor does not know whether its body will call a superclass constructor (perhaps indirectly or conditionally), the default superclass constructors are usually called unconditionally (both VDMTools and VDMJ do so) at the start of a subclass constructor, though this is not defined. If the body goes on to call an explicit superclass constructor, that is performed in addition. You could therefore call several constructors for the same superclass.

**Proposal** *If no explicit constructor is called in the proposed new syntax, then a default constructor is called at the appropriate position in the initialization sequence. If no default constructor is provided by the class, a blank one is provided which just initializes the instance variables*.

Once again, this is familiar and makes the construction sequence well defined. It would require minor changes to the tools.

**6. If an operation is called from inside a superclass constructor, does that call an overridden version in the subclass or does it always call the version for the superclass? Similarly, what happens when calling operations from instance variable initializers.**

This is not currently defined for VDM++. Existing OO languages differ in the way they do this (eg. C++ calls the local member, whereas Java calls the overridden one).

**Proposal** *Both constructors and instance variable initializers should call the local version of operations and functions*.

The motivation for this choice is that it means a class constructor behaves the same way, regardless of whether it is created standalone or as part of a hierarchy. That means that any formal analysis of the class still applies when the same class is composed in a hierarchy. VDMJ currently behaves like this. VDMTools would require changes.

**7. If a class has an invariant over its instance variables, is that invariant calculation suspended during construction? How does that work with class hierarchies?**

This is not currently defined in VDM++, but in general it is not possible to preserve an invariant while initializing all the variables over which that invariant applies. For one thing, all variables start of as undefined values.

**Proposal** *Invariants are only enforced once the construction of an object is complete. This applies recursively to invariants on superclasses in a hierarchy. On completion of construction, the invariant is evaluated once, and on state changes thereafter.*

This is common sense. VDMJ currently behaves like this. VDMTools would require changes.

**8. Are constructors inheritable?**

Most OO languages do not allow the inheritance of constructors, but in principle this is perfectly possible. It is not currently defined by VDM++ and tools differ. An inherited constructor would allow a subclass object to be constructed by calling a superclass constructor (but with the subclass name). It is roughly equivalent to calling the local variable initializers followed by the superclass constructor.

**Proposal** *VDM++ should not allow constructor inheritance.*

There are two reasons for this. It is unfamiliar, in that most OO languages (including possible code generation targets) do not do this; and it would not fit with the proposal that constructors call local rather than overridden members. VDMJ currently does not implement constructor inheritance. VDMTools would require changes.

## 4 Object Oriented Features

**1. What are the rules for polymorphic functions with regard to overloading and overriding?**

This is not currently defined in VDM++. Polymorphic functions are a strange case because their type parameters are under-defined, raising the natural question of when they can be overridden in a hierarchy, or overloaded.

**Proposal** *A polymorphic function can be overridden by a function of the same name, with the same shape of parameter types as the superclass (for example,* `seq of @T` *and* `set of @T` *are not the same shape). Polymorphic functions with the same name can overload each other if they have parameters that are a different shape.*

Both VDMJ and VDMTools would require some modification to do this.

**2. What are the rules for curried functions with regard to overloading and overriding?**

This is also currently undefined in VDM++, and is similar to the issue of polymorphic functions though the parameter types are well defined.

**Proposal** *A curried function can be overridden by another function with the same type signature (i.e. including all the function type returns). Polymorphic functions, curried functions and simple functions can all overload each other as long as they are distinguishable by parameter types*.

Both VDMJ and VDMTools would require some modification to do this.

**3. Do members that don't have covariant return types and contravariant parameter types (with respect to a superclass member) override the superclass, or are they just overloads?**

The issue here is to do with *strong behavioural subtyping* (SBS) [7]. Without SBS, formal analysis and verification is frustrating with object oriented specifications. For example, a simple operation call `op(123)` would be well defined in VDM-SL and refer unambiguously to an operation with pre- and postconditions that allow the call to be analysed in the current expression context.

But in an object oriented specification, the operation call is really a call to `self.op(123)`, where the type of self is unknown — except to say that it is either the type of the current class *or a subclass*. If we are actually being called from a subclass (i.e. from an inherited operation) and that class overrides the `op` operation, then `op(123)` could currently do absolutely anything, rendering any formal analysis of the calling operation invalid.

To avoid this chaos, SBS limits the definition of subclasses to have behaviours that are a subset of their superclass' behaviour. This comes in two parts: the types of the parameters and return types of the overriding member are restricted by SBS, so that the override is guaranteed to "fit" in the same calling context as the overridden operation; secondly, the pre- and postconditions of the override have to be no more restrictive/at least as restrictive as the superclass, so that the behaviour of the subclass "fits" within that of the overridden operation (see (4) below).

**Proposal** *VDM++ should implement strong behavioural subtyping in class hierarchy overriding*.

Without this, the formal analysis of anything other than trivial VDM++ specifications would be extremely difficult, and specifications would be "fragile", requiring reanalysis if classes are used in different contexts. Both VDMJ and VDMTools would require some modification to do this. The SBS requirement would produce new, well-defined proof obligations.

**4. How do preconditions and postconditions of overridden functions and operations affect the preconditions and postconditions of the overriding function or operation?**

This is related to strong behavioural subtyping as mentioned above. The proposal is consistent with SBS.

**Proposal** *An overriding member's preconditions can be no stronger than the overridden member's preconditions. An overriding member's postconditions must be stronger than the overridden member's postconditions. An overriding member must have a covariant return type and contravariant parameter types (else it is an overload).*

Both VDMJ and VDMTools would require some modification to do this. The SBS requirement would produce new proof obligations.

**5. Can static functions/operations be called via an object reference? If so, do static members take part in polymorphic behaviour?**

It should be considered here that functions in VDM-10 are all implicitly static, and yet they are expected to be callable via object references. They are therefore expected to act like normal polymorphic calls.

**Proposal** *Static functions and operations can be called via an object reference, and do act polymorphically.*

Note that neither C++ nor Java act this way. In both languages a static member call is statically bound using the declared type of the object reference — e.g. `obj->op()` is the same as `Cls::op()`, where `Cls` is the class name of the declared type of `obj`. Both VDMJ and VDMTools currently work as proposed.

**6. In the case of "diamond inheritance", where multiple inheritance has common superclasses, how is the hierarchy constructed?**

Consider the hierarchy in Figure 1. The problem here is that both B and C would normally want to construct A — i.e. they would have some sort of initializer list or "super" call to construct A in the way they require. But if there is only one shared instance of A in the final D object, we can't construct it in a way that guarantees to satisfy both B and C.

In C++, base class sharing like this is called *virtual inheritance*. To avoid the conflict, the C++ compiler insists that the most derived class (D in this case) constructs A itself, before any other construction occurs. That avoids the conflict between B and C, but from a formal language perspective this is a big problem. It means that B and C cannot influence the construction of A, even though they may depend on a particular construction in order to function. In particular, any postcondition of their constructors could easily be broken by D.

**Proposal** *In diamond inheritance, subclasses have their own private copies of common superclasses.*

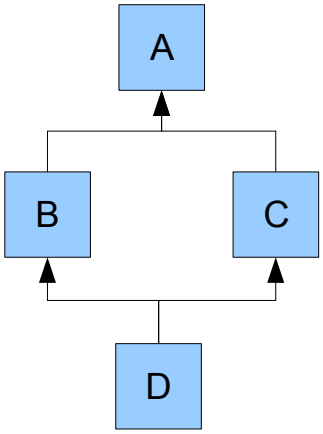


Fig. 1: Diamond inheritance

Construction then proceeds in the left-to-right, top-to-bottom order that you would expect. That allows any analysis and proof work done for B and C in isolation to still be applicable when the classes are composed in a hierarchy. This would require modifications to both tools.

## 5 Operation Pre/Postconditions

**1. What are the prototypes and semantics of a VDM++ operation's precondition and postcondition functions?**

In VDM-SL, an operation precondition has visibility of the state of the local module, and the postcondition has visibility of the state both before the call and after the call. This is managed conveniently by "Sigma structures": the state of a VDM-SL module is also available in a read-only mode via a structure of the same name as the state "record". Since a VDM-SL operation cannot directly affect state in other modules, this means that the Sigma record contains the complete state of the system as far as the pre- and postconditions are concerned[2].

As described earlier, state visibility in VDM++ is much greater than in VDM-SL. Public state from arbitrary classes can be accessed directly by an operation. Therefore the state of the system that should be available to operation pre- and postcondition functions is potentially the total state of the specification, including "self" and all static

[2] Although imported operations see state in other modules and can affect the operation, so perhaps modular VDM-SL is deficient here too?

variables and all instance variables of all objects which can be referenced via those variables. This means that the prototypes of operation pre- and postconditions in VDM++ cannot be as simple as those in VDM-SL.

**Proposal** *VDM++ pre- and postcondition functions are similar to those in VDM-SL, except with "self" parameters rather than Sigma parameters. The self objects represent a deep copy of the state. Operations which reference static variables do not have pre- and postcondition functions*.

This is unsatisfactory and still very difficult to implement in the tools. VDMJ implements a shallow copy of the "old" object (only local instance variables); VDMTools does not even define the pre- and postcondition functions in VDM++. The syntax of VDM++ must change in any case to permit (old) instance variables to be referred to in a postcondition or an externals clause — presumably something like `iv.field` (grammatically, old names are *identifiers*, whereas they would have to be more like *state designators*).

## 6 Further Issues

This paper has summarized the key issues and proposals in [1]. However, there are several OO issues that have not been covered here or in [1]. In particular, there are subtle binding issues in the case of overloaded members, some of which are overridden, especially in light of the type checker's *possible semantics*.

Note that [1] does *not* cover issues regarding the concurrent threaded aspects of VDM++, in particular whether thread scheduling is deterministic. There are also issues in VDM++ and VDM-RT where raw operation names are used, but when several overloaded operations of that name exist, e.g. `#act(op)`. As noted in [1], existing OO programming languages differ in the semantic areas discussed in this paper. This will be a challenge for automatic code generation from VDM++.

## References


1. Nick Battle. VDM++ Object Oriented Issues, Issue 0.2, 2009.
2. Nick Battle. VDMJ Design Specification, Issue 1.1, 2010.
3. CSK Systems Corp. VDMTools — The VDM++ Language Manual ver.1.0, 2008.
4. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.
5. Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1), January 2010.
6. P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.
7. Wikipedia. Liskov substitution principle — Wikipedia, the free encyclopedia, 2010. [Online; accessed 31-August-2010].

# The Case for Simple Object-Orientation in VDM++

Erik Ernst

Department of Computer Science, Aarhus University, Denmark
eernst@cs.au.dk

**Abstract.** In the process of defining a precise semantics for VDM++, there are many possible criteria for the selection of features and their detailed properties. We take an outsider's view on the known difficulties and ambiguities, and argue for a strong emphasis on the most foundational features. Based on such a simple core language with few but powerful primitives, additional features could then be supported by means of a reduction to the core language, which allows for an open-ended set of additional features, including multiple variants. This approach reduces the complexity for tool implementers, and ensures semantic transparency for users. The use of additional features may represent a trade-off between convenient and familiar usage and semantic transparency, but at least the division between core and additional features is explicit, such that the user is in control of the trade-off.

## 1 Introduction

This paper is an outsider's view on the ongoing standardization process for the VDM++ specification language. It is based on experience in the area of object-oriented language design and formalization, but it will surely reveal a rather limited amount of knowledge about the creation and history of the existing variants and extensions of VDM, as well as the associated tools. Nevertheless, it is our hope that the input will be useful in the standardization process. The focus is entirely on the semantics of the underlying programming language aspect of the specification language.

We argue that it will be beneficial to aim for a simple core language, focusing on the most crucial object-oriented entities and mechanisms such as objects, methods, and classes. Other language concepts—e.g., static state, access modifiers, and constructors—would then be outside the core language, but they could be supported by means of an interpretation in terms of core language constructs, that is, as a kind of syntactic sugar.

The rest of this paper will extend on this position in Sect. 2, discuss which features to include in Sect. 3, how the omitted features can be expressed in Sect. 4, and how they could be given a well-understood status in Sect. 5. Section 6 briefly covers related work, and Sect. 7 concludes.

## 2 Basic Language Design Rationale

We believe that it would be useful for VDM++ to embody sufficient generality to cover the specification of software written in several different object-oriented languages. However, the semantics of concrete, full-fledged, object-oriented languages differ in a large

number of more or less subtle ways, which implies that a full and direct coverage of all features of any of them will create serious and subtle problems in the treatment of other languages.

As a concrete example, consider inheritance. In the Java programming language a class may implement two interfaces containing identical declarations of a signature of the method $m$; it must then contain an implementation of $m$ with this signature, and this implementation will be the meaning of invocations of $m$ on a reference of either interface type. In C#, it is possible to provide two distinct implementations $M_1$ and $M_2$ of $m$ associated with the two interfaces, such that invocations on the first interface will run $M_1$, and $M_2$ will run for the other one. With the Java semantics, it is not possible to get two different behaviors for invocation at the two interface types; with the C# semantics, it is not guaranteed to work if $m$ is called at the type of the class, because $m$ is ambiguous. This situation could arise in practical code when refactoring causes some code to be moved to a subclass, or it could arise in a theoretical setting where standard semantic specifications rely on the absence of name clashes in each declaration scope. Even for such a seemingly small, subtle corner of the language semantics, there is no generalization of the semantics that will faithfully cover both.

We believe that the most safe and useful approach to deal with semantic incompatibilities like this is to support a restricted basic set of features in VDM++ which is as faithful as at all possible to the core of every language that VDM++ is intended to represent, but which stops short of making commitments to all the incompatible corner cases. Hopefully, it will then be possible to make such choices explicitly, possibly at the cost of a more verbose program. In the example of having $m$ in two distinct interfaces, it would be possible to move $m$ to a fresh super-interface and have it shared, or rename one of them, and avoid the clash. The rationale is that the resulting program would express the intentions more clearly, whether the final software is intended to be implemented using Java, C#, or any other language among the intended targets.

Note that it is assumed that such a renaming process is feasible, even though renaming a method in an interface may affect many locations in client code in a large system. The rationale behind this assumption is that the work on a specification is intended to start early and expected to influence the design of the software. When this is indeed the case, those detailed design decisions will be made in context of the specification, which means that the problems (such as name clashes) are avoided during the construction of the system rather than meticulously removed after the fact. If, moreover, the software is simply more comprehensible, maintainable, and portable as a result of the avoided corner cases, that is simply a valuable extra benefit.

The opposite position would emphasize that unanticipated reuse of code (including the ability to resolve all kinds of name clashes in flexible ways, and the ability to avoid editing existing source code when reusing it) is crucial in large software projects. We propose that semantic clarity is given priority over the proliferation of sophisticated code adaptation mechanisms, even if they would expand on the ability to support unanticipated code reuse. After all, correctness is an overarching goal of any serious specification effort, and simple straightforwardness in the code is very helpful in achieving this goal.

In summary, we argue that subtle semantic incompatibilities should be weeded out of VDM++ by choosing a simpler semantics covering the core of all intended target languages, rather than aiming for total coverage of any one or all of the target languages. Programs will then have to be constructed or adjusted to fit in this context, and they may have to be expressed in a somewhat more verbose manner. We believe that this trade-off serves the goals of having a specification in the first place.

## 3 Included Features

In order to express object-oriented semantics that fits practical software, the semantic primitives must reflect the core structure of the languages used to express this software. Given that formal specifications are intended to provide explicit representations of known properties of the software it seems natural to focus on statically typed languages, because static types is just another kind of explicit representation of such properties. Following the mainstream, this implies that object, method, and class are the core concepts for VDM++, and name declarations should include explicit types. Classes provide the blueprint of objects, in the sense that an object includes state and behavior, and the class of the object specifies the state space in terms of a set of declarations and the behavior in terms of a set of method implementations.

Shared structure in classes is commonly achieved using inheritance and exploited by clients using subtyping, and these two concepts should also be part of the semantic core.

Inheritance may be single or multiple, and there is a plethora of ways to handle name conflicts, lookup rules, access control, and other issues. We propose that inheritance should be multiple (in order to include basic C++ semantics and the concept of interfaces), but that it should not include detailed commitment to various sophisticated conflict resolution strategies. Name clashes and the associated lookup rules should rely on explicit specifications (disambiguation etc.) rather than using the rules of a particular language.

One possible approach would be to use single inheritance as the starting point, and to use an approach to multiple inheritance where the outcome of the composition of the superclasses is specified explicitly whenever there is a potential conflict, rather than computing it from the superclass graph in a complex and language specific manner.

Subtyping could be derived directly from subclassing, assuming that subclassing is defined in such a way that it is always sound to do so. Alternatively, it could be established entirely explicitly, which would be more expressive, but probably incompatible with the semantics of one or more target languages. After all, it is generally accepted that a subclass is a subtype, but the variability is enormous when going beyond that point.

Finally, it is obvious that a specification language like VDM++ should include specification elements, even though these elements are not necessarily present in the target languages. Thus, methods should have pre- and post-conditions and classes should have invariants.

# 4 Omitted Features

This section lists a number of concrete language features and argues that they can and should be omitted, illustrating how they can be replaced by more explicit constructs in a simpler core language.

## 4.1 Static Features

The notion of static state is supported by several object-oriented languages. The semantics of this concept is that certain global variables may have names that include the names of classes as a prefix, almost as if each class would be a singleton object with state of its own—but generally without the logical consequence that one should be able to get the class of the class, create a new instance (i.e., a new class), and obtain a fresh copy of this 'class state'.

This concept could be directly represented by global variables, possibly allowing names on the form `C.x` where `C` is the name of an existing class. Lookup could be aided by syntactic sugar which allows omitting the class name inside the class of that name.

Initialization of static state is discussed separately below, in Sect. 4.3.

Similarly to static state, the notion of static methods corresponds to global procedures with the added naming twist of including a class name. They could be directly represented as global procedures, with similar syntactic sugar as in the case of static state.

If global state and global procedures are considered inappropriate then static state and static methods should be treated likewise, in which case there could be a simple option that prohibits this type of feature in a given specification.

## 4.2 Constructors

Constructors specify user-defined initialization behavior. They correspond to global procedures which implicitly include a primitive allocation operation and possibly an invocation of a superclass constructor unless an explicit invocation is given, and then they effectively return the newly constructed object; here, 'effectively' hints at the fact that the passing of the object among super- and subclass constructors is implicit, as well as the final delivery of the object to the caller.

This view on constructors explain the facts that they are not inherited, they are 'invoked on classes' rather than on objects (usually using a special `new` operator along with the class name, rather than using normal invocation syntax), and they do not return anything, syntactically.

The most difficult issue with constructors is that they are expected to establish the class invariant for the newly created object. However, this is a very subtle process due to the fact that constructor code may call arbitrary code in the system. In C++, virtual member functions are redefined to use direct invocation of the implementation in the class of the current constructor rather than normal late binding, but in Java and

elsewhere, the normal semantics of method invocation is retained. In other words, constructors tend to pollute the entire system with non-standard method call semantics or the danger of having objects around whose invariant does not hold.

Using global methods with explicit primitive allocation that receives explicit arguments for every part of the object state avoids exposing an object that does not satisfy its invariant, at the expense of needing to separate out the incremental computation of that list of construction arguments to a chain of class specific global procedures for each superclass. In return, the anomalous features of constructors are avoided entirely, as well as the subtle differences between different languages such as direct/late-bound method invocation.

### 4.3 Declarative Initialization

Initialization expressions associated with declarations impose a certain level of declarative implicitness, because there is a need to choose the ordering of initialization behavior such that it respects the dependencies among the initialized features. For an individual object, we have proposed that the initialization is generally made atomic by requiring the entire state at the primitive allocation time.

With global state (such as static variables), it is not likely to be pragmatically convenient to require such an atomic allocation approach. Initialization could be made atomic on a per-module basis, as a hybrid approach, but a multistep approach seems unavoidable in complex systems, and we proceed to discuss such an approach for global state.

It is a difficult issue to decide how to choose a global initialization order for a given system, because initialization expressions are generally unrestricted with respect to the use of the language semantics, and hence it is an undecidable problem to determine the exact dependencies in general. In concrete cases, however, it is reasonable to expect programmers to be able to explicitly specify an order that works for a given, total system.

It may also be possible to write initialization procedures for parts of a system, say for a single module or for a set of modules that have mutual dependencies, but all the possible scenarios for such modular solutions may be handled by ordinary programming of explicit procedural code.

When a program runs, often by running a global procedure called something including the name `main`, it relies on global state having been initialized before it is used. This could be achieved by writing a new `main` procedure for the total, concrete system being run, which at first explicitly calls a procedure that initializes the global state of the system, and then calls the original `main`.

This explicit approach allows for expressing arbitrary concrete semantics, at the expense of writing all this initialization code where at least one top-level procedure is not reusable in a different system. However, a mechanism similar to syntactic sugar could be added in order to generate this code in a systematic way from a given set of initialization expressions, in case the semantics of a particular language is desired and sufficient for a given system.

### 4.4 Overloading

Static overloading consists in distinguishing declared names with the same spelling based on some additional features which accompany this name in the declaration and at usage points. For instance, static overloading of method names work by extending the name of the method with the types of the actual arguments as discriminators in the selection of one of the methods with that name for a given call site. This type of feature gives rise to an absolutely stunning amount of messy detail, and we propose that the subtleties and incompatibilities of this entire topic area is avoided by simply requiring that declared names are different in each scope.

A kind of syntactic sugar for removing static overloading could be provided, by means of name mangling which explicitly extends the names with the external discriminators on the declaration side, and includes the knowledge about declarations for the treatment of application sites (such as call sites in case of a method declaration).

### 4.5 Nesting

Class nesting (e.g., inner classes in Java) does not have a semantics in mainstream languages that goes beyond what could be expressed using a flat class space; access to enclosing objects would then be supported by having an explicit reference to the nearest enclosing object, and using an explicit chain of such `enclosing` references rather than lexical scope rules. The syntactic sugar needed to remove nesting would probably be more extensive than the other examples, but the `javac` compiler uses this technique which demonstrates that it is entirely plausible.

### 4.6 Access Control

Access declarations such as `public` and `private` serve to constrain the use of certain features by certain pieces of code, thus helping programmers to reason about the code by reducing the amount of code that they need to consider in order to understand the treatment of a given declared feature. This aspect of programs is generally defined to be a purely compile-time issue, and it should not be hard to add this to a given specification language by means of some kind of annotation (possibly simply a specially formatted comment), along with a separate tool that checks for conformance.

## 5 Well-understood Reductions

The whole point of omitting many features as described in the previous section is that the core language is simple to understand and simple to reason about. It also covers more target languages because it does not commit to many "dark corners" where the target languages have subtly (or wildly) incompatible semantics. It also gives priority to the explicit expression of the desired semantics, in order to cover many different choices faithfully, and at the expense of added verbosity.

Now various kinds of syntactic sugar kick in, in order to provide the pragmatic benefits of concise notation and familiarity. It is possible to have any number of language

styles available, as long as each of them is equipped with the required syntactic sugar to reduce specifications to their core form.

It could be argued that a simple core language along with syntactic sugar that reduces programs in more pragmatic languages down to the core form creates complications for the comprehensibility of practical specifications. In short, we start with a nice, readable specification in a familiar language, but it is necessary to look at a much larger specification produced by syntactic sugar elimination, in order to understand what it really means.

It would be highly desirable to provide an alternative to this scenario that avoids the need to look at large, generated specifications. This could to some extent be achieved by analyzing the syntactic sugar elimination and proving, once and for all, that this elimination process and the semantics of the core specification language is such that the original pragmatic ('sugared') language can be considered to have a specific, well-known semantics.

This would be useful in one more way, namely because it enables code generation based on the sugared language, which enables it to use the features of the most closely related target language directly, rather than via an expansion into a core language specification which might cause serious problems for the time or space efficiency.

For satisfaction of proof obligations it may not be a problem to work with the core language, because it has a simpler semantics and may be much easier to create proofs for than the sugared language, even if there will be a larger number of proofs.

## 6 Related Work

This paper is almost exclusively a reflection on the description of the difficult issues encountered while working towards a formalization and standardization of the VDM++ specification language described in [1]. The VDM++ method and specification language at different points over time are described in [4, 3, 2].

## 7 Conclusion

This paper offers an outsider's view on the VDM++ standardization efforts, and argues that it would be useful to strive for a simple core language having simple and widely used object-oriented features, and using added explicitness and verbosity to enable a precise modeling of the more subtle features of several more or less semantically incompatible target languages, probably supported by a relatively rich layer of syntactic sugar that enables end users to write their specifications in a language that is closer to their preferred target languages. In particular, we discussed how a number of concrete language mechanisms can be eliminated and emulated by using explicit coding with simpler constructs; for instance, static features and static overloading could be eliminated, and initialization order could be made explicit and hence avoided as a general, undecidable problem. The expected outcome would be that VDM++ will be able to cover a broad range of statically typed object-oriented languages faithfully and in a manner that is suitable for reasoning about correctness, with the potential for offering a number of familiar and concise surface layers based on syntactic sugar.

## Acknowledgments

Peter Gorm Larsen and Nick Battle were very helpful in letting me get a glimpse of the various VDM specification languages and associated tools, the existing literature, and the issues uncovered in the implementation and specification of VDM++ so far.

## References

1. Nick Battle. VDM++ Object Oriented Issues. Technical report, Fujitsu Services, 2009. Version 0.2, 15/12/09.
2. E. H. Dürr and N. Plat (eds.). VDM++ language reference manual. Technical Report AFRO/CG/ED/LRM/Vll, Cap Volmac, The Netherlands, August 1995.
3. Stuart Kent and Richard Moore. An axiomatic semantics for VDM++: OO aspects, 1993.
4. K. Lano and S. Goldsack. Integrated formal and object-oriented methods: The VDM++ approach. In *In proceedings of Methods Integration Workshop*. Leeds Metropolitan University, (Supported by BCS FACS), March 1996.

# The Object-Orientation Discussion Session

Editor: Ken Pierce

School of Computing Science, Newcastle University, NE1 7RU, UK
K.G.Pierce@ncl.ac.uk

## 1 Introduction

A plenary discussion was held after the two talks were given by Nick Battle and Erik Ernst. This section summarises the points raised in that discussion, as well as key points raised during the talks themselves. The discussions fell broadly into two categories. First, issues regarding the history of VDM++, including its intended purpose and the reason for its evolution. Second, issues regarding the future of VDM++, such as the features of a possible core language and potential forms of semantics.

The general sense from the morning session was that we as the VDM and Overture community should start again. VDM++ has grown over the years in response to the wishes of users and projects, but perhaps it is now time to take a step back. The discussions gave us the chance to take a clean slate and was seen as an opportunity to open up the floor and discuss how we want to change things. Radical changes were not discouraged.

Erik Ernst summed up the discussion succinctly with an outsider's view on the forces acting on the VDM and Overture community:

**Programming language view.** From a pragmatic point of view, there should be a small gap between the modelling language and implementation language(s), allowing for tool support and code generation.

**Specification language view.** The modelling language should permit reasoning about software, including the ability to underspecify where necessary, as well as establish proofs.

**Object-oriented view.** The language should support object-oriented constructs such as inheritance.

Our challenge as a community is to find a balance between these three forces. There are clearly trade-offs between expressiveness and analysability that need to be considered.

## 2 History and Purpose of VDM++

Discussion about the present state of VDM++ suggested that people believe VDM++ has lost its way. Cliff Jones noted that VDM++ gets *talked* about as a programming language. During the creation of VDM-SL, the designers began with a notion of refinement and a proof theory and perhaps VDM++ has lost sight of this original goal.

Nick Battle added that while VDM-SL makes sense (in that it can be used to model systems), much of the detail of object-orientation moves away from modelling. It's object-orientation for its own sake, which is hard to analyze formally. Cliff asserted that if you can't reason about a specification, you haven't got a specification. Leo Freitas suggested that many of the problems are caused by starting with how they *are* in a programming languages, not asking how they *should be* in a programming language.

On the creation of VDM++, Peter Gorm Larsen stated that there was pressure in the mid-1990s to use more object-orientation in software and that this is still the case. Therefore the aim of VDM++ was to provide people who program with object-orientation a means to express more abstract models of their systems, but still be able to compare their specifications to UML class diagrams, for example. Nico Plat added that VDM++ has grown over the years in response to customer wishes, but perhaps now is the time to take a step back. Perhaps fundamental changes are required.

Cliff agrees that there are a number of things missing from VDM-SL and he is not opposed to extending VDM with object-orientation, but doesn't want all the complexity. Good modularisation is one missing feature, as is a notion of concurrency (or more abstractly, a way of defining the order in which things happen), to which Marcel Verhoef agreed. Cliff also believes that there are specifications which can be expressed naturally with a notion of objects (e.g. the Mondex electronic card system) and features of object-oriented languages (and in particular concurrency) that are well worth adopting (e.g. the work on "$\pi o \beta \lambda$"). He advocates starting from the top-level and considering what can be deduced from a class definition. Abstraction is the key.

John Fitzgerald suggested that the main questions in our effort to refocus are:

- What is VDM++ for?
- Who are our target users?
- Who are we trying to help?
- What's the typical usage scenario?

There are these forces pulling us in different directions and we might not be able to satisfy them all, but the best way might to be to start by asking who we're trying to help.

Erisk Ernst suggested that VDM++ should be a tool that makes it possible for people to establish a greater degree of provable knowledge about systems. It should be compatible with mainstream languages, but remain strict at the same time, making proofs about programs practical. To this John Fitzgerald added that it could be seen as a universal modelling language for systems. As JML is to Java, so VDM++ is to a wider range of languages.

## 3 Future Direction of VDM++

Discussion on the future direction of VDM++ focussed mainly on the potential consequences of adopting Erik's approach and what features we would wish to keep or remove. One thread of discussion was how to judge which language features should appear in the core language and which should become syntactic sugar.

Peter suggested that overloading, constructors etc. could be removed. Cliff advocated starting without inheritance. Peter responded that this would drive away people who familiar with object-oriented programming languages and that it should therefore be in the core language (eventually). Cliff would also like to see specification of concurrency and scheduling handled at an abstract level (e.g. through rely-guarantee conditions), before considering it at an implementation level (e.g. through threads).

Marcel wondered if we want to have some form of reflection (either in a core language or VDM++), since it is a feature of other object-oriented approaches. Erik suggested that minimal reflective support (such as lookup and dynamic calling) should be okay (with some form of exception handling or run-time checks). The ability to edit at run-time however is clearly very bad for reasoning.

Numerous people suggested that a good way to decide this was to take some of the existing examples and rewrite them in a core language to see what they look like. Based on that, we can decide what is needed in the core language and what can be left as syntactic sugar. Perhaps many surface languages (targeted to specific languages like C++ or Java) are not needed.

It was also suggested that certain language features could be removed altogether if they are not used in practice. John Fitzgerald wondered if we have a large enough code base in VDM++ to assess the impact of removing certain language features? How much use is made of the features that might not appear in a core VDM++ language? A survey of the FeliCa[1] work and other examples in the repository could be beneficial.

In response to this, Nick noted that a lot of the issues he presented are corner cases and that at present, most examples seem to work. They need to be considered in a tool, but are not necessarily used in practice. So either VDMJ works the same as VDMTools, or more likely, these features aren't used and general examples are simple. Peter noted that this would imply little cost to moving to a simpler core language.

One major problem is that object-oriented programming languages (such as Java and C++) are subtly different, therefore catering to more than one target language is hard, particularly for things such as code generation. Atomicity assumptions in different languages may also differ. Leo noted that Object-Z had similar problems to those Nick discovered. Nick expects to find more issues as work continues, though he would be much happier if his proposals were accepted in VDMJ/Overture and VDMTools.

John Fitzgerald asked Erik if he was aware of any other attempts to use Erik's core language approach for other languages or frameworks from which we can learn? Erik replied that he was not aware of any with the same constraints: the fact that VDM++ already exists (which fixes a number of language choices) and the fact that we want to target many languages.

Cliff suggested that there are two issues. The first is that you want a specification that can be implemented in many languages (including object-oriented languages). These implementations may be more or less complex. The second is that some specifications are easier to write with object-oriented thinking, e.g. the specification of the Mondex system benefits from some form of object thinking, where cards can be thought of as stateful objects. However, only simple examples are likely to be tractable.

[1] The FeliCa work is closed source, but it was suggested that it might be possible to write a tool to perform a survey without having to view the source code directly.

As a solution to the problem of multiple target languages, Cliff suggested that abstraction should be the key. The way to get to an efficient implementation in a target language is through abstraction, not syntactic sugar. Be abstract, state the properties you need (including assertions about concurrency, for example) and then use your programming language constructs in the design process to reach an efficient implementation. Erik believes that abstraction can be applied, but is worried about the need for proof rules for complex target languages.

Various people warned against making the core language too simple. Erik noted that where simplifications imply definition of a new primitive, this can lead to an explosion of expressive power. For example, existential types (a consequence of object-orientation) versus objects as primitives. Cliff also noted that explosion of proof obligations could be a problem. If there is too much syntactic sugar, then the good features about abstraction (such as object-orientation) do not help to simplify the proofs when translated into the core language.

Leo Freitas introduced the idea of looking to Microsoft's Boogie as an example. It is a very simple language, into which programs in C, C++, Spec#, C# and so on can be translated and verified. Because it caters for many diverse target languages it is a very small language, which can generate more proof obligations than code. The idea would not be to emulate Boogie, but to have an intermediate language.

The subject of the form of the semantics was raised by Cliff. Peter noted that an axiomatic semantics already exists for the original VDM++, but not for any modern features. He also notes that ISO VDM-SL has a core abstract syntax and denotational semantics, with advanced features are defined in terms of this core language (though John Fitzgerald pointed out that VDM++ would be on a different scale).

John Fitzgerald raised the point that a semantics is a tool for something, perhaps developing an interpreter or proof support. Does a denotational semantics satisfy either of those needs? Peter replied that is was sufficient during the writing of the VDM-SL portion of VDMJ. Nick was able to ask questions and Peter was able to answer them by looking at the ISO standard. Whether or not an operational semantics would be easier for the tools is an open question.

Cliff believes it is obvious that we can give a semantics for an object-oriented language or for a real-time language, but not necessarily both at the same time — and that the semantics will look very different to that of VDM-SL. Peter noted that no matter what method we choose for concurrency or real-time modelling, it will all map down to the same core language. So the core language will have to have the capability of giving semantics to concurrency and distribution aspects.

Cliff noted some delicate problems in giving a real-time semantics to a language. For example, if you can block another process, you have to eliminate behaviours. If the progress of process $A$ is dependent on process $B$ and $B$ can't meet $A$'s time limits because it's dependant on process $C$, then you then have to freeze all the other behaviours and show that those behaviours disappear. Cliff doesn't know how to do this.

## 4 Conclusions

Overall there was agreement that we can pursue *both* Nick and Erik's suggestions in parallel, to the benefit of the Overture and VDM community. As noted by Nick Battle, VDMTools and VDMJ/Overture differ in their current implementations and this needs to be addressed in the short term. It was agreed that Nick's proposals should be incorporated into VDMTools and VDMJ/Overture.

In the medium to long term, the community will look at a core language and decide the features that we really need. We should advocate the removal of features that shouldn't be used and focus on our strength — of being as abstract as possible, but not more so. Our goal should be a core language with a well-defined semantics. This will allow robust tool support (with the potential for code generation) *and* proof support.

It was noted that the long term can easily become indefinite, so it is up the Overture and VDM community as a whole to move this process forward.

# A Semantic Model for a Logic of Partial Functions

Matthew J. Lovert

School of Computing Science, Newcastle University, NE1 7RU, UK
matthew.lovert@ncl.ac.uk

**Abstract.** Partial functions and operators arise frequently in program specifications. The application of partial functions and operators can give rise to non-denoting (undefined) terms. Non-denoting terms that are then arguments to strict relational operators lead to non-denoting logical values which First-Order Predicate Calculus gives no meaning to. One logic that copes naturally with propositions over terms that can fail to denote is a non-classical (three-valued) logic entitled the *Logic of Partial Functions (LPF)*. The aim of this paper is to provide semantics for LPF, which is done through a *Structural Operational Semantics (SOS)* which provides an intuitive introduction into how LPF copes with non-denoting terms that can arise.

## 1 Introduction

Terms that can fail to denote proper values arise from partial operators such as taking the head of a list and from applications of recursive functions; such terms occur frequently in program specifications [6, 11, 7]. This raises the question of how proofs about such terms can be conducted formally. To illustrate the issue, consider the following (deliberately partial over all integers) function [6]:

$$zero : \mathbb{Z} \to \mathbb{Z}$$
$$zero(i) \quad \triangleq \quad \textbf{if } i = 0 \textbf{ then } 0 \textbf{ else } zero(i-1)$$

this function returns $0$ when $i \geq 0$. However, when $i < 0$, $zero(i)$ will fail to denote a value of the expected type and such a term is referred to as a non-denoting (or undefined) term. Now consider the following property of this function[1]:

$$\forall i \in \mathbb{Z} \cdot zero(i) = 0 \vee zero(-i) = 0$$

A reasonable understanding of the $zero$ function suggests that this property is $true$; in particular, the disjunction is true for the least fixed point interpretation of the recursive definition of $zero$. However, on closer inspection it becomes clear that one of the disjuncts will fail to denote a value, with the exception of the case when $i$ denotes $0$. It

[1] The function might look to be perversely partial but it and the property have been deliberately chosen to be as simple as possible to illustrate the issues around non-denoting terms (e.g. in the property considered, there is no obvious "guarding" predicate to be used on the left of an implication). In realistic applications, it is frequently difficult to spot the defined domain of a function — [6, 8] use a function of two parameters where definedness depends on a relation between the two arguments.

becomes quite clear that the truth of this property relies on the truth of disjunctions such as:

$$zero(1) = 0 \vee zero(-1) = 0$$

which reduces to [2]:

$$0 = 0 \vee \bot_{\mathbb{Z}} = 0$$

since $zero(-1)$ does not, in the least-fixed point denote an integer; with strict (undefined if either operand is undefined) relational equality this further reduces to:

$$\textbf{true} \vee \bot_{\mathbb{B}}$$

which makes no sense in FOPC. Since we are interested in reasoning formally about such properties, we have decided to make use of a non-classical (three-valued) logic known as the *Logic of Partial Functions (LPF)* [1] which copes naturally with propositions over terms that can fail to denote. In LPF the above property is $true$ and its proof presents no difficulty.

The objective for this paper is to provide semantics for LPF through a *Structural Operational Semantics (SOS)*. Because decision procedures etc. can go underneath the notion of provability in the logic ($P \vdash Q$), it is important to fix the semantics of truth in a model ($P \models Q$). The author of this paper is currently undertaking research on mechanised tool support for proofs involving non-denoting terms using LPF.

### 1.1 Outline

In Sect. 2 we provide an overview of LPF. In Sect. 3 we define the abstract syntax and the context conditions for our chosen expression constructs before presenting a *Structural Operational Semantics (SOS)* which defines their evaluation according to the semantics of LPF. Finally Sect. 4 highlights future work alongside some conclusions.

## 2 The Logic of Partial Functions (LPF)

When terms involve the application of partial functions and operators, they can fail to denote proper values. Over the years many different approaches have been suggested to handle such non-denoting terms which we will not discuss here, instead we refer the reader to [5, 6, 11] for surveys of a number of these approaches. In this paper we focus our attention only on an approach known as LPF[3] which is a first order predicate logic designed to handle non-denoting logical values that can arise from terms that apply partial functions and operators.

[2] Where in this example we represent the non-denoting term as $\bot_{\mathbb{Z}}$ and the non-denoting logical value as $\bot_{\mathbb{B}}$.

[3] A brief history of LPF: Three-valued truth tables for the propositional operators are given in [12]; Peter Aczel supervised Koletsos' research [13] in which he gives a proof theory for such a logic; Cliff Jones suggested that Jen Cheng [4] apply this to programming tasks [1]; the typed version of LPF is covered in [10]. LPF is the logic that underlies the *Vienna Development Method (VDM)* [9, 2, 7].

LPF copes with undefinedness by accepting that where one (or both) operands of propositional operators fail to denote they will not yield one of the logical values ($true$ or $false$); the interpretation of quantifiers is extended in a compatible way. A shorthand for talking about this is to say that there is a third logical value: $undefined$ ($\bot_{\mathbb{B}}$), but –for reasons that become clear below– we prefer Blamey's notion of "gaps" in the value space [3]. From this point forward, $\bot_{\mathbb{B}}$ and $\bot_{\mathbb{Z}}$ are to be understood as ways of representing "gaps".

The truth tables in Fig. 1 (presented in [12]) illustrate the way in which the propositional operators in LPF have been extended to handle logical values which may fail to denote.

| $\vee$ | **true** | $\bot_{\mathbb{B}}$ | **false** |
|---|---|---|---|
| **true** | **true** | **true** | **true** |
| $\bot_{\mathbb{B}}$ | **true** | $\bot_{\mathbb{B}}$ | $\bot_{\mathbb{B}}$ |
| **false** | **true** | $\bot_{\mathbb{B}}$ | **false** |

| $\wedge$ | **true** | $\bot_{\mathbb{B}}$ | **false** |
|---|---|---|---|
| **true** | **true** | $\bot_{\mathbb{B}}$ | **false** |
| $\bot_{\mathbb{B}}$ | $\bot_{\mathbb{B}}$ | $\bot_{\mathbb{B}}$ | **false** |
| **false** | **false** | **false** | **false** |

| $\Rightarrow$ | **true** | $\bot_{\mathbb{B}}$ | **false** |
|---|---|---|---|
| **true** | **true** | $\bot_{\mathbb{B}}$ | **false** |
| $\bot_{\mathbb{B}}$ | **true** | $\bot_{\mathbb{B}}$ | $\bot_{\mathbb{B}}$ |
| **false** | **true** | **true** | **true** |

| $\neg$ | |
|---|---|
| **true** | **false** |
| $\bot_{\mathbb{B}}$ | $\bot_{\mathbb{B}}$ |
| **false** | **true** |

Fig. 1: The LPF truth tables for disjunction, conjunction, implication, and negation.

These truth tables provide the strongest possible extension of the familiar propositional operators and can be viewed as "describing a parallel lazy evaluation of the operands" [7]: a result is delivered as soon as possible, for example $true \vee \bot_{\mathbb{B}}$ evaluates to $true$ and this result will still be valid (there will be no contradiction) even if the second (non-denoting) operand later evaluates to $true$ or to $false$. This fits with our liking of Blamey's notion of undefinedness as "gaps". The issue of how to handle such "gaps" is central to the semantics given in Sect. 3.

It is also worthwhile noting that these propositional operators enjoy the familiar equivalences of classical logic; so, for example, $p \vee q$ has the same meaning as $q \vee p$, and $p \;\Rightarrow\; q$ has the same meaning as $\neg\, q \;\Rightarrow\; \neg\, p$. In addition the quantifiers of LPF are a natural extension (of the propositional operators) viewing existential quantification as a disjunction and universal quantification as a conjunction as is standard in FOPC. An existentially quantified expression in LPF is $true$ if a witness value exists even if the quantified expression is undefined (or $false$) for some of the bound values. Such an expression is $false$ if no witness value can be shown. This follows *mutatis mutandis* with universally quantified expressions.

Considering again the $zero$ partial function presented above, when $i < 0$ an application of this function can be thought of as denoting an undefined value of the appropriate type (say $\bot_{\mathbb{Z}}$) but we again prefer a notion of "gaps" in the value space.

In LPF the following property of the $zero$ function is $true$[4]:

$$\forall i \in \mathbb{Z} \cdot i \geq 0 \;\Rightarrow\; zero(i) = 0$$

[4] Of course, this simple property could be rewritten over a restricted set as: $\forall i \in \mathbb{N} \cdot zero(i) = 0$. This reflects one of the alternative approaches to handling, or avoiding, partial functions — but restricting types becomes messy for functions of more than one argument where the required "dependant type" is a predicate of all arguments [6]. Alternatively one may view the antecedent as a guard but in later examples this is not plausible.

it is less obvious how to provide a mechanical rewrite rule for the contrapositive of the first property above:

$$\forall i \in \mathbb{Z} \cdot \neg (zero(i) = 0) \ \Rightarrow \ i < 0$$

more fundamentally, there is no obvious type limitation (or guarding clause) that resolves:

$$\forall i \in \mathbb{Z} \cdot zero(i) = 0 \vee zero(-i) = 0$$

moreover, the following property is also $true$ in LPF:

$$\exists i \in \mathbb{Z} \cdot zero(i) = 0$$

in contrast, in FOPC, these properties fail to denote unless one finds some other way of avoiding the non-denoting Boolean values that arise.

Why is LPF not universally accepted? Clearly, there is a reluctance to adopt any non-classical logic. Specifically, one looks for those things that are unfamiliar in a non-standard logic. The only significant "surprise" in LPF is that the law of the excluded middle ($e \ \vee \ \neg \, e$) does not hold because the disjunction of two undefined Boolean values is still undefined — so $zero(-1) = 0 \vee \neg (zero(-1) = 0)$ is not considered to be a tautology.

For expressive completeness, a defined ($\delta$) operator has been introduced into LPF: $\delta(e)$ returns $true$ if $e$ is defined (it is $true$ or it is $false$). This gives LPF the deductive power of classical logic for defined expressions.

In fact, the most weighty argument against the adoption of LPF is the body of research and engineering that has created automatic tools for classical logic. This is precisely why the author of this paper is researching mechanised proof support tools for LPF.

## 3 Structural Operational Semantics (SOS)

The semantic formalisation approach that we use to provide the semantics for LPF is an SOS specification (introduced by Gordon Plotkin [17]). Our SOS specification provides an intuitive introduction to the semantics of LPF and how LPF addresses the issues of handling propositions that can include terms that fail to denote values. It is beneficial to provide such a formalisation as doing so allows us to be clear about the semantics of the logic before we begin with mechanising proof support tools for it. Additionally such a specification will provide a means of checking whether any mechanisation implemented is correct. Before we introduce the rules which formalises the semantics of LPF for expression evaluation we must first present the abstract syntax and the context conditions for the numerous expression constructs of our language.

### 3.1 Abstract Syntax

Our basic language includes numerous logical expression constructs which we introduce using abstract syntax [5]. Below, we use a type map (see Sect. 3.2) and a memory

[5] We use abstract syntax instead of concrete syntax to illustrate our expression constructs as our goal is to convey just the necessary information instead of worrying about how to write our expressions.

store (see Sect. 3.3) to provide our definition. In our language all expressions must be of the type BOOL or of the type INT. This restriction is made to be able to simplify the semantic rules that follow but at the same time even with just these two types we can adequately describe the issues encountered with non-denoting terms.

A constant value is itself an expression. Other expressions in our language include using a valid identifier, a relational (equality) expression, propositional logic expressions (disjunction and negation), a universal quantification expression[6], and a function call expression[7].

$$Expr = Value \mid Id \mid Equality \mid Or \mid Not \mid Forall \mid FuncCall$$

$$Value = \mathbb{B} \mid \mathbb{Z}$$

$$\begin{array}{lrcl} Equality & :: & a & : & Expr \\ & & b & : & Expr \end{array}$$

$$\begin{array}{lrcl} Or & :: & a & : & Expr \\ & & b & : & Expr \end{array}$$

$$Not :: a : Expr$$

$$\begin{array}{lrcl} Forall & :: & a & : & Id \\ & & b & : & Expr \end{array}$$

A function in our language always takes a (single) integer argument and returns an integer result[8]; a function definition thus contains a parameter name and a resulting expression (an expression — that might include recursive calls to the function). Such functions have no free variables; the free variables of their *result* expressions is only the *parm*eter.

$$\begin{array}{lrcl} Func & :: & param & : & Id \\ & & result & : & Expr \end{array}$$

A function call requires the name of the function and an argument to pass to the function.

$$\begin{array}{lrcl} FuncCall & :: & function & : & Id \\ & & arg & : & Expr \end{array}$$

[6] All expressions have to be explicitly closed by quantifiers. In addition we only consider quantification over integers for simplicity.

[7] Conjunction and implication follow in virtually the same way as disjunction, other relational operators follow in virtually the same way as the relational equality operator, not only for the abstract syntax but in a context condition and in our semantic rules that follow, and as a result we do not present them in this paper. The reader should also assume that arithmetic expressions, a defined judgement ($\delta$), a conditional expression (useful for recursive function definitions) and an existential quantification expression are also defined. How such expressions should be defined should become clear from how we define our other expression constructs.

[8] We have chosen to simplify the semantics by limiting functions to the one argument, and constraining the parameter type and the functions return type to integers — this is of course trivial to change.

For a function call we need to be able to access the called function in one of our context conditions and in our semantic rules. To do this we create a map from function names to function records.

$$\Gamma = Id \xrightarrow{m} Func$$

Where $\Gamma$ is the set of all possible functions, and $\gamma$ ($\gamma \subseteq \Gamma$) is used to represent a specific set of functions.

### 3.2 Context Conditions

In order to be able to perform type checks in our language we introduce a map called $Types$ that maps variable identifiers to the type of data that they can store.

$$Type = \text{BOOL} \mid \text{INT}$$

$$Types = Id \xrightarrow{m} Type$$

The context condition (presented in Fig. 2) returns the type of an expression if that expression is well-formed, otherwise the context condition will return ERROR. For example, a constant expression is well-formed and the type of that constant will be returned. An identifier is well-formed if it is in the domain of a given $Types$ map, and if so its type (INT or BOOL) will be returned; and so on for the rest of our expression constructs. We use this context condition to rule out ill-formed expressions, so that we do not have to attempt to give any semantics to ERROR expressions such as $mk\text{-}Or(true, 1)$. We only attempt to give semantics to those expressions that are well-formed and thus have an appropriate type.

We also provide a context condition so that we can rule out ill-formed function definitions from consideration in our semantic rules. The context condition presented in Fig. 3 returns $true$ if a given function is well-formed, otherwise it returns $false$. A function is considered well-formed if its result expression is of the same type as the expected result type (INT) of the function.

We intend that only the parameter of a function should be used as a variable within a function as illustrated in the context condition presented in Fig. 3, where the result expression must be checked with only the parameter as a variable in the given $Types$ map.

### 3.3 Rules

Having presented the syntax of our language and having ruled out ill-formed expressions and function definitions from further consideration, we can now move on to our primary task of defining the semantics of LPF, through an SOS specification.

All expressions in our language that reduce to a constant value are defined. Such values cannot be reduced any further. The constant values present in our language are the Boolean values $true$ and $false$, and the integers $(\cdots, -1, 0, 1, \cdots)$. If an expression can be evaluated to a member of one of these two sets then it is fully evaluated (no more evaluation can occur) and we refer to this as the evaluated expression denoting a value. For instance, the expression $0$ is denoting, the expression $mk\text{-}FuncCall(zero, 0)$

$$
\begin{array}{l}
\mathit{wf\text{-}Expr} : \mathit{Expr} \times \mathit{Types} \times \Gamma \to (\mathit{Type} \mid \textsc{Error}) \\
\mathit{wf\text{-}Expr}(e, \mathit{vars}, \gamma) \triangleq \\
\quad \textbf{cases } e \textbf{ of} \\
\quad\quad e \in \mathbb{B} \to \textsc{Bool} \\
\quad\quad e \in \mathbb{Z} \to \textsc{Int} \\
\quad\quad e \in \mathit{Id} \land e \in \textbf{dom } \mathit{vars} \to \mathit{vars}(e) \\
\quad\quad \mathit{mk\text{-}Equality}(a, b) \to \textbf{let } l = \mathit{wf\text{-}Expr}(a, \mathit{vars}, \gamma) \textbf{ in} \\
\quad\quad\quad\quad \textbf{if } l = \textsc{Int} \land l = \mathit{wf\text{-}Expr}(b, \mathit{vars}, \gamma) \\
\quad\quad\quad\quad \textbf{then } \textsc{Bool} \\
\quad\quad\quad\quad \textbf{else } \textsc{Error} \\
\quad\quad \mathit{mk\text{-}Or}(a, b) \to \textbf{let } l = \mathit{wf\text{-}Expr}(a, \mathit{vars}, \gamma) \textbf{ in} \\
\quad\quad\quad\quad \textbf{if } l = \textsc{Bool} \land l = \mathit{wf\text{-}Expr}(b, \mathit{vars}, \gamma) \\
\quad\quad\quad\quad \textbf{then } \textsc{Bool} \\
\quad\quad\quad\quad \textbf{else } \textsc{Error} \\
\quad\quad \mathit{mk\text{-}Not}(a) \to \textbf{if } \mathit{wf\text{-}Expr}(a, \mathit{vars}, \gamma) = \textsc{Bool} \\
\quad\quad\quad\quad \textbf{then } \textsc{Bool} \\
\quad\quad\quad\quad \textbf{else } \textsc{Error} \\
\quad\quad \mathit{mk\text{-}Forall}(a, b) \to \textbf{if } \mathit{wf\text{-}Expr}(b, \mathit{vars} \dagger \{a \mapsto \textsc{Int}\}, \gamma) = \textsc{Bool} \\
\quad\quad\quad\quad \textbf{then } \textsc{Bool} \\
\quad\quad\quad\quad \textbf{else } \textsc{Error} \\
\quad\quad \mathit{mk\text{-}FuncCall}(\mathit{id}, \mathit{arg}) \to \textbf{let } a = \mathit{wf\text{-}Expr}(\mathit{arg}, \mathit{vars}, \gamma) \textbf{ in} \\
\quad\quad\quad\quad \textbf{if } a = \textsc{Int} \land \mathit{id} \in \textbf{dom } \gamma \\
\quad\quad\quad\quad \textbf{then } \textsc{Int} \\
\quad\quad\quad\quad \textbf{else } \textsc{Error} \\
\quad \textbf{others } \textsc{Error} \\
\quad \textbf{end}
\end{array}
$$

Fig. 2: A context condition to check whether an expression is well-formed.

denotes $0$ and is thus denoting, but while the argument of $\mathit{mk\text{-}FuncCall}(\mathit{zero}, -1)$ is denoting such an expression can never denote a member of one of these two sets of values and thus this expression is not denoting — it is an undefined expression.

We also need to introduce a map that we refer to as a memory store which maps the variable identifiers to the values that they store at runtime.

$$\Sigma = \mathit{Id} \xrightarrow{m} \mathit{Value}$$

Thus $\Sigma$ is the set of all possible memory stores in our language and $\sigma$ ($\sigma \in \Sigma$) is used to represent a specific memory store.

Our SOS specification is presented as a series of inference rules which define the valid expression evaluations that can occur for the different expression constructs we have defined. The semantic relation that we use to model the process of expression evaluation is:

$$\xrightarrow{e} : \mathcal{P}((\mathit{Expr} \times \Sigma \times \Gamma) \times \mathit{Expr})$$

$$wf\text{-}Func : Func \times Types \times \Gamma \to \mathbb{B}$$

$$wf\text{-}Func(mk\text{-}Func(p, r), vars, \gamma) \triangleq wf\text{-}Expr(r, \{p \mapsto \text{INT}\}, \gamma) = \text{INT}$$

Fig. 3: A context condition to check whether a function is well-formed.

for example, if we evaluate an expression with a given memory store ($\sigma$) and function set ($\gamma$), the result is an expression. Where required, we use $\xrightarrow{e}*$ for the reflexive, transitive closure of $\xrightarrow{e}$. There is no way in our semantics to update $\Gamma$.

Our first semantic rule simply returns the value to which an identifier is mapped in a given memory store.

$$\boxed{Id\text{-}E}\ \frac{id \in Id}{(id, \sigma, \gamma) \xrightarrow{e} \sigma(id)}$$

Informally the $Id\text{-}E$ rule says that if the expression (in this case $id$) is an identifier, then the expression $id$ (with respect to a memory store $\sigma$ and a function set $\gamma$) can be evaluated to (replaced with) its value in $\sigma$.

The following set of semantic rules defines strict (weak) equality [8] which is defined to return a result only if both operands denote values; otherwise, the relational (equality) expression will fail to denote a value of the expected type.

$$\boxed{Equality\text{-}L}\ \frac{(a, \sigma, \gamma) \xrightarrow{e} a'}{(mk\text{-}Equality(a, b), \sigma, \gamma) \xrightarrow{e} mk\text{-}Equality(a', b)}$$

$$\boxed{Equality\text{-}R}\ \frac{(b, \sigma, \gamma) \xrightarrow{e} b'}{(mk\text{-}Equality(a, b), \sigma, \gamma) \xrightarrow{e} mk\text{-}Equality(a, b')}$$

$$\boxed{Equality\text{-}E}\ \frac{a \in \mathbb{Z}; b \in \mathbb{Z}}{(mk\text{-}Equality(a, b), \sigma, \gamma) \xrightarrow{e} [\![=]\!](a, b)}$$

Where $[\![op]\!]$ provides a semantic function for the syntactic object $op$ — thus $[\![op]\!]$ is the expected result from evaluating $op$ on its two operands.

Considering the $Equality\text{-}E$ semantic rule if both $a$ and $b$ are not fully reduced (evaluated) to integer values then a result from an equality expression in question will never be returned thus making the equality expression non-denoting. The reader should now notice how non-denoting terms that are operands to such strict relational operators can lead to non-denoting logical values.

The semantic rules that we present all represent a small-step semantics unless stated otherwise. The small-step semantics allow for interleaving of steps in different expression branches as can be seen from the rules for the relational equality operator. It is less important to have such interleaving for the strict relational equality operator as both operands must denote anyway, but it is important for logical operators such as disjunction since they have to cope with the "gaps" that can occur. This is because

in LPF we can return a result even in the presence of "gaps" in operands, as long as there is enough information available from evaluating the other operand. For example, $mk\text{-}Or(\bot_{\mathbb{B}}, true)$ can be evaluated to $true$ even though the first operand has not been fully evaluated[9]. Without having such interleaving considering the first operand of the previous example as containing a term that will never denote (e.g. arising from our function call expression construct such as $mk\text{-}Equality(mk\text{-}FuncCall(zero, -1), 0)$ — see later), without such interleaving being able to occur we may start to evaluate the first operand and with a big-step semantics we could not stop evaluation without evaluating this operand to a constant Boolean value (of which it will never denote one). The following set of semantic rules illustrates the evaluation of the disjunction logical operator according to the truth table presented in Fig. 1.

$$\boxed{Or\text{-}L}\ \frac{(a, \sigma, \gamma) \xrightarrow{e} a'}{(mk\text{-}Or(a, b), \sigma, \gamma) \xrightarrow{e} mk\text{-}Or(a', b)}$$

$$\boxed{Or\text{-}R}\ \frac{(b, \sigma, \gamma) \xrightarrow{e} b'}{(mk\text{-}Or(a, b), \sigma, \gamma) \xrightarrow{e} mk\text{-}Or(a, b')}$$

$$\boxed{Or\text{-}E1}\ \frac{}{(mk\text{-}Or(\textbf{true}, b), \sigma, \gamma) \xrightarrow{e} \textbf{true}}$$

$$\boxed{Or\text{-}E2}\ \frac{}{(mk\text{-}Or(a, \textbf{true}), \sigma, \gamma) \xrightarrow{e} \textbf{true}}$$

$$\boxed{Or\text{-}E3}\ \frac{}{(mk\text{-}Or(\textbf{false}, \textbf{false}), \sigma, \gamma) \xrightarrow{e} \textbf{false}}$$

The two rules $Or\text{-}E1$ and $Or\text{-}E2$ can be seen as "coping with gaps" in that they can return a value even if one of their operands fails to denote.

The choice of which rule is used is non-deterministic; there is no notion of fairness, so we have no control over which rule is used. Ideally we would like each operand to be evaluated in parallel and then have an elimination rule return a result once enough information is available from at least the one evaluated operand. Or alternatively to simulate this parallel evaluation by performing one evaluation step on the left hand operand and then one evaluation step on the right hand operand and then repeating this process until enough information is available to be able to apply an elimination rule (to complete the evaluation of a disjunction expression).

The fact that we have no control over when and what semantic rule is evaluated could be problematic. For example, considering the disjunction example from Sect. 1 where we set $i$ to the value 1. One may evaluate the left hand operand to $true$ and then try to evaluate the right hand operand continuously (with multiple applications of a rule) and this right hand operand does not denote (but the function can still be evaluated further — it will just not denote — see the function call semantic rules later). Thus an

[9] It could be that this operand could be fully evaluated or that this operand will fail to denote a value.

elimination rule may never be applied and thus the disjunction expression may never be evaluated to $true$, even though a result could be returned according to our understanding of LPF. Consider:

$$\begin{array}{l} mk\text{-}Or(mk\text{-}Equality(mk\text{-}FuncCall(zero, 1), 0), \\ \qquad\qquad\qquad\qquad mk\text{-}Equality(mk\text{-}FuncCall(zero, -1), 0)) \\ \rightarrow mk\text{-}Or(\textbf{true}, \bot_{\mathbb{B}}) \\ \rightarrow \textbf{true} \end{array}$$

clearly the expression is $true$ in LPF as one of the operands of the disjunction operator denotes $true$. However, with functions we do not know when or if functions will terminate generally (cf. Turing's Halting Problem). In this example there is more rewriting that could occur on the non-denoting operand, and thus we could always end up rewriting (in this case) the right hand (non-denoting) operand more. This is despite the fact that an elimination rule could be applied to complete the evaluation of this expression in such a situation.

The following set of semantic rules defines the evaluation of the negation logical operator.

$$\boxed{Not\text{-}A}\ \frac{(a, \sigma, \gamma) \xrightarrow{e} a'}{(mk\text{-}Not(a), \sigma, \gamma) \xrightarrow{e} mk\text{-}Not(a')}$$

$$\boxed{Not\text{-}E1}\ \frac{}{(mk\text{-}Not(\textbf{true}), \sigma, \gamma) \xrightarrow{e} \textbf{false}}$$

$$\boxed{Not\text{-}E2}\ \frac{}{(mk\text{-}Not(\textbf{false}), \sigma, \gamma) \xrightarrow{e} \textbf{true}}$$

We now consider the task of defining the rules for our universal quantification expression[10]. The following semantic rule states that if there exists an integer $i$ which when applied to the expression $e$ causes $e$ to evaluate to $false$ return $false$, even if the expression $e$ fails to denote with certain values of $i$. Clearly the choice of the value for $i$ is important. Notice that the quantifier can result in "gaps".

$$\boxed{Forall\text{-}F}\ \frac{\exists i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}, \gamma) \xrightarrow{e}* \textbf{false}}{(mk\text{-}Forall(t, e), \sigma, \gamma) \xrightarrow{e} \textbf{false}}$$

For the following semantic rule, it is necessary that for every integer substituted for $i$, $e$ must evaluate to $true$.

$$\boxed{Forall\text{-}T}\ \frac{\forall i \in \mathbb{Z} \cdot (e, \sigma \dagger \{t \mapsto i\}, \gamma) \xrightarrow{e}* \textbf{true}}{(mk\text{-}Forall(t, e), \sigma, \gamma) \xrightarrow{e} \textbf{true}}$$

We are using quantifiers to define the quantifier above and this is the core issue which we plan on addressing in future work. For now, think of the use of the existential

[10] Here we define the quantification rules using big-step semantics, whereas the rest of our semantic rules are defined using small-step semantics. The small-step semantics allow for interleaving of the steps in different expression branches.

quantifier above the line as shorthand for an infinite disjunction (using the LPF disjunction logical operator already introduced), and the use of the universal quantifier above the line as shorthand for an infinite conjunction (both over the set of integers).

We now move onto the task of showing a way of introducing non-denoting terms ("gaps") in the first place, which is done through the use of our function call expression construct. The $FuncCall\text{-}A$ semantic rule represents the small-step semantics for evaluating the argument expression to be passed to a function. This rule is to be utilised until the argument expression has been reduced to a constant value.

$$\boxed{FuncCall\text{-}A}\ \frac{(arg, \sigma, \gamma) \xrightarrow{e} arg'}{(mk\text{-}FuncCall(id, arg), \sigma, \gamma) \xrightarrow{e} mk\text{-}FuncCall(id, arg')}$$

Any argument used in a function call must denote, otherwise we do not evaluate the function. Once (if) the argument expression has been reduced to a constant value we can now attempt to evaluate the functions result expression.

$$\boxed{FuncCall\text{-}E}\ \frac{arg \in \mathbb{Z}}{\begin{array}{c}(mk\text{-}FuncCall(id, arg), \sigma, \gamma) \xrightarrow{e} \\ mk\text{-}FuncInter(\gamma(id).result, \gamma(id).param, arg)\end{array}}$$

Here we make use of another expression construct that combines the necessary information from a function call expression and from function being called itself. The new expression construct (which represents a function call under evaluation) contains the result expression from the function (e.g. a conditional expression) as well as the functions parameter identifier and the value passed into the function. This expression construct is used to allow for the current state of the result to be stored (alongside the parameter data) so that the evaluation can resume from where it left off previously if any interleaving of the steps in expression branches occurs.

$$Expr = \ldots \mid FuncInter$$

$$\begin{array}{lll} FuncInter :: & result & : Expr \\ & paramid & : Id \\ & param & : Expr \end{array}$$

The following two semantic rules define the rest of the small-step semantics for evaluating a function call expression. The first semantic rule is used to make a further step in evaluating the result of the function each time it is applied[11]. The second semantic rule returns the result of the function call expression once (if) the functions result expression has been evaluated to an integer value.

$$\boxed{FuncInter\text{-}A}\ \frac{(res, \sigma \dagger \{paramid \mapsto param\}, \gamma) \xrightarrow{e} res'}{\begin{array}{c}(mk\text{-}FuncInter(res, paramid, param), \sigma, \gamma) \xrightarrow{e} \\ mk\text{-}FuncInter(res', paramid, param)\end{array}}$$

[11] The parameter is included in the memory store during an evaluation step of the functions result, but notice that the updated memory store is not returned by the semantic rule. Only the updated result expression along with the parameter information to (possibly) be used to update the memory store in the same way later is returned by this semantic rule. This is to achieve the necessary variable scoping since interleaving of steps in different expression branches is allowed and is necessary for LPF.

$$\boxed{\mathit{FuncInter\text{-}E}}\ \frac{res \in \mathbb{Z}}{(\mathit{mk\text{-}FuncInter}(res, paramid, param), \sigma, \gamma) \xrightarrow{e} res}$$

## 4 Conclusions

Over the course of this paper we have formalised the semantics of LPF for the evaluation of numerous expression constructs using an SOS specification. One of the reasons for carrying out this work was to provide a formalisation of a non-classical logic notably LPF in particular to formalise how LPF copes with propositions that may contain potentially non-denoting terms (where "gaps" can be present in the value space), for instance from the application of partial (recursive) functions and operators. Our formalisations were created with the full intent of in the future mechanising support for proofs about such propositions using the logic LPF. The formalisations that we have provided in this document not only allow us to be more confident that we fully understand the semantics of LPF before we begin with a mechanisation, but they also provide a means of checking whether any mechanisation of LPF which we come up with is correct.

As in most cases there is much more work to be done. Our SOS specification has a problem when addressing the quantifiers in that we are using quantifiers to define quantifiers. This is not acceptable because if the meta-language interpretation of the quantifiers changes then so does the implied semantics. We plan on addressing this shortly by carrying our SOS definition over to a denotational semantics by providing a set theoretic definition of the values that are denoted by expressions. We will then finish off this piece of work by showing how such a denotational semantics definition can be used as a basis for proofs about propositions over terms that can fail to denote.

The major task ahead of us is mechanising support for proofs in LPF. This can be broken down into two separate subclasses of problems for further research. The first is to research how fundamental techniques such as *unification* and *resolution* work with such a non-classical logic. The second activity relates to actually implementing mechanised tool support for proofs about propositions over terms that can fail to denote values.

Since non-denoting terms arise frequently (e.g. from the application of partial functions and operators) in program specifications [6, 11, 7] which raises the question over how proofs about such terms can be conducted formally, and since a large body of research and engineering has created tools for classical logic, as well as approaches to coping with non-denoting terms having attracted much research over the years (e.g. [15, 3, 1, 16, 4, 6, 11, 14, 7]), we feel further research on mechanised proof support tools for LPF will be of significant benefit.

## Acknowledgements

The content of this paper has benefited greatly from discussions with Cliff Jones, Jason Steggles, and Joey Coleman. The author of this paper also gratefully acknowledges the funding for his research from an EPSRC PhD Studentship.

## References


1. H. Barringer, J.H. Cheng, and C. B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.
2. Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.
3. S. R. Blamey. *Partial Valued Logic*. PhD thesis, Oxford University, 1980.
4. J. H. Cheng. *A Logic for Partial Functions*. PhD thesis, University of Manchester, 1986.
5. J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. Technical Report UMCS-90-3-1, Manchester University, February 1990. Preprint of [6].
6. J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. C. P. Woodcock, editors, *3rd Refinement Workshop*, pages 51–69. Springer-Verlag, 1991.
7. J. S. Fitzgerald. The Typed Logic of Partial Functions and the Vienna Development Method. In D. Bjørner and M. C. Henson, editors, *Logics of Specification Languages*, EATCS Texts in Theoretical Computer Science, pages 427–461. Springer, 2007.
8. J. S. Fitzgerald and C. B. Jones. The connection between two ways of reasoning about partial functions. *IPL*, 107(3–4):128–132, 2008.
9. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.
10. C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.
11. Cliff B. Jones. Reasoning about partial functions in the formal development of programs. In *Proceedings of AVoCS'05*, volume 145, pages 3–25. Elsevier, Electronic Notes in Theoretical Computer Science, 2006.
12. S. C. Kleene. *Introduction to Metamathematics*. Van Nostrad, 1952.
13. G. Koletsos. Sequent calculus and partial logic. Master's thesis, Manchester University, 1976.
14. Alexander Krauss. Partial recursive functions in higher-order logic. In *Int. Joint Conference on Automated Reasoning (IJCAR 2006), LNCS*, pages 589–603. Springer-Verlag, 2006.
15. J. McCarthy. A basis for a mathematical theory for computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland Publishing Company, 1967.
16. O. Owe. An approach to program reasoning based on a first order logic for partial functions. Technical Report 89, Institute of Informatics, University of Oslo, February 1985.
17. G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

# Towards Dynamic Reconfiguration of Distributed Systems in VDM-RT

Claus Ballegård Nielsen

Aarhus School of Engineering, Denmark
clausbn@cs.au.dk

**Abstract.** The VDM-RT dialect enables the modeling and validation of distributed embedded real-time systems by expressing the distributed architecture as CPUs connected by busses. A limitation of current VDM-RT is that the deployment of systems on CPUs as well as the network topology between CPUs is static during execution. This is a constraint when modeling types of distributed computing systems that have high demands for scalability and adaptability in environments characterized by constant change. These demands can be met by flexible design with a dynamic behavior in which the ability to adapt to system modification and extensions is made possible through dynamic reconfiguration and migration mechanisms. This paper presents the initial results of a study in extending VDM-RT to enable dynamic reconfiguration during the runtime execution of a model. New language constructs are introduced for expressing dynamic reconfiguration of the network topology as well as object migration to new CPUs.

## 1 Introduction

Many embedded distributed systems are designed assuming a static set of hardware and software configurations. However newer kinds of computing systems have high demands for adaptability, scalability and availability made possible through flexible design. These systems either have quality of service as a principal requirement, which entail recovery and failover safety measures that requires alternation of system architecture and data migration on running systems, or they have an autonomic [3] or ubiquitous [8] behavior in which system adaption to environment changes is a natural part of the system functionality. These demands can be met by enabling dynamic behavior in the system architecture, meaning that the system has the ability to adapt to system modification and extensions through system reconfiguration and object migration mechanisms. In the context of this paper system reconfiguration denotes changing the configuration of the network topology during runtime execution. While object migration denotes the act of moving an object from one executable unit to another during execution while still maintaining the internal state of the object. The term dynamic reconfiguration will denote both system reconfiguration and object migration.

In the development of embedded systems formal methods has been used to model and analyze the system requirements in reference to the system design. Formal technique such as VDM-RT provides exploration and validation of system design and behavior as well as performance analysis by enabling the creation of an executable formal model of a specific distributed real-time system [6, 5, 2]. However in VDM-RT both the

deployment of objects on CPUs and the network topology between CPUs is preconfigured in the formal model and stays static throughout the execution of the model. This makes it difficult to model systems that require a dynamic behavior, whether it being for QoS demands or because of the system nature, e.g. ubiquitous computing.

This paper has its emphasis on introducing new language constructs to VDM-RT for expressing the system reconfiguration and object migration. This involves the creation of language constructs which may require semantics adjustments of VDM-RT. Such an alternation is necessary to provide a way of expressing the dynamic changes during run-time execution.

The paper is organized as follows; the new language constructs and the semantics for the dynamic reconfiguration extension is explained in section 2. The proposed extension will be evaluated through the application to a case study in section 3. Finally concluding remarks and future work is presented in section 4.

## 2 A Dynamic Reconfiguration Extension of VDM-RT

The objective is to introduce the dynamic reconfiguration extension into VDM-RT with as few changes to the VDM-RT language dialect as possible. An approach for expressing reconfiguration in a precise and efficient manner is to be established, while still keeping the language structure and basic philosophy of VDM-RT intact.

The intent of the extension is to provide the means for creating high-level and abstract models of complex reconfigurable distributed systems. The goal is not to deliver completeness or proof of an entire system functionality or system requirements fulfillment, but to aid system design decisions, create an overview and identify potential functionality and design flaws in the early development phases. The analysis and evaluation of these system properties has its foundation in the simulation of the system through the executable VDM-RT model.

In VDM-RT the distributed system architecture is formed on the basis of two predefined classes BUS and CPU, which represents a communication line and a processing unit respectively. The distributed system itself is defined in the special **`system`** class, in which the architecture is constructed by creating relations between instances of the predefined classes and normal VDM objects. The architecture is laid out in the following way:

- A CPU is connected to another CPU over a BUS,
- the different distributed object are connected through object references and
- finally these objects are deployed to specific CPUs.

During the execution of the model the integrity of the communication between the distributed objects can be validated by comparing the communication (operation calls) processing over object references with the BUS connection between CPUs. If an object deployed on one CPU communicates with an object deployed to another CPU, then the CPUs in question must be connected to the same BUS connection. Meaning that although objects are connected by object references there must be a BUS connection as well to enable communication. This is illustrated in fig. 1 where Obj1 will not be able to communicate with Obj3, because of the missing BUS connection, although they are

connected by object references. Communication is point-to-point and there is no possibility of broadcasting with VDM-RT.

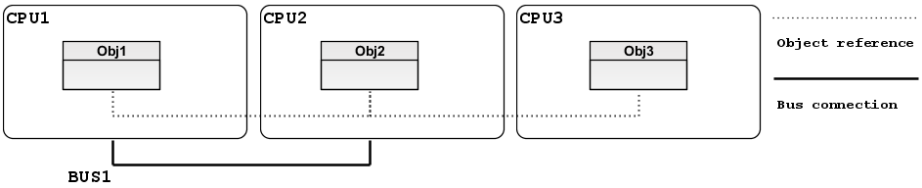


Fig. 1: Basics of VDM-RT

When looking into an approach for describing system reconfiguration this construct of double layers of references becomes interesting. The way the distributed architecture is constructed requires the modeler to have knowledge of the CPU and object relationship, which makes sense in a static system where CPU and objects are tightly connected. However when modeling a more dynamic system architecture it might be just as relevant to enable the description of object to object interconnection without requiring knowledge of the association between an object and a specific CPU. This approach will provide a layer of transparency to the remote communication, because object references do not need to know how communication is conducted on the BUS. In this way the modeler can to some extent look past the CPUs and concentrate on object to object interaction.

In context of system reconfiguration this means that if a language extension was made for connecting or disconnecting CPUs from a BUS the modeler would need to have deep insight into the internals of the system class, as well as keep track of which objects is deployed on which CPU in order to connect objects later on. Instead the proposed extension will be able to describe system reconfiguration at the object to object level.

Given that the distributed system architecture is defined in the special **`system`** class and all BUS and CPU declarations is contained within this class, it will be used as the heart of the dynamic reconfiguration. Consequently the **`system`** can be considered a type of network manager [7], but a specific network manager class is not created. When having the system class a manager class would just add an extra layer.

The changes are implicitly added to the three classes (**`system`**, BUS, CPU) as public operations, meaning that with a reference to an instance of e.g. the **`system`** class it will be possible to request reconfiguration. This has the advantage that the reconfiguration can be performed from wherever the need of reconfiguration arise, consequently the disadvantage of this decentralization is the risk of losing the overview.

The proposed additional operations are presented in table 1, where Sys represents an instance of a **`system`** class and BusX and CpuX denotes X instance of the respective classes.

| Operation | Parameter | Description |
|---|---|---|
| Sys.ConnectToBus(obj , bus) | Object $*$ BUS | Connect object with a BUS. |
| Sys.DisconnectFromBus(obj , bus, idle) | Object $*$ BUS $*$ bool | Disconnect object from BUS. |
| Sys.Migrate(obj, cpu, idle) | Object $*$ CPU $*$ bool | Migrate object to a specific cpu. |
| Sys.ActivityOnBus(obj, bus) | Object $*$ BUS | Determines the objects current BUS activity |
| Sys.IsConnected(obj, obj) | Object $*$ Object | Are objects connected by a BUS |
| BusX.IsConnected(obj) | Object | Is the object connected to BusX. |
| CpuX.IsDeployed(obj) | Object | Is the object deployed to CpuX |

**Table 1:** Dynamic reconfiguration operations

The extension proposed does not introduce any new classes or language keywords into VDM-RT since the dynamic reconfiguration is enabled by adding new operations to existing predefined classes. Combined with the existing VDM-RT semantics this will prevent the risk of ambiguity and keyword conflicts with existing VDM models, because in the current VDM-RT semantics the BUS and CPU can only be instantiated by the modeler, but not extended or inherited. The same applies for the **`system`** class, although it is defined by the modeler the implementation is limited by the syntax checker to only allow the definition of the constructor. This ensures backward compatibility as conflicts with any operations defined in existing models do not need to be considered, given that they cannot exist in any validated model.

### 2.1 ConnectToBus - Establishing a New Connection

The ConnectToBus operation requires the object reference to an object as well as to a BUS in order to connect the object to the specified BUS. The ConnectToBus operation is used to establish communication between an object and other objects which are already connected to the given BUS. The reconfiguration will occur as an atomic operation, meaning that it it will be uninterruptible as it appears to the rest of the system to take no time.

As the operation merely establishes a connection and does not alter existing connections there is no need to take care of reference changes nor current communication of the concerned objects. Meanwhile as connections in the VDM-RT semantics are made between CPUs the operation will have to resolve an object reference to a specific CPU. Internally this means that ConnectToBus will lookup the CPU on which the object is deployed and then establish the connection with the BUS. Once the ConnectToBus operation is called the reconfiguration is to be considered successful. If the connection already exists nothing will be changed.

When a ConnectToBus operation is completed no event or other type of notification will be supplied implicitly to the concerned objects to give awareness of the reconfiguration change. Instead it is left for the modeler to ensure that the distributed objects are aware of new connections if this information is needed by the remote objects.

It should be emphasized that the operation only creates a connection from a CPU to a BUS, it is up to the modeler to establish the reference between objects. This is illus-

trated in fig. 2 where an object reference between Obj1 and Obj3 exists, but the BUS connection is needed to establish communication. Listing 1 demonstrates how this connection can be created by using the object reference.

```
--connect obj 1 to BUS 1
Sys.ConnectToBus(obj1, Sys‘bus1);
```

Listing 1: Create a new connection

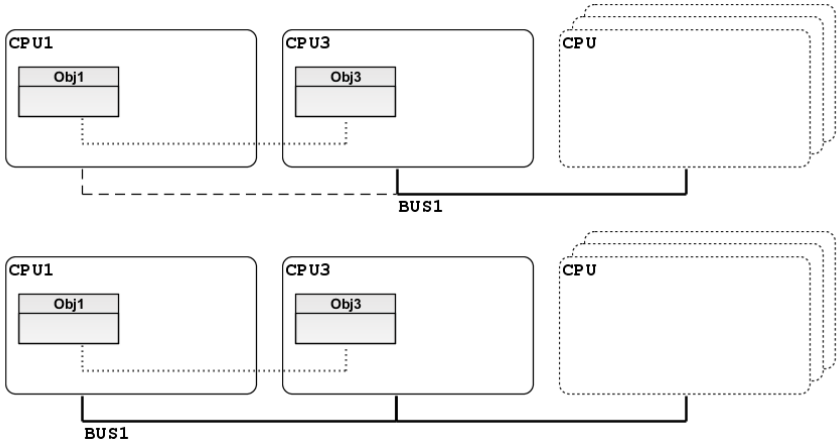


Fig. 2: System Reconfiguration

### 2.2 DisconnectFromBus - Tearing Down a Connection

The DisconnectFromBus operation is used to tear down an existing connection between an object and a BUS. Disconnecting an active entity from its network is a disruptive action. The reconfiguration will happen in an atomic way and there will be no graceful disconnection or notification to the influenced objects. It is the task of the modeler to ensure that any object which may be using the connection is notified prior to the reconfiguration.

When DisconnectFromBus is called the supplied object will be disconnected from the supplied BUS and all messages of current requests and replies will be dropped from the network, without error. Correspondingly any attempt to initiate communication with the disconnected object over the involved BUS, subsequent to the time of reconfiguration, will cause a runtime-error.

```
Sys.DisconnectFromBus(obj1, Sys‘bus2, false);
```

Listing 2: Tearing down a connection

The operation call is shown in listing 2. The third parameter of the operation is a boolean value that indicates if the BUS should be idle before the disconnection occurs, which allows for more gentle disconnection. If a true value is passed to the function it will implicitly call the ActivityOnBus operation (explained in section 2.4) and the operation will block until the bus becomes idle. To ensure that the messages on the BUS will be processed the idle check is done before the operation enters the atomic state. Given that the operation blocks until the BUS is idle there is a potential risk that it may never unblock due to constant BUS activity. This will indicate a scenario where the communication is so intense that a graceful disconnect is not possible.

It should pointed out that although DisconnectFromBus is called with an object reference, the entire CPU that the given object is deployed on, will be disconnected from the BUS. This means that the modeler must be aware that because connection and disconnection occurs on a CPU to BUS basis, a system reconfiguration will affect the connectivity of all objects deployed on the CPU in question.

### 2.3 Migrate - Moving Objects

Migrate moves an object to a new CPU as an atomic operation while preserving its identity and state. When moving an object to a new CPU referential integrity is a concern because all existing object references to and from the object must be managed. When moving an object the existing references this object has to other objects is a dilemma, because what is considered to be part of the object and how much should be moved?

In this extension the following rule will be used: All transitive references will be included as part of the object, meaning that all objects that have been created by the migrating object and still referenced from it will be moved on migration as well as all objects created by these referenced objects, etc.

Concerning other objects' references to the migrating object, the following applies; as deployment on to CPUs is to be considered more of a virtual association [1] than a physical change, the internal reference of the object will remain the same even when moved to a new CPU. This means that existing object references to the migrated object will still be valid after the reconfiguration. Although some operation calls may now occur over a network, this approach will ensure that referential integrity is kept, there are however exceptions which is why the following rule is created. The target CPU, to which the object is migrated, must be connected to the same BUS as the source CPU from which the object is migrated. This is done to ensure a connection over which the object can be transferred and at the same time it has the benefit that the objects current requests and replies with other objects will still be able to process over the BUS. The exception occurs when the migrating object has communication with remote objects through a different BUS than the BUS on which the target and source CPU is connected.

This is shown in fig. 3 where Obj1 is able to communicate remotely with both Obj2 and Obj3 through different buses. After the migration, Obj1 and Obj2 can communicate internally, but communication with Obj3 is no longer possible. In fact, any attempt of communicating between Obj1 and Obj3 after the point of reconfiguration will give a runtime-error. The Migrate operation does nothing to prevent this type of error because it will require the creation of new BUS connections automatically and because it is

considered to be the responsibility of the modeler to ensure that communication is still possible following reconfiguration. If VDM-RT automatically ensured communication following a reconfiguration it will prevent the modeling of systems with a focus on network integrity. However it does apply that any message on the BUS that were send prior to reconfiguration will still be processed after the reconfiguration, if possible by the BUS connection. If a message send prior to reconfiguration is unable to reach its destination because of a missing BUS connection it will be lost, but it will not cause a runtime-error.

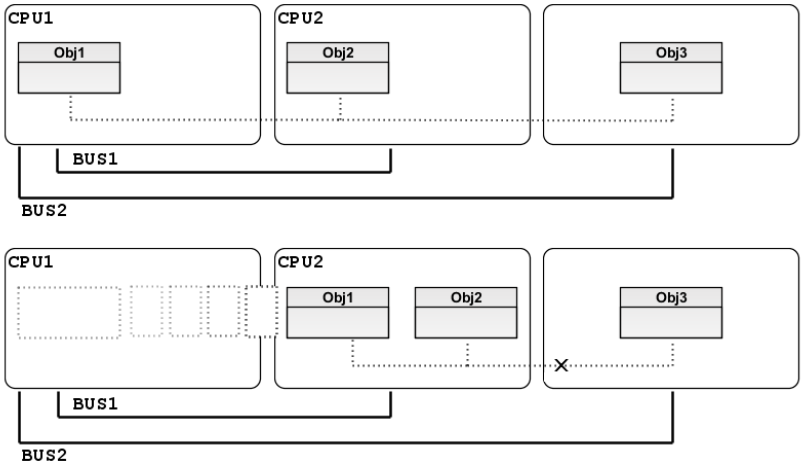


Fig. 3: Object migration

Listing 3 shows the migration of obj1 to cpu2. The third parameter of the operation is a boolean value that indicates if the BUS should be idle before the migration should occurs. The details of this parameter is explained in section 2.2

```
Sys.Migrate(obj1, cpu2, true);  -- migrate obj 1 to cpu 2
                                -- once BUS becomes idle
```

Listing 3: Migrating an object to another cpu

### 2.4 ActivityOnBus - Messages on the Bus

The introduction of dynamic reconfiguration increase the need for knowledge of the current activity an object has on a bus. Because a change in network topology or an object migration can result in the loss of message data currently in transmission on a BUS, providing more information on current remote operation invocations will benefit the modeler in describing the changes.

The ActivityOnBus operation takes an object and a bus as its parameters and returns an enumerated type depicting the current activity the given object has on the given BUS.

The enumeration returned is defined as follows.

**Idle:**
The object has no communication on the bus.
**ProcessingRequest:**
The object is currently processing an operation invocation by a remote object.
**WaitingForReply:**
The object is waiting on a reply from an operation invocation on a remote object.
**RequestAndReply:**
The object is both processing a request and waiting for a reply.

The different indications supplied by the enumeration will allow the modeler to use different strategies for reconfiguration based on the current state of the object.

### 2.5 IsConnected and IsDeployed - Determining Status

The IsConnected and IsDeployed operations are added to the **`system`**/BUS and CPU classes respectively to provide information of the current topology and object deployment. This is done from a management perspective and to aid the modeler in establishing an overview of the current system configuration.

The IsConnected operation in the **`system`** class evaluates if two objects are connected, the IsConnected operation on the BUS class assess if a specific object is connected to the BUS on which the operation is called, while the IsDeployed operation on the CPU class determines if an object is deployed on this particular CPU.

## 3 Evaluation of the Extension Applied to a Case Study

The case study revolves around a Vehicle Monitoring system designed to improve road safety by having vehicles in near proximity form a co-operative wireless network through which information can be shared. The autonomous movement of vehicles mean that the system is highly based on distributed computing and has a high exchange of network relations with constant connections and disconnections. A more detailed explanation can be found at `http://www.cs.au.dk/~lausbn/vemo/`

The case study model contains a controller that has knowledge of all vehicles and it is therefore able to calculate when vehicles are in range of each other. The controller manages when vehicle are allowed to communicate. The issue with the existing model is that, because VDM-RT has a static network topology, the vehicles are constantly connected to the BUS. This can lead to problems because when developing the model an unintentional implementation might be created in which vehicles start to initiate communicate directly while disregarding the controller. Potentially creating scenarios in which vehicles are able to communicate even though they are too far apart to actually establish a network. This is possible because the model is not benefitting from the interpreter check of BUS and object connectivity because of the static network layout.

With a dynamic reconfigurable model the topology can change, meaning the vehicles can be connected and disconnected as they move around which means that a closer

representation of the real life scenario is created. By using dynamic reconfiguration functionality the VDM-RT interpreter check on BUS to object connectivity is able to ensure the correctness of the communication and will discover any communication disregarding the controller.

A challenge arises when modeling systems which have a large number of units with autonomous behavior that use wireless connections to communicate. They contain an unpredictable number of network connections which they may not be able to share as the simulated wireless network will not be able to reach. This is problematic when the number of BUS connections available is static. There are two possible solutions to this. The first is creating a pool of BUS connections from which a BUS can be borrowed to simulate a wireless network and then returned when there is no longer any in range of the network. This can be done with the current version of VDM-RT, but is involves the risk of exhausting the pool. The other approach would be to add the dynamic creation of buses in VDM-RT as this would enable a more realistic representation of wireless networks. This is however not a part of the extension presented in this paper.

Listing 4 shows the use of the reconfiguration extension in the calculation of vehicles in range which is contained in the controller class. Because the connection is only established when the vehicles are in range of each other the model becomes more precise and we gain a greater confidence in the system being modeled.

```
[...for all vehicles other vehicles in range are placed in
inrange...]
  --find new vehicles that has come in range,
  -- create BUS connections.
  let newConnections =  inrange \ currentConnections(unit)
  in VeMo.ConnectToBus(newConnections, VeMo`bus1);

  -- find vehicles that has gone out of range
  -- tear down the BUS connections.
  let lostConnections = currentConnections(unit) \ inrange
  in VeMo.DisconnectFromBus(lostConnections, VeMo`bus1);

[...Start data communication ...]
```

Listing 4: Use of the reconfiguration extension in the Controller class

## 4 Concluding Remarks and Future Work

In this paper a VDM-RT language extension as well as its semantics have been presented. We propose an extension of VDM-RT which enables the modeling of dynamic reconfiguration of distributed systems. The proposed extensions makes it possible to change the network topology of a system and migrate objects between CPUs during the runtime execution of a model. The language extensions will allow for experiments with deployment strategies and validation of connectivity between entities in a system

following reconfiguration. This will enable VDM-RT to model systems which are dynamic by nature such as autonomous systems.

Future work include adding tool support for the extension by integration into a branch of the Overture platform [4], as well as a further investigation into case studies in order to determine if there is a necessity for a special construct, such as a policy, for managing the dynamic reconfiguration or it is sufficient to managed it directly from controller objects in the model.

As this paper presents the initial phase of introducing dynamic reconfiguration into VDM-RT there are still elements which are undetermined. A process for representing messages being lost on the BUS, because of a reconfiguration, needs to be established as this will be a matter for synchronous operations that waits for at return value. Multiple approaches have been considered; exception handling could be introduced on the communication level so that an exception could be thrown on message loss, or some special type could be returned by an unsuccessful remote operation call. Furthermore the possibility of creating new instances of the BUS class dynamically needs to be investigated in further detail.

## Acknowledgments

The author would like to thank the Overture community for input and thoughts about the content of this paper. Special thanks go to John Fitzgerald and Anirban Bhattacharyya for their valuable comments and feedback.

## References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Fitzgerald, J.S., Larsen, P.G., Tjell, S., Verhoef, M.: Validation Support for Real-Time Embedded Systems in VDM++. Tech. Rep. CS-TR-1017, School of Computing Science, Newcastle University (April 2007), Revised version in Proc. 10th IEEE High Assurance Systems Engineering Symposium, November, 2007, Dallas, Texas, IEEE
3. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. IBM Systems Journal 42(1), 5–18 (2003)
4. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
5. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2008), ISBN 978-90-9023705-3
6. Verhoef, M., Larsen, P.G.: Enhancing VDM++ for Modeling Distributed Embedded Real-time Systems. Tech. Rep. (to appear), Radboud University Nijmegen (March 2006), a preliminary version of this report is available on-line at *http://www.cs.ru.nl/∼marcelv/vdm/*
7. Wegdam, M.: Dynamic Reconfiguration and Load Distribution in Component Middleware. Ph.D. thesis, University of Twente (2003), ISBN 90-75176-36-8
8. Weiser, M.: The computer for the twenty-first century. Scientific American 265(3), 10 (September 1991)

# Overview of VDM-RT Constructs and Semantic Issues

Kenneth Lausdahl[1] and Marcel Verhoef[2] and Peter Gorm Larsen[1] and Sune Wolff[3,1]

[1] Aarhus School of Engineering, Dalgas Avenue 2, DK-8000 Aarhus C, Denmark
[2] Chess, PO Box 5021, 2000 CA Haarlem, The Netherlands
[3] Terma A/S, Hovmarken 4, DK-8520 Lystrup, Denmark

**Abstract.** An inventory of current semantic models of the Vienna Development Method (VDM) notations are presented, for which their purpose, strengths and weaknesses are assessed. The focus will be on VDM Real Time (VDM-RT) with (multi-threading and multi-core) concurrency, communication and real-time. Areas are identified where the semantics is currently unclear, incomplete or even undefined. Challenges in adopting novel language concepts are investigated, for example for modelling uncertainty in real-time distributed systems. Approaches taken by other formalisms are presented and suggestions are offered how these ideas could be applied in the context of VDM-RT. The result of this work aims to become a roadmap for the definition of a full semantics of VDM-RT, on the short term focused on symbolic execution (simulation), but needs to be amenable to formal proof and exhaustive search (model checking) in the future.

## 1 Introduction

The Vienna Development Method (VDM) was first conceived at the IBM laboratories in Vienna in the early seventies [7, 4]. Later on different dialects of VDM evolved at different places in the world and BSI and subsequently ISO standardised a version of VDM called VDM-SL [10, 11]. As the object-oriented paradigm gained popularity, an object-oriented extension called VDM++ was developed in the European research project called Afrodite [2]. In order to provide better facilities for modelling real-time embedded and distributed systems a VDM Real Time (VDM-RT) extension was developed [8, 15].

The initial version of VDM-RT was developed in a European research project called VDM In Constraint Environments (VICE) but this was developed for a single CPU [8]. At the beginning of the PhD work by Verhoef this VICE version was used to attempt to model a distributed real-time embedded system [16]. This motivated extensions of VDM-RT to incorporate constructs enabling the formulation of distributed systems and as a consequence it was possible to model this case study in a much better fashion.

The semantics made for VDM-SL are very strong, and gives a solid foundation for the language. These semantics are described using the denotational semantics style. Less work has been put into the semantics for VDM++ and VDM-RT, and hence more issues are present. It is some of these issues for VDM-RT that we will describe in this paper, and present possible solutions.

After this introduction, Section 2 provides an introduction to the constructs from VDM-RT that are extensions compared to the VDM++ notation. Afterwards Section 3

provides a list of semantics issues in the current version of VDM-RT that needs to be solved. This is followed by Section 4 which provides a list of the corresponding issues in relation to the co-simulation that is developed as a part of the DESTECS project. At the end of the article, Section 5 provides an overview of the future work in this area and finally Section 6 gives a few concluding remarks about the practical plan for producing the necessary semantic definitions.

## 2 Constructs in VDM Real-Time

In VDM++ models are structured in classes and inside these one can define constant values, instance variables (the state of instances of a class), functions with expressions as their body and operations which have statements as body and can access and modify instance variables that are visible inside a class. In addition a class can have a thread and synchronisation of threads is controlled using permission predicates that are logical expressions describing the requirements for activation of a particular operation. VDM-RT is an extension to VDM++ and from here it is important to note that static operations can be defined inside VDM++ classes. In VDM-RT it is furthermore possible to make use of asynchronous operations in VDM-RT using the **`async`** keyword. This will spawn a new thread that will be executed concurrently with the calling thread.

In VDM-RT a special **`system`** class administers the static topology of components inside the system. This includes the ability to distribute instances of classes (objects) to a special type of predefined `CPU` class. For each `CPU` the designer can specify the scheduling mechanism as well as the capacity described as number of computations per time unit. In order to enable communication between active threads in the system, `CPU`s can be connected by `BUS`es. For each individual `BUS` the designer can specify the bandwidth as well as which `CPU`s are connected by the `BUS`. All instances created inside a deployed instance will be residing on (deployed to) the same `CPU`. A special virtual `CPU` and a special virtual `BUS` are implicitly created. The virtual `CPU` will be used to all instances that are not explicitly deployed to a `CPU` declared in the system class. This is used for the things that are outside the system (i.e. the environment classes and the debugging desired by a user).

In VDM-RT, there is a notion of time but per default no unit of time is assumed so that is up to the user. However, the convention most frequently used is that the time unit is milliseconds. In order to access the current global time, the keyword **`time`** is used. All VDM constructs have been assigned a default duration and thus the semantics include a time penalty. If the user can give estimates of fixed execution times, this can be described using the **`duration`** statement which then overrule the default durations of the body statements. If, instead, the execution time is relative to the speed of the `CPU` on which the thread is deployed, the **`cycles`** statement can be used which again overrule the default durations. In the case of nested durations and cycles these are also overruled by the outermost duration/cycles statement. The virtual `CPU` is special in the way that here all default durations are set to zero in order for the timing influence of the elements outside the system in a simulation having minimal impact (unless the user wishes that to happen by inserting duration statements).

Semantically, a multi-cpu VDM-RT model can either conduct an execution step or progress the global time (for all resources). So conceptually speaking there is a master scheduler that allows all `CPU`s to progress until they need to take a time step. When all the `CPU` schedulers have done this they report back how large a time step they would like to take. For `BUS`es, the analogy is whenever the next message is due to be delivered at a receiving `CPU`. The master scheduler then takes the smallest of these time steps and tell all `CPU` schedulers to advance with this time and execute again if they have no remaining time to wait. The semantics provided in [14] on purpose does not include anything for the scheduling, since in essence, it corresponds to a reduction of the possible interleavings of the concurrent threads executing.

Finally it is possible to specify periodic threads using a 4-tuple $(p, j, d, o)$: $p$ describes the period; $j$ is the jitter, $d$ is the minimum time between invocations of a periodic operation and $o$ is the initial offset (see Fig. 1). Note that this syntax does not allow to specify sporadic behaviors. Sporadic threads are threads which are periodic, whereby only a value for $d$ is specified.

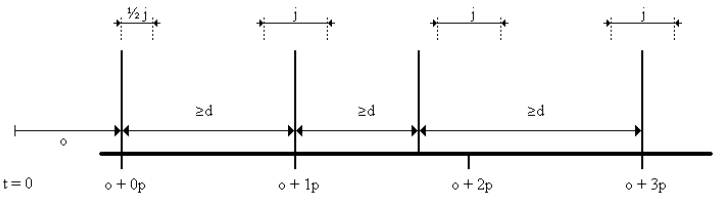


Fig. 1: Period (p), jitter (j), delay (d) and offset (o)

The implementation of the VDM-RT semantics is visible for the executable subset in the interpreters in VDMTools [3] and Overture [6]. In this paper we will limit our investigation to the VDMJ interpreter from Overture.

## 3 Open Semantic Issues in VDM-RT

The use of VDM-RT has lead to the identification of missing semantics definitions, or to constructs which are currently defined without distribution having been taken into account. Lastly, the semantic definition of some constructs deviates from the reality implemented in the VDM-RT interpreter. This all makes it hard for a user to specify a model which is faithful to the real world distributed system.

### 3.1 Public variables

The semantics of the current VDM-RT interpreter allows read and write access to all public variables in a model without the use of a BUS. This means that any public instance variable or value can be accessed from any CPU in a system without taking

distribution into account, where variables located on different CPUs require BUS communication. This will also influence the timing of the CPUs accessing the variable.

We suggest that all access to such public variables are done through a BUS if the CPU reading or writing the public variable is different from the CPU where the variable is located. We suggest that BUS communication is used for any public variables including static variables. Alternative solutions might be to assume that `get` and `set` operations are always implicitly available on class member variables. External references across CPU boundaries, even through public variables, should always be performed through these implicit `get` and `set` operations, much akin to the solution taken in `C#`. Another approach could be to disallow public member variable access across CPU boundaries by static or run-time checking.

### 3.2 Static variables and operations calls

In the semantics of the current VDM-RT interpreter static variables are globally available to all class instances in a system. Static variables can both be read and written to without taking deployment into account, thus no replication of static variables is done between CPUs in the system. The semantics does not distinguish if a class with a static variable is deployed to more than one CPU, or from which CPU a static variable is accessed. However, by ignoring distribution of static variables, the semantics positions itself far from reality since such distribution has to be carefully performed to ensure that static variables are updated globally at a particular time.

Static operations are allowed to be called from any class instance in a system and are executed on the CPU of the caller. Deployment is not considered thus no BUS communication. Ignoring distribution of static operations causes a number of problems, (1) by ignoring distribution and using static operations to access static variables the problem of static variables applies as described above; (2) if a static operation is called from a CPU which does not have any knowledge of the static operation(since no instances of its class is deployed to it) then it should not be possible to execute it locally, however, this is currently possible. If a static operation should be executed it must exist in the system and be deployed to a CPU; (3) If a static operation is accessed and an instance of a class where it is defined is deployed in a system, access to the definition should happen through a BUS, if accessed from another CPU than the one hosting the static definition. This is not the case since all static calls bypass all BUS communication.

We suggest to change the way static variables and operations are interpreted in VDM-RT.

Static variables should only exist within the CPU they have instances that are deployed on restricting direct access from other CPUs. If more than one instance of a class declaring a static variable are deployed to the same CPU the static variable should naturally be shared. Thus, in this way static variables will become shared per CPU.

Static operations should be available throughout the system, however, the execution of the operation may vary based on the deployment, which can be split up into three alternatives:

**Static definition available on calling CPU** The caller of a static operation is located on the same CPU as at least one instance of the class declaring the static operation.

In this case the operation can be accessed directly and all execution will take place at the same CPU.

**Static definition only available on virtual CPU** No instances of the class declaring the static operation exists on any system CPU, thus the caller does not have direct access to the static operation. In this case the static operation will be hosted by the virtual CPU, and the caller has to access the static operation through a BUS connection to the virtual CPU. The static operation is executed on the virtual CPU and the call on the callers CPU.

**More than one CPU hosts the static definition** If more than one CPU holds instances of the class declaring the static operation and no instance exists on the CPU of the caller, a CPU is selected non-deterministically to host the static operation call. Thus in this case the calling CPU request a call of the operation, this is send over the BUS to the host CPU which activates and executes the static operation and returns the result over the BUS to the calling CPU.

The suggestion provided for static operations does not give semantics to permission predicates specifying the scope of history counters for static operations. This is a problem if a static operation is evaluated on a non-deterministically chosen CPU where the history counters are local to the chosen CPU.

### 3.3 Time advance and periodic threads

The VDM-RT semantics in [14] and of the VDM-RT interpreter specifies that time must progress when expressions or statements are executed on a system CPU. However one exception is explicit use of a duration statement which require the virtual CPU to advance time as well. The semantics defines periodic threads as statements being activated for execution each $P$ time units on a CPU, it does not give any guarantee about the execution, only that it can execute after its activation time. However no semantics are defined for the case where a model is blocked but contains periodic threads which causes a model to wrongly deadlock. A deadlock occurs in the current semantics if a model consists of at least two threads, one being main and another being a periodic thread, where main blocks after starting the periodic thread. The reason for this deadlock is that time does not progress if nothing is executing, which occurs when main blocks and the current periodic thread finishes at time $t$ where $t < P$ and where $P$ is the period of the periodic thread running. If $t < P$ then the activation time of the next periodic thread is not reached and the model will deadlock. This is not aligned with the idea of periodic threads, where all periodic threads can be activated independent of any model state. The current semantics does not give any special semantics to multiple periodic threads located on the same CPU. The semantics only allow one thread to execute on a CPU at any point in time. However this is a problem if periodic threads are located on the virtual CPU and have the same period which in effect should let them execute synchronously because time is ignored on the virtual CPU.

We suggest that the semantics allow forward time jump in the execution of a model if all threads are blocked and one or more periodic threads exist. If a model is blocked at time $t$ it can suggest the master scheduler to jump to $P_e$ if $t < P_e$ when $P_e$ is the time of the next period of the periodic thread which has the earliest activation time.

## 4 Open Semantic Issues with Co-Simulation

Extending VDM real-time to support co-simulation as used in DESTECS [1] requires a number of new semantic definitions and clarifications of existing informal definitions. From [14] it shows that semantics must be given to *interrupts* and time must be given an international standard such as [s] seconds, so other simulators e.g. from the continuous time domain, can synchronise time during simulation.

### 4.1 Interrupts

Interrupts is a new way to connect model and environment where interrupts spawns new threads called *interrupt handlers* and influences the scheduling of a model. Interrupts can occur at any point in time during execution. In the semantics of VDM-RT a model is connected to the environment through a class instance deployed to the virtual CPU which feeds environment events into the model and allows the model to read environment state. We propose to extend the VDM-RT semantics with definitions for interrupt handling. An interrupt is an event occurring in the environment which is pushed into the model and handled by an *interrupt handler*. An interrupt handler must execute with a variable priority higher than all normal threads and it should be possible to determine if an interrupt handler has completed within a certain time frame. To enable interrupts in VDM-RT a more advanced scheduler, capable of prioritising interrupts, is required along with a clear definition of how interrupts influences the execution of both normal and asynchronous threads. Lastly, interrupt handlers should be asynchronous operations. Listing 5 is a proposal of a definition of an interrupt handler.

```
class InterruptHandler
operations
public async notify : () ==> ()
notify == is subclass responsibility;
end InterruptHandler
```

Listing 5: VDM `InterruptHandler` signature

However the definition in Listing 5 should not be part of the model but built into the tool. In Listing 6 an example is given of an interrupt handler for a button press event in the environment.

```
class ButtonPressInterruptHandler is subclass of InterruptHandler
public async notify : () ==> ()
notify == ...
end ButtonPressInterruptHandler
```

Listing 6: VDM `ButtonPressInterruptHandler` example for Button Press

We propose that interrupt handlers are to be registered on a CPU and not in a system. All interrupt handlers are then to be executed on the CPU they are bound to. The CPU can then be extended with an operation to register the interrupt handler such as:
`public regIntHandler : InterruptHandler * int ==> ()`
An interrupt handler should then be able to be registered by instantiation of the handler

class and then deployed to the CPU as shown in Listing 7 where an instance of the button press interrupt handler is registered on `cpu1`.

```
system S
...
cpu1 : CPU;
handler : ButtonPressInterruptHandler;
... -- inside the constructor
cpu1.regIntHandler(handler,MAX_PRIORITY);
...
end S
```

Listing 7: An example showing how an interrupt hander can be registered on a CPU.

Note that an alternative approach, involving a syntax change to the VDM RT notation, was proposed in [13].

### 4.2 Measurement of time in VDM-RT

In VDM-RT all expressions and statements have a predefined default execution time specified which is set to 2 time units for all expressions and statements. The semantics defines two constructs to explicitly override the default durations at runtime: durations and cycles. Duration statements define the number of time units the enclosing statement takes to override any inner (default) durations. The cycle statement defines the number of cycles the enclosing statement takes, the time units can be derived by dividing the number of cycles by the CPU speed+1.

However, defining time as time units is not adequate to enable a co-simulation with a continuous time simulator, since time must be synchronised between the two simulation engines for the outcome of the simulation to be valid. The architecture of the VDM-RT controller will not correspond to reality if synchronisation of time is wrong, since it will affect the calculation speed of the architecture of the VDM-RT controller.

We suggest to give time steps a unit. This will enable synchronisation with a continuous time simulator. If such a unit is chosen to be *[s] seconds* then we would also suggest that the capacity of CPUs is changed into calculations per seconds specified in *[Hz]* and BUS rates to be specified in *[bit/s]*. Duration statements should also use seconds to specify how long the enclosing statement takes. Alternative, if no time unit is chosen but a clear connection between CPUs, BUSes, duration and cycle statements is defined, then a mapping between the VDM-RT model time and the connected continuous time co-simulator notion of time can be defined as a parameter to the VDM-RT interpreter (mapping logical time units in VDM-RT into real-time and vice versa).

Lastly, we would also suggest to change default durations for all expressions and statements from durations into cycles, this way a model's timing will change if the CPU capacity changes. Furthermore, each expression and statement should have individual cycles specified. The cycles of an assignment expression given by `x := y + z` can be calculated by breaking it down into low level commands as shown in Table 1 where the `mov` move command is used to move `y` and `z` into two registries and then `cmp` is used to combine them and move them to `x`. This is 4 cycles in total for the assignment.

If this assignment was to be executed on a CPU running at 2 KHz, performing 2000 cycles / second then the assignment would take 0.002 second to complete.

| Cycles of VDM assignment := | |
|---|---|
| Instruction | Description |
| `mov` | Move y to registry |
| `mov` | Move z to registry |
| `cmp` | Combine the two registries |
| `mov` | Move result to x |
| 4 | Total cycles |

Table 1: Calculation of VDM equals by use of assembly instructions.

### 4.3 Limitation of Co-Simulation interface

The co-simulation interface is limited to support a small set of types understood by both discrete time and continuous time simulators. This set will consist of types such as booleans, integers and doubles/reals no complex structures can be transferred. However, a common way of handling decimal numbers is required so simulation engines will be able to compare decimal numbers across the interface. A question remaining about restricted types e.g. an integer with an invariant attached. Should this be enabled and when should they then be checked?

## 5 Future Work

In addition to the issues raised above there are a number of additional desirable extensions to VDM-RT that we think are worth consideration in the future. When semantics work for VDM-RT is undertaken we propose that such possible language extensions are also considered for their semantics consequences for including each of them can be based on a proper semantic foundation when a decision is to be taken by the VDM-10 language board. The desirable extensions include:

- As indicated in Section 3.3 it is necessary to have multiple virtual CPUs in order to specify fully time independent input stimuli in the environment. Essentially a new virtual CPU is assigned to each new instance which is created outside the the scope of the `system` class. This would have the advantage that the threads executing on objects deployed at the virtual CPU would be fully independent of each other from a timing perspective.
- Enabling the use of the **`time`** keyword in permission predicates and pre and post conditions for operations. This would give more expressive power to the VDM modeller but it is so far unclear what the semantic consequences would be and how it would impact the performance of the interpreter.

– Identically, history counters (as used in permission predicates) should be extended with a notion of time. For example, `#age` which would return the amount of time passed since the last `#req` was passed. This could be used to specify response time requirements as a precondition, or maximum operation elapse times as a postcondition. For more examples, see [13].
– As indicated before, add the ability to specify so-called sporadic periodic threads, for example using the concrete syntax

```
threads sporadic (d) drinkWine
```

where $d$ only specifies the minimum interarrival time between two periodic calls to `drinkWine`.
– Include probabilistic capabilities in the semantics for both periodic (and sporadic) threads and duration statements. For example, we could write

```
duration(10, 100) drinkBeer();
```

to specify that the execution of the operation `drinkBeer` would take an arbitrary amount of time chosen from the $[10, ..., 100]$ interval. Arbitrary could mean that the strategy used is specified as a global parameter at simulation level (i.e. a random value taken from a normal or exponential distribution). As an extension, one could even explicitly specify how the value from the interval is taken by adding an extra parameter, which refers to an algorithm, taking the two bounds as input parameters, which on return computes the value. This would for example allow the ability to specify caching behaviour (first call is expensive, next calls are cheap). The same strategy could be followed for the looseness provided in periodic threads with jitter or sporadic threads. However evaluation of the algorithm must be done without affecting the timing of the model in which it is declared.
– The current system class forces a static topology of the system components. It is worthwhile analyzsng what the semantic consequences would be for enabling a dynamic deployment architecture [9].
– What about including multiple BUSes between two CPUs and if that is specified how shall the routing of such messages be decided upon?
– Shall it be possible to let the user set a limit for buffer size for the messages received at a CPU? Currently, when a remote call is attempted, the caller is not blocked when the BUS is busy. It assumes that there exists an infinite buffer between the BUS and each CPU. Detecting design bottlenecks is therefore potentially hampered because these limits do exist in reality. Perhaps the buffer size should be a global simulator setting, or a configurable item per CPU or per CPU/BUS interconnection.
– Would it be desired to introduce a new construct to express *broadcast* messages, allowing a model to contain public static broadcast operations which can be picked up from any class on any CPU? If introduced should it then be a new keyword **`broadcast`** or a new use of **`static`** and **`async`**.
– What would the semantics be if we had multiple system classes in order to describe system of systems?
– What are the semantics with VDM exceptions (ie. errors) thrown in inter-CPU calls? In particular, what happens to exceptions raised by async calls? Do they propagate over the physical CPU border? What happens with synchronous calls

across a CPU boundary? What if the message gets lost on the bus (relevant in the context of the DESTECS project)? Does this lead to an exception on the caller CPU?
- We do not have explicit definitions of the scheduling policies. In fact some of the names of these are misleading. How shall it be possible to define more of these in the future both for CPUs and BUSes? Can they be specified in a late bound fashion, as part of the model, rather than as a built-in part of the language?

## 6 Concluding Remarks

This paper have pointed out a number of semantics issues that needs to be solved for the VDM-RT language. The plan for the future is to get an agreement in the Overture Language Board about what the semantics shall be like and then define the semantics of the VDM-RT extensions in relation to the ISO VDM-SL standard. This will most likely be in the same operational style as used in [14] but inspiration will also be taken from other languages with similar constructs such as Circus with time and resources [12]. Similarly, for the probabilistic extensions, inspiration can be gained from the MODEST language and supporting tools [5]. With respect to the definition of the VDM RT semantics, one could for example break up the work into tasks such as:

- Define core abstract syntax (CAS) for VDM
- Define mappings from VDM dialects to CAS
- Define CAS semantics

Here the semantics could be split up into areas of interest (e.g. concurrency scheduling, object orientation, time and distribution/deployment).

At the practical level the plan is that Kenneth Lausdahl and Peter Gorm Larsen will spend a week at York University with Jim Woodcock at the beginning of November 2010 to work on this semantic definition. We hope that more volunteers are interested in taking part in this semantics effort.

## Acknowledgements

The work reported in this paper have partly been carried out in the DESTECS project which have partially been funded by the European Commission. In addition, we would like to thank Nick Battle for proof reading a draft of this paper.

## References

1. J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. Jovanovic, K. Pierce, and Wouters F. Design support and tooling for dependable embedded control software. In *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*. ACM, April 2010.
2. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

3. John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008.
4. J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*, 2008. edited by Benjamin Wah, John Wiley & Sons, Inc.
5. Arnd Hartmanns and Holger Hermanns. A modest approach to checking probabilistic timed automata. In *QEST*. IEEE Computer Society, September 2009.
6. Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1), January 2010.
7. Peter Lucas. VDM: Origins, Hopes, and Achievements. In Airchinnigh Bjørner, Jones and Neuhold, editors, *VDM '87 VDM – A Formal Method at Work*, pages 1–18. VDM-Europe, Springer-Verlag LNCS 252, 1987.
8. Paul Mukherjee, Fabien Bousquet, Jérôme Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.
9. Claus Ballegaard Nielsen. Towards dynamic reconfiguration of distributed systems in vdm-rt. In *Semantic Issues in VDM: a BCS-FACS and Overture Workshop*, September 2010.
10. P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.
11. Nico Plat and Hans Toetenel. A formal transformation from the BSI/VDM-SL concrete syntax to the core abstract syntax. Technical Report 92-07, Delft University, March 1992.
12. Adnan Sherif, Ana Cavalcanti, He Jifeng, and Augusto Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22:153–191, 2010.
13. Marcel Verhoef. On the use of VDM++ for specifying real-time systems. *Proc. First Overture workshop*, November 2005.
14. Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2008. ISBN 978-90-9023705-3.
15. Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 147–162. Lecture Notes in Computer Science 4085, 2006.
16. Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System Architecture Evaluation Using Modular Performance Analysis – A Case Study. In Tiziana Margaria, Bernard Steffen, Anna Philippou, and Manfred Reitenspiess, editors, *Proc. 1st International Symposium On Leveraging Applications of Formal Methods (ISOLA 2004)*, pages 209–220, Paphos, Cyprus, November 2004. University of Paphos, Department of Computer Science. Also available at *http://www.cs.ru.nl/research/reports/info/ICIS-R05005.html*.

# The Real-Time/Co-Simulation Discussion Session

Editor: Sune Wolff[2,1]

[1] Aarhus School of Engineering, Dalgas Avenue 2, DK-8000 Aarhus C, Denmark
[2] Terma A/S, Hovmarken 4, DK-8520 Lystrup, Denmark
swo@iha.dk

## 1 Introduction

The following is a summary of the group discussion of the VDM-RT and co-simulation session of the Overture workshop. Due to limited time, not all groups discussed all issues raised by Kenneth Lausdahl and Marcel Verhoef during their presentation. The following notes were taken during these three group sessions.

## 2 Group: Mathhew Lovert, Leo Freitas, Erik Ernst, Nico Plat, Marcel Verhoef (minutes), Gudmund Grov

### 2.1 Issue 1 — Public variables

Issue: Accessing public variables on another CPU is done instantly without taking distribution constraints into account.

Question: does VDM-RT support the ability to specify data sharing architectures? In principle yes, i.e. by sharing public instance variables across CPU boundaries. This gives rise to the issue raised in the talk by Kenneth (Section 3.1 in the paper). In our opinion this should be solved by implicitly assuming the availability of ”get” and ”set” operations for each instance variable declared. Referring to an instance variable should always be replaced (behind the scenes) as a call to the “get” or “set” operation, which in all cases will lead to the desired effect. Cross-platform access will “automatically” take the performance penalty associated with bus communication into account.

### 2.2 Issue 2 — Static variables

Issue: Static variables are global and available from all CPUs without taking distribution into account.

With respect to this issue, the suggestion is to abandon static instance variables altogether, because the added complexity does not help and their use remains unrealistic, both from the modeling perspective as well as the practical situation that it describes. The solution is to create an encapsulating class that keeps references to normal objects (class instances) and pass an instance of this encapsulation around whenever the system class is created. All resources and deployed objects can then refer to these global instance variables at their convenience and even treat them differently at a local level, much akin to the workspace class that existed in the original VDM++ language.

### 2.3 Issue 5 — Interrupts

Issue: Interrupts are desired for co-simulation since incoming events can be prioritized.

We agreed that this issue was not so much about the definition of interrupts (for which we believe the existing syntax is already sufficiently powerful), but that the ability to specify thread priorities is currently lacking. This becomes relevant if priority-aware schedulers are considered.

### 2.4 General Notes

At York University, a recent PhD graduate has worked on operational semantics for Handel-C, an extension of C that allows for the fine-grained definition of concurrency. The language supports many of the features included and proposed in VDM-RT, such as durations (with upper and lower bounds). The semantics is rooted in Unified Theories of Programming, leading to an algebraic specification that is manipulated using rewrite rule logic. The work was performed by Juan Perna. This appears to be very relevant related work that needs further investigation. Contact through Leo Freitas / Jim Woodcock.

The debate continued on the real need for OO in a modeling language. Do we really need it? Or would VDM-SL with some extensions for concurrency for example suffice? Furthermore, it was commonly agreed that checking some other well-known paradigms, such as ADA, POOSL, Handel-C and ERLANG may be beneficial for the semantics discussion.

Erik raised the open question as to whether `duration(0)` [and `cycles(0)`] should be allowed at all inside classes that are deployed on a CPU? Is there use for explicit `duration(0)` specifications inside the system?

## 3 Group: Peter Gorm Larsen, John Fitzgerald, Jan Broenink, Claus Nielsen and Sune Wolff

### 3.1 Issue 6 — Measurement of time in VDM-RT

The main issue is that no semantics definition exists which defined the relation between `CPU` and `BUS` capacity and `duration` and `cycles`. The historical reason for this was to give freedom to the model designer, to choose time units which suited his needs. The relations described in the VDM Language Manual only describe suggestions and do not give a clear definition.

**Short term plan:** The group agreed that the language manual should be updated to clearly describe the suggestion of Kenneth Lausdahl and Marcel Verhoef, namely that `CPU` capacity is specified in calculations per second [Hz], `BUS` capacity is specified as bits per second and `duration` is specified in seconds.

**Long term plan:** The use of a separate theory describing dimensions and conversion rules should be investigated. A model can then specify the unit used any necessary conversions are done automatically. This would also be a major advantage in a co-simulation setup, where multiple models can select their unit of choice and not have to perform conversions directly.

In his presentation, Kenneth Lausdahl also suggested changing default `duration` to be specified using `cycles` instead. The group quickly agreed on supporting this suggestion. Currently, the default `duration` is of two time units and it was suggested to make this more fine-grained to reflect reality more closely. The group supported this suggestion, but also stated the importance of user configurable default `cycles`.

### 3.2 Issue 7 — Limitation of co-simulation interface

Current design of the co-simulation interface suggests that only simple types (e.g. integers, boolean and reals/doubles) can be transferred, as opposed to more complex types such as custom types or class hierarchies. The group discussed whether a language extension was needed in order to transfer more complex data types. A common agreement was that this was currently not needed — if it is crucial to transfer a record type for example, appropriate conversion could be made in each model instead.

Another issue is that invariants are not available in the co-simulation interface — and if they are to be made available, when should they hold? The group discussed limiters in 20-sim, which act similar to invariants in VDM. A limiter forces a signal above/below a certain limit, directly altering the signal instead of reporting a breach of an invariant. The group agreed that for now, invariants are to be kept in the discrete controller and hence saw no issue.

Finally, the precision of decimal numbers in the interface was raised. Since the group already agreed to keep values transferred simple (i.e. using only double precision floating points), the precision is defined by the data type and hence there is no issue right now. This is certainly a point worth re-investigating in the future when the co-simulation interface must support transfer of multiple data types both simple and complex.

## 4 Group: Nick Battle, Kenneth Lausdahl, Ken Pierce and Shin Sahara

### 4.1 Issue 1 — Public variables

We concluded that this should not be allowed. However the tool could wrap such a public variable access in an operation `get` and `set` which then should require a `BUS` to be present between the `CPU`s where the public variable might be shared.

### 4.2 Issue 2 — Static variables

In general we were in favor of removing *static* from VDM-RT. We had two ideas:

**Global:** The idea here is that all static variables are placed in one storage unit in the system (like an instance of a class). Then all static variable access would be done to this storage according to the recommendations we made to Issue 1: Public Variables.

**CPU local storage:** All static variables are only static within the CPU they are used on. This is akin to the real world in the case where no explicit synchronization between distributed units of a system is implemented.

### 4.3 Issue 3 — Static operations calls

Simply copy the definition onto each CPU and evaluate the operation on the CPU they are called from. This does not require a change to the syntax, just changes to the semantics and documentation.

### 4.4 Issue 4 — Time advance and periodic threads

Agreed to the proposed suggestion.

### 4.5 Issue 6 — Measurement of time in VDM-RT

The idea of changing the default duration to default cycles should be done. All agreed it would be a good idea to use a standardised unit to define the system.

### 4.6 Issue 7 — Limitation of the co-simulation interface

The invariant only needs to hold when time progresses in the VDM model.

# Conclusions

The 8th Overture Workshop was a great success and possibly one of the most productive so far. Thanks go to the organisers, for making the workshop run smoothly; to the contributors, for all their hard work; to the other participants, for a very productive discussion; and finally to the BCS-FACS for their hospitality and use of their facilities.

The aim now is to keep up the momentum so that together we can drive forward the development of VDM and Overture, for the benefit of ourselves and our users — especially those potential users whom we hope to attract in the future. To this end, a plan of action was created to map out the next steps on our journey.

As noted previously, it was decided in the short term that Nick Battle's proposals will be incorporated into VDMJ/Overture and VDMTools. In the longer term, the work was broken down into five areas (given below). People interested in contributing to the individual tasks were noted.

**Define core VDM semantics:** John Fitzgerald, Peter, Erik, Ken, Joey (mainly reviewing) and Cliff (reviewing).
**Define core abstract syntax:** Peter, Erik, John Fitzgerald, Nick, Ken and Joey (mainly reviewing).
**Explore existing examples:** Peter, Erik, Ken, Nick and Leo.
**Define syntactic sugar needed for core VDM semantics:** Marcel.
**Create tool for extracting model statistics:** Nick.

In addition to producing a road map for the near future, we successfully bid to hold a workshop at the FM 2011 conference in Limerick, Ireland. The date is 20th June 2011. We would welcome all participants to return and perhaps contribute papers. We would also love to see new participants joining us. Based on this year's success, it promises to be an excellent workshop. See you in Limerick!