SEMANTICS OF THE
VDM REAL-TIME DIALECT

Electrical and Computer Engineering
Technical Report ECE-TR-13

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: Semantics of the VDM Real-Time Dialect

Subtitle: Electrical and Computer Engineering

Series title and no.: Technical report ECE-TR-13

Authors: Kenneth Lausdahl, Joey W. Coleman and Peter Gorm Larsen
Department of Engineering – Electrical and Computer Engineering,
Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: http://www.eng.au.dk

Year of publication: 2013 Pages: 65
Editing completed: May 2013

Abstract: All formally defined languages need to be given an unambiguous semantics such that the meaning of all models expressed using the language is clear. In this technical report a semantic model is provided for the Real-Time dialect of the Vienna Development Method (VDM). This builds upon both the formal semantics provided for the ISO standard VDM Specification Language, and on other work on the core of the VDM-RT notation. Although none of the VDM dialects are executable in general, the primary focus of the work presented here is on the executable subset. This focus is result of parallel work on an interpreter implementation for VDM-RT that chooses one of the pos-sible interpretations of a given model that is expressed in VDM-RT, based on the semantics presented here.

Keywords: Vienna Development Method, VDM, real-time, VDM-RT.

Please cite as: Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. 2013. Semantics of the VDM Real-Time Dialect. Department of Engineering, Aarhus University, Denmark. 65 pp. - Technical Report ECE-TR-13

Cover image: Created by Overture.

ISSN: 2245-2087

SEMATICS OF THE VDM REAL-TIME DIALECT

Kenneth Lausdahl, Joey W. Coleman and Peter Gorm Larsen

Aarhus University, Department of Engineering

Abstract

All formally defined languages need to be given an unambiguous semantics such that the meaning of all models expressed using the language is clear. In this technical report a semantic model is provided for the Real-Time dialect of the Vienna Development Method (VDM). This builds upon both the formal semantics provided for the ISO standard VDM Specification Language, and on other work on the core of the VDM-RT notation. Although none of the VDM dialects are executable in general, the primary focus of the work presented here is on the executable subset. This focus is result of parallel work on an interpreter implementation for VDM-RT that chooses one of the possible interpretations of a given model that is expressed in VDM-RT, based on the semantics presented here.

Contents

Chapter 1

Introduction

1.1 Styles of Semantic Definitions

Semantic models can be given in many different styles (e.g. axiomatic, denotational and operational). When using an *axiomatic* definition style, the meaning of a model expressed in a formal language is provided by describing its effect on assertions about the state of the model. The most well-known axiomatic definition style is known as Hoare Logic [Hoa69]. When using a *denotational* definition style, the meaning of a model expressed in a formal language is provided in a compositional way using mathematical objects [Str67, Sto77]. Here there is a clear distinction between syntactic and semantic domains [Sco82]. When using an *operational* definition style, the meaning of a model expressed in a formal language is provided through the definition of computational steps that may be taken [Plo81]. Here there is a distinction between the notions of small-step semantic definitions and big-step semantic definitions. Both of these are used in the semantic model given in this technical report.

1.2 The Vienna Development Method

VDM's origins lie in the work on semantics of programming languages at IBM's Vienna Laboratory [BJ78]. The basic modelling language is based on discrete mathematics with set theory, and its denotational semantics have been standardised [LP95]. A proof theory has also been defined, based on the typed Logic of Partial Functions (LPF) [BCJ84, JM93, BFL$^+$94].

Basic VDM models are expressed in a specification language (VDM-SL) that supports the description of data and functionality. Data is defined by means of types built using constructors that define structured data and collections such as sets, sequences and mappings from basic values such as Booleans and numbers. These types are very abstract, allowing the user to add any relevant constraints as data type invariants. Functionality is defined in terms of functions and operations over these data types. Functions and operations can be defined implicitly using pre-conditions and post-conditions that characterize their behaviour, or explicitly with specific algorithms. The syntax of VDM may be expressed either by using a mathematical notation or by using an ASCII syntax that can be readily input on an ordinary keyboard[1].

An extension of VDM-SL, called VDM++, supports the object-oriented structuring of models and permits direct modelling of concurrency [FLM$^+$05]. VDM++ was originally developed in the European research project "Afrodite". A further extension of VDM++ is VDM Real-Time (VDM-RT), which enables modelling of real-time and distributed systems. VDM-RT was first developed in the European research project "VDM In Constrained Environments" (VICE), but this initial version only allowed for

[1] In this technical report we will consistently use the ASCII syntax when we show example models that we give semantics for, and the mathematical syntax whenever we provide auxiliary functions/expressions used in the definition of the semantic model.

a single CPU [MBD^{+}00]. This was extended to cope with distributed systems in Marcel Verhoef's PhD thesis [Ver09, VLH06].

Using VDM-RT it is possible to define a distributed architecture with multiple CPUs and the busses that connect them. Multiple threads may be present on each CPU and the scheduling policy for these is parametrized per CPU.

All three VDM dialects are supported by an open source tool called Overture [LBF^{+}10]. An executable subset of the VDM dialects –including non-deterministic elements– can be simulated using the built-in interpreter [LLB11]. The simulation will exhibit the behaviour of one of the valid semantic interpretations in the presence of looseness.[2]

1.3 Structural Operational Semantics

We use the Structural Operational Semantics (SOS) format [Plo81, Plo04] to present the semantic definitions in this technical report. An SOS description consists of two major elements: a set of type definitions that describe the static structure of the system; and the definitions of the transition relations that describe the behaviour of the system. The type definitions may also be accompanied by context conditions — further constraints on the types that are analogous to the static checking done by a programming language compiler.

The logical notation used in this technical report is the basic VDM-SL type system and expressions. This notation is used to define the static structure of the VDM-RT language.

In an SOS definition, the entire system is modelled as a *configuration* containing all of the information needed to capture the state of a system at any given point. A configuration is typically given as a tuple, in this case of the listed components.

The behaviour of a system is defined through the use of transition relations, at least one of which must involve the system's configuration type. In a small-step SOS definition the overall system behaviour is typically defined using a transition relation from configurations to configurations.

The transition relations are defined through the use of inference rule schemata where each rule's conclusion defines a subset of the entire transition relation. The least relation that satisfies all of the inference rules is taken to be the relation defined.

In the work presented here we focus on the executable subset of VDM-RT and thus the collection of SOS rules defined will be incomplete in the sense that it is not supplying the SOS rules for the semantics of VDM expressions. For VDM-RT expressions we take the semantic model provided for VDM-SL [LP95] (described further below in Section 3.1). This also means that we do not go into the rules relevant to deal with undefinedness, i.e. using LPF.

1.4 Structure of this Technical report

After this introduction, Chapter 2 provides an overview of the main concepts in VDM and the specific features of VDM-RT. Then Chapter 3 provides an overview of the existing related work on semantics of VDM dialects. Afterwards Chapter 4 illustrates how SOS is used to give the semantics of VDM-RT, building on top of the previous semantic efforts. Finally, Chapter 5 provides concluding remarks about the work presented here. Appendix A provides the full SOS semantics of VDM-RT.

[2] The notion of looseness is explained in Section 3.1.3.

Chapter 2

Overview of VDM and VDM-RT Features

2.1 System Modelling in VDM

The use of VDM involves the development and analysis of models to help understand systems and predict their properties. Good models exhibit abstraction and rigour. *Abstraction* is the suppression of detail that is not relevant to the purpose for which a model is constructed [Kra07]. The decision about what to include and what to omit from an abstract model requires good engineering judgement. A guiding principle in VDM is that only elements relevant to the model's purpose should be included; it follows that the model's purpose should be clearly understood and described. *Rigour* in the semantics makes it possible to perform a mathematical analysis of the model's properties in order to gain confidence that an accurate implementation of the modelled system will have certain key characteristics.

In computing systems development, modelling and design notations with a strong mathematical basis are termed formal. VDM models, although often expressed in an executable subset, are developed primarily for analysis such as formal proofs rather than serving as final implementations.

2.2 Model Structure

In VDM, models consist of representations of the data on which a system operates and the functionality that is to be performed. The data represented includes the externally visible input/output and internal state data. The functionality includes the operations that may be invoked at the system interface as well as auxiliary functions that exist mainly to assist in the definition of the operations.

The VDM++ language extends VDM-SL (without modules) with facilities for specification of object-oriented systems, and structures models into class definitions. Each of the class definitions has similar elements to a single VDM-SL specification and, relative to the usual object-oriented languages, state variables take on the role of instance variables and operations play the part of methods. The remainder of this section will restrict consideration to VDM-SL, with VDM++ considered at a later stage.

2.3 Modelling Data

Data models in VDM are built on basic abstract data types together with a collection of type constructors. A full account of VDM-SL data types and type constructors is provided in current texts [FL98, FL09].

Basic types include Booleans, numbers (natural, integer, rational and real) and characters. Note that, in accordance with VDM's abstraction principle, these correspond the mathematical notions of numbers, and are not bounded by constraints due to their representations in computing hardware[1]. If a user wishes

[1] Naturally, tools that support VDM have the same sort of representational constraints as are found in most programming languages.

to specify these limits because they are relevant to the problem being modelled, it is possible to do so explicitly by means of invariants. Invariants are logical expressions (predicates) that represent conditions to be respected by all elements of the data type to which they are attached.

The VDM ISO Standard permits both an ASCII and mathematical syntax; where the ASCII syntax is considered more accessible for readers unfamiliar with the notations of discrete mathematics. Keywords are, by convention, shown in bold face. Consider, as a simple example, a system for monitoring the flight paths of aircraft in a controlled airspace. A simple data type definition representing the `Latitude` of an aircraft would be given as follows:

```
Latitude = real
```

If it is desired to restrict the `Latitude` to the range of numbers from -90 to 90 inclusive, an additional condition is added to the data type in the form of an invariant. This extended type definition is as follows:

```
Latitude = real
inv lat == -90 <= lat and lat <= 90
```

The invariant is an integral part of the data type. Thus, it is not possible to create a value of type `Latitude` that does not respect the invariant. The modeller must ensure that all functions and operations that create such elements respect the invariant.

More sophisticated data types are built using constructors. A record type constructor permits the definition of tuples with named fields. For example, assuming definitions of types representing `Latitude`, `Longitude` and `Altitude`, it is possible to define a type of values representing aircraft position, as follows:

```
Position :: lat  : Latitude
            long : Longitude
            alt  : Altitude
```

A value of type `Position` is a composite whose component values can be extracted by giving the field names. Thus, the `Longitude` component of a position ‘`p`’ is given by ‘`p.long`’. VDM-SL also contains type constructors for building union and Cartesian product types.

Models are typically built around structured collections of values, so VDM-SL provides type constructors that support several collection types: sets (finite unordered collections), sequences (finite ordered collections), which both uses 1-relative indexes, and mappings (finite functions). For example, one may wish to define a type to model the path of an aircraft as a finite sequence of positions. The corresponding definition is:

```
FlightPath = seq of Position
```

Thus, an element of type `FlightPath` is a finite sequence of position records. Given a value `fp` of type `FlightPath`, the initial `Altitude` is expressed as `fp(1).alt`. If modelling a flight control system that must manage several aircraft, it would be appropriate to define a type that relates aircraft identifiers to their flight paths as a mapping:

```
FlightDetails = map AircraftId to FlightPath
```

A mapping in VDM is the abstract model of an associative array; individual associations are represented using an "arrow" notation, e.g. {3 |-> "text1", 7 |-> "text2"} represents an association between numbers and character strings. In the flight details example, the mapping represents a finite collection of flight paths indexed by the aircraft identifier. Given a flight details mapping fd and an aircraft identifier a, the following expression denotes the initial Altitude of a:

```
fd(a)(1).alt
```

Several special basic types also facilitate abstraction. The token type is used to denote values whose representations are immaterial. Tokens can be compared for equality, but have no internal representation so no other operators may be applied to them. Tokens are particularly useful for defining types that are necessary to a model but for which no individual elements are required. For example, if the air traffic model is concerned primarily with flight paths rather than call signs, the modeller may choose not to give a detailed representation for the AircraftId type, preferring to use a token type:

```
AircraftId = token
```

2.4 Modelling Functionality

Functionality is described in terms of functions and operations that accept input values and deliver output values belonging to the types defined in the model. As with data, VDM-SL contains features to support abstraction of functionality.

Each basic type and type constructor has associated syntax allowing values to be expressed. For example a sequence of four natural numbers might be expressed directly as follows:

```
[3, 7, 7, 2]
```

Comprehension notations allow more sophisticated constructions. For example, the following expression represents a sequence of all the squares of numbers up to 25:

```
[n**2 | n in set {1,...,25}]
```

The types are equipped with operators that allow complex expressions to be constructed. For example, given a value s belonging to a sequence type, the expression **len** s denotes the length of the sequence. Two sequences s1 and s2 may be concatenated by an infix operator: s1 ^ s2.

As in programming languages, some operators are partial, i.e., undefined for certain values of their arguments. For example, a sequence look-up such as the expression s(i) is undefined if the sequence s contains fewer than i elements. Such misapplications of partial operators correspond to potential runtime errors in a corresponding implementation. The behaviour of a real computing system when such an error occurs is not usually predictable. An error message may be returned, or an infinite loop may be entered, for example. Since such behaviour can rarely be known at modelling time, VDM treats them all as mathematically undefined in the semantics. From a tool perspective it is possible to automatically generate proof obligations ensuring that such internal consistencies will never appear [AL97, RL10].

Functions may be described explicitly or implicitly. An explicit function definition is an expression that denotes the result to be returned in terms of input parameters. Returning to the air space management example, the modeller may wish to specify a function that adds a new position on to the end of a flight path. The function definition is given as follows:

```
AddPos: FlightPath * Position -> FlightPath
AddPos(fp,p) == fp ^ [p]
```

Implicit function definitions provide an important abstraction capability in VDM. While an explicit definition like the one shown above is concise, the presence of the concrete algorithm in the definition's body may bias a reader implementing the model towards a particular implementation, for example by using a corresponding concatenation operator built in the implementation's programming language. An implicit definition describes a function purely in terms of the result to be delivered, with no direct reference to any algorithm to be used in the computation. This definition is given in terms of a logical (Boolean) expression that must be satisfied by the result. This expression is termed a post-condition. A classical example is a specification of a function for computing the square root r of a natural number n:

```
SQRT(n:nat)r:real
post r * r = n
```

Here the required result is merely characterized, with no bias towards any particular implementation. In particular, it will be noted that the post-condition does not constrain the result to be either positive or negative; the modeller has indicated that either result will suffice provided that it is a square root of the input n. Such implicit specifications are valuable when the provision of an algorithmic description would obscure the meaning of the model. The disadvantage is that an implicit operation specification is not directly executable.[2] In the airspace management example, an implicit specification might be used for a function to select a specific aircraft for landing, as in the following example. Here the **in set dom** construction means that the result returned is present in the domain of the flight details mapping structure:

```
Select(fd:FlightDetails)a:AircraftId
post a in set dom fd
```

There are cases where neither explicitly- nor implicitly-defined functions are sufficient. For example, the function above would not be able to return a result if the flight details mapping fd were empty. The function description is thus not satisfiable for all valid inputs. Therefore, the non-emptiness of the input fd is a pre-condition on the successful application of the function. Such pre-conditions are recorded explicitly in VDM. So, a satisfiable specification of the Select function would be as follows:

```
Select(fd:FlightDetails)a:AircraftId
pre dom fd <> {}
post a in set dom fd
```

Conditions, like invariants, provide a means of recording constraints that are often left unrecorded in informal descriptions of computer-based systems. In the example above, the pre-condition is required in order to ensure that the function is capable of returning a correct result in accordance with the post-condition. An implicit specification can be considered a contract: an implementation of the operation promises to return a result satisfying the post-condition provided the calling environment ensures that the pre-condition is satisfied. If the pre-condition is not satisfied, no guarantees about behaviour are made.

[2] Although the desirability of direct execution has been debated in the literature [HJ89, Fuc92].

2.5 Modelling State and Operations

Many systems have persistent state variables that are read and modified by operations, and which retain data between operation invocations. In VDM, such systems are modelled by defining a distinguished state variable of a defined type, and operations that, like functions, deliver outputs from inputs but which may also have side effects on the state variables.

A state-based version of the airspace management system might have a single state variable of type `FlightDetails`, modelling the current state of the airspace:

```
state Airspace of
  fd: FlightDetails
end
```

An operation to add a new aircraft with a single position `p` in its flight path might be specified implicitly as follows. Note the use of `~fd` to denote the state variable's value before execution of the operation. This decorated version is required since the post-condition describes a mathematical relation between the pre-operation and post-operation state. The **`munion`** operator used in the post-condition here forms the union of two mappings provided the two mappings do not disagree (any values that are in both domains must map to the same range value).

```
New(a:AircraftId,p:Position)
ext wr fd: FlightDetails
pre a not in set dom fd
post fd = ~fd munion {a |-> [p]}
```

Operations may be specified explicitly as well as implicitly. Where state variables may be modified, the language for expressing such explicit operation definitions is close to that of a classical imperative programming language, albeit one with very abstract data types. For example, the following explicit definition of the `NewOp` operation contains a single assignment to describe the updating of the `fd` state component.

```
NewOp: AircraftId * Position ==> ()
NewOp(a,p) == fd := fd munion {a |-> [p]}
pre a not in set dom fd
```

Full details of implicit and explicit specification styles for both functions and operations can be found in the VDM-SL literature [FL98, FL09].

2.6 Modelling Object-oriented and Concurrent Systems in VDM++

VDM++ provides facilities for the creation of object-oriented descriptions of systems. The core elements of classical VDM-SL –types, values, expressions, functions, and operations– are present. The extended language also provides for models based on class definitions in which each object's local state is represented as instance variables. Information hiding and multiple inheritance is also supported.

VDM-SL is limited to the description of sequential system models, although such models may be implemented in a parallel computing framework. The challenge in modelling concurrent computation is that separate threads (independent sequences of computations) may communicate through shared variables and inconsistencies can arise when two or more independent threads access a shared instance variable simultaneously. There has been considerable research on handling shared variable concurrency in VDM, notably by extending the pre/post-condition framework with rely and guarantee conditions that state, respectively, the properties that an operation requires to be invariant and the properties that it guarantees to maintain during its execution [Jon96].

The rely/guarantee approach has been a significant contribution to design methodologies for concurrent systems generally. VDM++ takes a rather pragmatic line. Here inconsistencies may arise through simultaneous access to shared objects by separate threads. These are avoided by providing synchronization constraints in the form of permission predicates that describe the conditions under which an operation may be carried out. A permission predicate may refer to an instance variable used as a flag to prevent other threads from using an object that is being used in a critical way by another thread. It may also access special variables representing the number of times each operation in an object has been requested, activated or completed, or representing the number of currently active invocations of the current operation. Consider a simple model in which a sensor produces data, writes it to a buffer object and this data is consumed by a consumer object. The buffer object provides a data model of the buffer and operations to `Put` and `Get` data. The consumer object should only invoke the `Get` operation on the buffer when there is actually data to get. This restriction could be modelled by allowing a special value **`nil`** to indicate emptiness of the buffer, in which case the permission predicate (denoted by the keyword **`per`**) on the `Get` operation in the buffer object is of the form shown below:

```
per Get => data <> nil
```

If such a special **`nil`** flag is not available, one could count the number of completed `Put` and `Get` operations and permit a `Get` operation under the condition specified as follows:

```
per Get => #fin(Put) - #fin(Get)  = 1;
mutex(Put,Get)
```

Here `#fin(op)` represents the number of completed occurrences of the operation `op`. The mutex condition enforces mutual exclusion of the `Put` and `Get` operations.

Permission predicates are different from operation pre-conditions. A permission predicate determines whether a request to perform an operation will be granted or blocked. If the permission is denied, another thread may be executing. A pre-condition is a well-formedness constraint on an operation invocation; if it evaluates to false when an operation call is requested, the modelling equivalent of a run-time error occurs because the caller has not satisfied the pre-condition, and has thus broken the contract.

2.7 Modelling using VDM Real-Time

The VDM-RT extensions to VDM support the description and analysis of real-time and distributed systems. They include primitives for modelling deployment over a distributed hardware architecture and support for asynchronous communication. Within a special **`system`** class, the modeller can specify computation resources (CPUs) connected in a communication topology by busses. Two predefined classes, `CPU` and `BUS` allow scheduling and performance characteristics of CPUs and busses to be readily expressed. The *system* class is a definition that groups an architectural model described using CPUs and busses with the instances that must be deployed onto that architecture.

The semantics of VDM have been extended with a notion of time so that any thread running on a computation resource and any message in transit on a communication resource can cause time to elapse. Each construct in the modelling language has a default time associated with it. Models that contain only one computation resource are compatible with models in plain VDM++.

Operations may be specified as asynchronous, allowing the caller to resume computation in its own

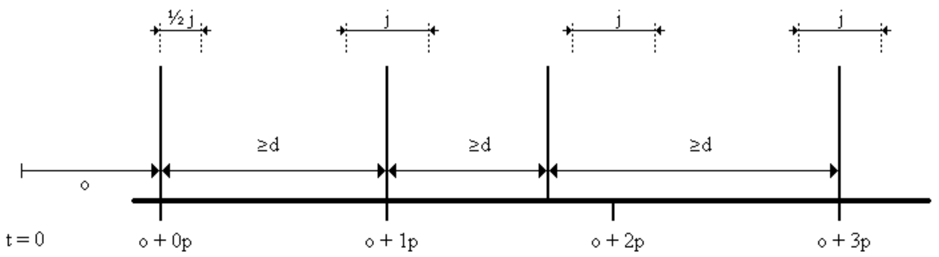

Figure 2.1: Period (p), jitter (j), delay (d) and offset (o)

thread immediately after the call is initiated. A new thread is created, automatically started and scheduled to execute the body of the asynchronous operation (without return values). Special (**duration** and **cycles**) statements may be used in operation bodies to specify time delays that are either independent of or dependent upon processor capacity. The time delay incurred by a message transfer over a bus can be made dependent on the size of the message being transferred and on the bandwidth of the bus.

The semantic model given for duration statements is compositional and enables validation of the runtime execution time. As a result, a top-down design approach can be used that delegates the implementation of sub-components to individual teams. The validation of runtime execution time checks the runtime execution time the body of a duration against the specified runtime of the duration to ensure that it not exceed the maximum allowed time. Furthermore, duration statements are also used as synchronization barriers in the semantic model. A thread synchronizes its transactional state when it completes a duration from its body. A more detailed description of durations is provided in Chapter 4.

In VDM-RT where time is explicit it is also possible to make threads periodic, so that their behaviour is repeated over time. The syntax of this is:

```
periodic (period, jitter, delay, offset)(op)
```

where:

period is a non-negative, non-zero value that describes the length of the time interval between two adjacent events in a strictly periodic event stream (where jitter = 0)

jitter is a non-negative value that describes the amount of time variance that is allowed around a single event. We assume that the interval is balanced [-j, j]. Note that jitter is allowed to be bigger than the period to characterize so-called event bursts.

delay is a non-negative value smaller than the period which is used to denote the minimum inter-arrival distance between two adjacent events.

offset is a non-negative value which is used to denote the absolute time value at which the first period of the event stream starts. Note that the first event occurs in the interval [offset,offset+jitter].

op is the operation that will be invoked in the object, in a new thread.

The relationship between the time-based fields in the *Periodic* construct is illustrated in Figure 2.1.

Chapter 3

Related Semantic Models

3.1 The Semantics of VDM-SL

The formal semantics of VDM-SL is included in the VDM-SL ISO standard [LHB^{+}96]. It is written in a denotational style based on basic set theory with least fixed point semantics for recursive definitions [LP95]. The domain universe for VDM-SL has been inspired by [TW90]; it provides denotations for all values expressible in VDM-SL. The meta-notation used for expressing the formal methods has itself be precisely described but here we refer the reader to [LP95] for an explanation of these. In this section we intend to give the reader a little insight into the style of the formal semantics of VDM-SL.

In traditional denotational semantics it is customary to provide a meaning function for each kind of syntactic component. Such a meaning function is a mapping from a syntax category to its meaning. This is done by means of a composition of the meaning of the components of the abstract syntax category. This means that in the case of a specification language with looseness, this approach would explicitly map the abstract syntax of the entire specification to the set of models which the specification denotes. However, this explicit style traditionally uses an order in which the definitions from the object language (in this case VDM-SL) must appear. In VDM-SL such an ordering is not defined and in general there can be mutual dependencies between definitions in different syntactic categories. The presence of looseness also makes definitions formulated with the explicit style more difficult to read [AL88]. Therefore the dynamic semantics of VDM-SL has been formulated in an implicit relational style instead of the traditional constructive style of denotational semantics.

3.1.1 *SemSpec* and *IsAModelOf*

The top-level function which gives meaning to a syntactic specification is defined as:

$$\mathit{SemSpec} : \mathit{Document} \to \mathbb{P}(\mathit{ENV})$$

$$\begin{array}{l} \mathit{SemSpec}(\mathit{doc}) \triangleq \\ \quad \{\, \mathit{env} \mid \mathit{env} \in \mathit{ENV} \cdot \mathit{IsAModelOf}(\mathit{env}, \mathit{doc}) \,\} \end{array}$$

“Candidate” models (also called environments) are taken from the set *ENV* which contains all maps from identifiers to possible denotations for VDM-SL constructs (including all values *VAL* and possible types, the socall domain universe). For any set A, $\mathbb{P}(A)$ denotes the powerset of A, i.e. the set of all subsets of A. *IsAModelOf* is a predicate which checks whether a given environment satisfies (in a formal sense) a given specification. If it does, it is called a *model* of the specification. The semantic function *SemSpec* for a given specification yields the set of all its models. The predicate *IsAModelOf* naturally needs to check whether all of the identifiers that have been defined in the specification are present in the environment. If this is the case, the environment is expanded with a number of constructs, with the definitions from the specification implicitly defined. Each component of the specification is now verified according to such an expanded environment.

Because the definitions can be mutually dependent upon constructs from different categories, this is done for each category (functions, types, operations, etc.) by a meaning function for that category. Here it is important that the denotation of the constructs from the category being verified is removed from the environment. The remaining part of the environment provides the context in which the meaning of the constructs in this category is to be found. In this way an order between the definitions becomes available because of the implicit style of definition. The rationale behind the removal of the constructs from the category being verified is that in case of mutual recursion between constructs in such a category the semantics of these constructs should not be affected by the denotations of those constructs in the candidate model.

3.1.2 Definers and Loose Definers

The meaning of the different kinds of definitions can be considered as an environment-to-environment transformation, adding more information to the environment. We call such transformations definers and loose definers (in case a construct can be potentially loosely specified). These can be explained by:

$$\begin{aligned}\mathrm{Def} &= \mathrm{ENV} \rightarrow (\mathrm{ENV} \cup \{\, \underline{\mathrm{err}} \,\}) \\ \mathrm{LDef} &= \mathbb{P}(\mathrm{Def})\end{aligned}$$

where $\underline{\mathrm{err}}$ is a special symbol indicating that in the given environment the syntactic definition cannot be given any sensible meaning.

Semantics of Constructs

Type definitions in VDM-SL are given a least fixed point semantics using the domain universe [Sch86]. No looseness is permitted in invariant expressions so types denote unique domain values from *DOM*.

Value (i.e. constant) definitions and function definitions can be loose, and the interpretation of this looseness will be discussed in the remaining part of this section. Mutually recursive definitions are given a least-fixed-point semantics unless they involve implicitly defined functions. In this case they are given an all-fixed-point semantics to keep all possible explicit definitions available. Functions can also be polymorphic, but for simplicity, we do not take that into account in this report.

Operation definitions can also contain looseness but here it is treated as non-determinism. Thus, an operation will denote a relation between input value (and state) and corresponding output value (and state). Implicitly defined operations are given an all-fixed-point semantics like for implicitly defined functions. The semantics of explicitly defined operations resembles a least-fixed-point semantics, but we cannot claim it to be so because there is no proper ordering between the operation denotations.

3.1.3 The Semantics of Looseness

We have mentioned the concept of 'looseness' a number of times above without being precise about its semantics. In this section explain how looseness can be interpreted.

Looseness can be interpreted in at least two different ways: as *under-determinedness* (allowing several different deterministic implementations) or as *non-determinism* (allowing non-deterministic implementations). As illustrated in [SS87, SS92] there are different types of behaviour that a non-deterministic semantics can exhibit, specifically demonic, angelic, and erratic. With the under-determined interpretation of looseness, functions are referentially transparent, as discussed in [SS88, SS90]. In VDM-SL, functions are given an under-determined semantics, while operations are given a non-deterministic semantics[1]. The complexity of an arbitrary combination of these can be found in [Wie89].

The difference between using the classical Hilbert epsilon operator [Lei69], the under-determined semantics and the non-deterministic semantics can be illustrated by a few examples. The expression:

```
(lambda v:nat & let x in set {1,2} in x)(5) =
    (lambda v:nat & let x in set {1,2} in x)(5)
```

is True in the Hilbert framework (using epsilon for the let-be expression) because the two choices from the same set must yield the same result. If we instead have two non-deterministic choices from the set, the comparison yields a non-deterministic choice between True and False. Considering the under-determined interpretation, we cannot use the same approach as Hilbert; choices in different parts of a program might be implemented differently even if they are made from sets that are equal.[2] However, due to the nature of the under-determined semantics, this is not a set of constant evaluators, since the choice of the resulting value (in this case either True or False) may of course depend on the argument environment, even when it does not depend syntactically upon the environment at all.

The difference between non-deterministic and under-determined semantics can be illustrated by another example. Consider the expression:

```
let func = lambda v:nat & let x in set {1,2} in x
in (lambda f: nat -> nat & f(5) = f(5))(func)
```

It yields True with the under-determined semantics because the two function applications yield the same result no matter which of the possible deterministic implementations of the function is considered. In a typical non-deterministic framework, non-deterministic implementations of the function would be allowed so the result would be a non-deterministic choice between True and False.

Also note that the first of the above examples is the result of β-reducing the second example. The two examples do, however, have different semantics, so β-reduction is *not* valid in general for VDM-SL functions. It is valid, however, if the argument evaluates, semantically, to a singleton set. For example, it would be valid when if it did not contain any uses of the let-be expression (or other constructs where looseness is introduced).

3.1.4 Internal versus External Looseness

For some systems, the behaviour should not be too precisely determined by the specification. The notion of looseness enables the designer to postpone certain decisions to a later stage of development (e.g. the final implementation stage). In general, looseness can be seen as a means to specify at a much higher level of abstraction than that of a final implementation, and in this way leave as much freedom as possible to the implementor.

The kind of looseness presented in the examples above is known as external looseness [HJ89] because the external behaviour of these expressions is not fully determined. This kind of external looseness is commonly used when the external behaviour must satisfy certain conditions that do not necessarily restrict the result to a unique value. We use the term external looseness when it is visible at a given abstraction level that different behaviour is permitted for a given specification.

[1] Using the terminology from the referenced papers, the kind of non-determinism used in operations in VDM-SL is strong, unbounded, erratic and loose non-determinism with singular binding.

[2] The rationale behind this is that even in cases where two functions are syntactically identical it is desirable that such loose functions can be implemented independently of each other.

Another kind of looseness is known as internal looseness. This may be used when the external behaviour of a system is defined to be deterministic, but freedom in some of the components of a specification is desirable. Such freedom can be used by the implementor to develop more efficient implementations of a given system. An obvious example of this would be an allocation of additional storage in a computer system. As a user of such storage we do not care about its physical location as long as we can use it freely (and possibly release it again later). Design decisions about the storage management inside a larger system could be left open by using looseness, but it would (hopefully) not be visible in the external behaviour of our system. Note that the given abstraction level is essential for this distinction, because the actual allocation function naturally is externally loose (at the function level), but if we look at the system level (and consider the allocation function to be hidden inside) the looseness is only visible internally.

3.1.5 Semantics of Expressions

An environment associates identifiers with values. Expression evaluation can be described as replacing each identifier in the expression by its value from the environment, and computing the result. Thus, the value of an expression is dependent on the environment in which it is being evaluated. So, one could take the type of the evaluation function for expressions (as used in [Mon85]) as:

$$\mathit{EvalExpr} : \mathit{Expr} \rightarrow \mathit{ENV} \rightarrow \mathit{VAL}$$

However, because expressions can be loose, it is possible for an expression to yield more than one value. A next attempt (used in [AL88] and [LAMB89]) could be:

$$\mathit{EvalExpr} : \mathit{Expr} \rightarrow \mathit{ENV} \rightarrow \mathbb{P}(\mathit{VAL})$$

Unfortunately this leads to very serious problems with the least-fixed-point semantics for recursive loose definitions. Therefore, the type of the evaluation function *EvalExpr* must ultimately be:

$$\mathit{EvalExpr} : \mathit{Expr} \rightarrow \mathbb{P}(\mathit{ENV} \rightarrow \mathit{VAL})$$

where the looseness has been abstracted ‘one level up’. We can now talk about deterministic expression evaluators for which least fixed points can be found. An expression will denote a set of such expression evaluators, which we call a loose expression evaluator. Because this technique is used for all kinds of expressions we define:

$$\begin{aligned} &\mathrm{EEval} &&= \mathrm{ENV} \rightarrow \mathrm{VAL} \\ &\mathrm{LEEval} &&= \mathbb{P}(\mathrm{EEval}) \end{aligned}$$

Sub-functions of *EvalExpr* are defined for all expression kinds. These functions are all written in roughly the same style:

$$
\begin{array}{l}
\mathit{EvalAnExpr} : \mathit{AnExpr} \to \mathit{LEEval} \\[1ex]
\mathit{EvalAnExpr}(\mathit{MkTag}(\text{‘}\mathit{AnExpr}\text{’}, (\mathit{expr}_1,\ldots,\mathit{op},\ldots,\mathit{expr}_n))) \triangleq \\
\quad \{\, \lambda \mathit{env} \,.\, \textbf{let}\ \mathit{val}_1 = \mathit{ev}_1(\mathit{env}), \\
\qquad\qquad\qquad \ldots, \\
\qquad\qquad\qquad \mathit{val}_n = \mathit{ev}_n(\mathit{env})\ \textbf{in} \\
\qquad\qquad\quad \mathit{AnOp}(\mathit{val}_1,\ldots,\mathit{val}_n,\mathit{op}) \\
\quad \mid \mathit{ev}_1 \in \mathit{EvalExpr}(\mathit{expr}_1),\ldots,\ \mathit{ev}_n \in \mathit{EvalExpr}(\mathit{expr}_n) \,\}
\end{array}
$$

where the function *AnOp* is the mathematical definition of the actual operation that occurs in *AnExpr*. *MkTag* is an abstract syntax level operator tagging syntactic constructs with the name of their ‘types’, *AnExpr* in this case.

3.2 The Semantics of VDM++

The formal semantics of VDM++ was first created in the European ESPRIT-III project Afrodite (Project number 6500) [DK92, De95, DGP94]. The intent was that the semantics of constructs from VDM-SL should be unchanged. An axiomatic semantics was provided in [KM93]. The concurrency aspects of VDM++ (introduced in Section 2.6 above) was given semantics in [Lan94] using Real-Time Logic (RTL) [Mok91]. However, VDM++ has changed substantially since that time and the only relatively complete semantics of VDM++, is a VDM-SL specification for the executable subset of VDM++: it is part of the specification used for the VDMTools interpreter [FLS08]. Unfortunately this document is not publicly available. The VDM interpreter inside Overture deterministically selects one of the mathematical models [LL91, LLB11]. Note that a significant difference here to the VDM-SL semantics mentioned above is that, because this is used for interpretation of specifications, instead of taking all the semantic models for the specification into account, it restricts itself to just a single one of these. It is theoretically possible to explore all such models from the executable subset but this is only of academic value [Lar94].

3.3 The Semantics of VDM-RT

As explained in Section 2.7, VDM-RT introduces a few concepts that affect the operational semantics of standard VDM++. These include:

1. A discrete clock that represents the progress of time in the entire VDM-RT model. This clock is usually referred to as the “wall clock”. The wall clock may have an arbitrary resolution, also called a “clock tick”, but is set to be 1 nanosecond in VDM-RT.
2. The ability to construct an explicit system architecture on which functionality can be deployed. For this purpose, the **`system`** construct is introduced in the syntax of VDM-RT and `CPU` and `BUS` classes are provided as first-class language citizens. The capacity, scheduling policy and task switch (or protocol) overhead of both architectural elements can be specified. The capacity of a CPU is defined by the number of clock ticks required to execute a single “CPU tick”. A CPU tick represents the time required to perform the fastest instruction on the CPU.
3. A default time cost is associated with each basic VDM construct. This cost is expressed as a positive integer, representing the number of CPU ticks taken to execute each instruction.
4. Time costs can be redefined using the **`duration`** and **`cycles`** statements. The **`duration`** statement can be used to define absolute time costs, while the **`cycles`** statement can be used to set the time penalty relative to the performance of the CPU on which the functionality is deployed.

Any state changes that are a result of computation are not made visible to other threads or resources until the time required for the state change has passed. Then the state change is committed and becomes visible to other threads, as the internal record of time is updated and time-related bookkeeping is dealt with.

The VDM-RT semantic model given in this report is underspecified with respect to some of the standard VDM++ constructs. There are cases where the standard behaviour of VDM++ is not appropriate for VDM-RT, and this is described in [LVLW10]. These cases include the following:

- Static access to variables in a distributed setting.
- Static operation calls in a distributed setting.
- Read of distributed variables without a bus.

The core issue is that the distributed nature of variables and calls would be ignored if the VDM++ semantics were directly adopted. Unfortunately, this is the case in the current implementation of the VDM-RT interpreter. The present solution is to disallow static access to variables, and to force cross-CPU reads of variables to use bus communication.

Chapter 4

Semantics of VDM-RT

4.1 Overview of Structure & Entities

To describe the VDM-RT semantic model we start with an overview of its static structure, giving the entities used in the semantic description, then we describe the semantic model's behaviour.

The entities used to describe the VDM-RT semantics form a hierarchy starting with the $VDMRT$ structure. At the top level, the $VDMRT$ structure records the CPUs in the system ($cpus$), the busses connecting the CPUs ($busses$), the current time that the model has reached ($time$), and the defined classes in the model ($classes$).

$$\begin{array}{lrcl} VDMRT :: & cpus & : & CPUs \\ & busses & : & Busses \\ & time & : & Time \\ & classes & : & Classes \end{array}$$

$$CPUs = Id_c \xrightarrow{m} CPU$$

$$Busses = Id_b \xrightarrow{m} Bus$$

$$Classes = Id_{cl} \xrightarrow{m} Class$$

The types Id_x where x is one of b, c, cl, f, o, op, v represent disjoint sets of identifiers for, respectively, busses, cpus, classes, functions, objects, operations, and variables. These identifier sets are arbitrarily large and, for the purposes of the semantics, inexhaustible. The use of disjoint sets allows a simplification in the semantics when a fresh identifier is needed.

CPUs in the VDM-RT semantics are a record of three fields: a map for the instantiated classes ($objects$), a map for all threads that exist over the course of execution ($threads$), and a natural number representing the speed of the CPU.

$$\begin{array}{lrcl} CPU :: & objects & : & Id_o \xrightarrow{m} Object \\ & threads & : & Id_t \xrightarrow{m} Thread \\ & speed & : & \mathbb{N}_1 \end{array}$$

The topology of connections between CPUs is recorded in the $busses$ map in the $VDMRT$ record; each bus connects a (non-strict) subset of CPUs from the $VDMRT$ record's $cpus$ field. A single bus records the set of CPUs it connects ($cpus$); a natural number that represents the speed of communication over the bus, typically much lower than the speed of a CPU (*speed*); and a queue of call and return messages, tagged with the target CPU.

$$\begin{array}{lrcl} Bus :: & cpus & : & Id_c\text{-}\mathbf{set} \\ & speed & : & \mathbb{N}_1 \\ & queue & : & (Id_c \times (CMessage \mid RMessage))^* \end{array}$$

We record the notion of states, Σ, as mappings from variable identifiers to VDM values.

$\Sigma = Id_v \xrightarrow{m} VDMValue$

The $Class$ structure contains all of the static detail of a class (as opposed to the $Object$ structure, below, that contains the dynamic details of instantiated classes). In this structure we record super classes ($parents$), constant values ($values$), the types of instantiated variables ($vars$), the set of associated operations and functions of the class (ops and $funs$, respectively), a set of class invariant functions ($invs$), and the initial action ($initial$) that an instantiated class should perform when it is started.

The behaviour of class records in this semantic model is such that, though they reference their parent classes initially, the post-initialization class record will be changed to becomes the "union" of it and all of its parent classes. So in the initialization step of the semantic model all of the defined classes in a subject model will be "flattened" into independent classes.

$$\begin{aligned} Class :: \; parents &: Id_c^* \\ values &: \Sigma \\ vars &: Id_v \xrightarrow{m} Type \times Expr \\ ops &: Id_{op} \xrightarrow{m} Op \\ funs &: Id_f \xrightarrow{m} Fun \\ invs &: Fun\text{-}\textbf{set} \\ initial &: Duration^* \mid Periodic \end{aligned}$$

Instantiated classes are represented by the $Object$ record. These structures contain only a reference to the static class details ($class$), the current state of variables in the class ($state$) and, optionally, a value giving a countdown until the next time a periodic thread needs to be created in the context of that object ($periodicCountdown$). This countdown value is pre-calculated every time a periodic thread is launched, based on the values in a $Periodic$ record in the $initial$ field of the class definition.

$$\begin{aligned} Object :: \; class &: Id_{cl} \\ state &: \Sigma \\ periodicCountdown &: [Time] \end{aligned}$$

The $Thread$ structure records a thread's current status ($status$), the values of variables in objects that are held aside pending a commit ($pending$), the object that gives the current execution of the thread ($context$), and the remaining statements to be executed by the thread ($body$). When the value of a variable has been changed by a thread but not yet committed, the new value is kept aside in the thread's $pending$ field until a certain amount of time has passed; this behaviour is described in Section 4.6.

$$\begin{aligned} Thread :: \; status &: \text{RUNNING} \mid \text{RUNNABLE} \mid \text{WAITING} \mid \text{PENDING} \mid \text{COMPLETED} \\ pending &: Pending \\ context &: Id_o \\ body &: (Duration \mid PartialDuration)^* \end{aligned}$$

The $PartialDuration$ and $Duration$ statements that comprise the $body$ of the thread are used to indicate the expected execution time of the contained block of statements. A $Duration$ statement has a $duration$ field that represents the expected execution time bound, and a $body$ field containing the sequence of statements to be executed. The expected time bound may be an expression that initially needs to be calculated, but it will be a constant value when the body actually starts execution; alternatively, the $duration$ may be EXECTIME, indicating that even though this duration has no time bound, recording the execution time is still necessary. A $Duration$ structure becomes a $PartialDuration$ structure if the execution of the duration's body cannot be completed during one step of the interpreter's execution. As an example, this will happen if the body invokes a synchronous operation in an object on a different CPU.

$$\begin{aligned} Duration :: \; duration &: \text{EXECTIME} \mid Exp \\ body &: Stm \end{aligned}$$

$$\begin{aligned} \mathit{PartialDuration} :: \mathit{duration} &: \textsc{ExecTime} \mid \mathit{Time} \\ \mathit{elapsed} &: \mathit{Time} \\ \mathit{body} &: \mathit{Stm} \end{aligned}$$

The atomicity of the outermost $\mathit{PartialDuration}$s and $\mathit{Duration}$s in a thread is all-or-nothing, but only so long as no operations are invoked on remote CPUs. If an operation is invoked on a remote CPU then the data associated with the parameters will be sent outside of the scope of the $\mathit{Duration}$; this allows intermediate data to 'leak' out of the duration and thus destroys the atomicity of the $\mathit{Duration}$ block. Atomicity in the sense of instantaneous execution is possible in the semantics by using a zero in the time field of a $\mathit{Duration}$; however, in the concrete language a zero value in the time field becomes a EXECTIME value in the semantics. This means that it is not actually possible to specify instantaneous durations.

The $\mathit{Duration}$ structure is included in the Stm type and, therefore, statements can contain nested durations. The behaviour of nested durations, and durations in general, is described in Section 4.7.

4.1.1 Durations and Transaction Synchronization

The $\mathit{Duration}$ record is used as a synchronization point in the semantics. When a $\mathit{Duration}$ construct has been completely evaluated the thread state is updated with all pending values that were calculated during the execution of the duration. The $\mathit{commitPendingValuesAndUpdateTime}$ function performs this update, and is found in the first hypothesis of the Big Step rule. The behaviour of thread state update is described in Section 4.6.

The overall time represented by each step of the whole semantic model is calculated for the next step at the top level, based on the next expected commit of pending values and the next expected start of a periodic thread.

4.1.2 Duration Composability

The semantic model of VDM-RT models time using semantic durations that are compositional. This allows a top-down specification approach where the time of sub-components are added together, thus enabling validation of runtime execution of durations. Validation checks that sub-durations do not exceed the given value of their containing duration.

The interpreter for VDM-RT described in [LLB11] implements a different semantic model that is non-compositional, and does not support a top-down design approach with respect to the time specifications of durations. Further, the interpreter does not implement any runtime validation of durations and it simply ignores the time values of nested durations. The result of this is that sub-components cannot easily be given to independent teams to implement without providing information outside the specification itself.

Consider the case of a duration with a 5 second time bound, and this duration has two sub-durations, composed sequentially, each with a 4 second time bound. In the semantic model described in [LLB11], this is valid, and the overall duration still has a 5 second time bound as the bounds of the sub-durations are ignored.

That semantic model leads to the problem that it is difficult to hand the specifications of the sub-durations off to implementor without changing their specification; clearly, to satisfy the overall duration's 5 second limit, the sub-durations must always complete in a time which sums to less than 5 seconds. However, their specifications allow for a sum of 8 seconds. Use of this semantic model means that, for the purposes of top-down design, the specifications are incomplete and require that additional information be known to the user of the duration blocks.

The semantic model presented in this report adds the validation of duration time bounds into the semantic model. This has the effect of making the specifications complete for the purposes of durations.

A sub-duration can be decomposed out of a larger one and implemented according to the given sub-specification without the need of additional knowledge.

4.2 Top-level Execution Rule

Sufficient structure has been described so far to move on to the behavioural rules of VDM-RT; the full set of structures and rules are found in Appendix A. The top-level rule, Big Step in Figure 4.1, gives the whole semantics of a running VDM-RT model[1]. There are six hypotheses to this rule, and each represents a phase in an execution step.

1. The first hypothesis is the internal update of the model state. It updates the model's present time, and then commits all of the pending values held in threads up to that point. Races between updates –where two or more threads would update the same variable– are handled in a non-deterministic manner, and no particular resolution mechanism is specified in this semantic model.

   A thread only has its pending values committed when the head of the thread's body is not a $PartialDuration$: this indicates that any previous duration had completed and any values in pending are ready to be made visible. This hypothesis also serves to decrement the $periodicCountdown$ fields of those objects that use it.

2. The second hypothesis is in the form of a transition relation as the action of the busses is inherently non-deterministic. This phase actually delivers messages on the busses to their target CPUs if sufficient time has passed. Where the message is an operation call, a new thread will be created on the target CPU to run the operation.

3. The third hypothesis potentially creates more new threads based on the timing of periodic threads. If there are objects with $periodicCountdown$ fields that have reached zero then the appropriate new threads are created for those and the $periodicCountdown$ field is recalculated based on the $Periodic$ record in that object's class definition.

4. The fourth hypothesis performs any potential context switches, allowing a CPU to change from one running thread to another. Note that this phase happens after the creation of new threads so that those new threads have the potential to start execution within this step of the execution.

5. The fifth hypothesis is also a transition relation, as the execution of VDM-RT statements may be non-deterministic. This transition attempts to execute the first duration of the body of every active, running thread in the model. The number of threads attempted will be no greater than the number of CPUs in the system, as each CPU may only execute in one thread per step. If it is not possible to fully execute the duration at the head of a thread's body, then a $PartialDuration$ will replace that duration on the head of the body. The partial duration will have the remainder of the original duration's body that remains to be executed, and execution will continue during the next step that the thread is active. This hypothesis also exposes the minimum time until the next commit of pending values.

6. The sixth hypothesis calculates the time at which the next action in the interpreter must happen. This may be due to things such as threads with pending variables, the creation of a new periodic thread, and so forth. This results in the minimum amount of the time until the next action that the interpreter handle and this serves as τ for the next Big Step.

[1] Note that for readability purposes the central rules have numbered hypothesis lines to allow for easier reference in the explanation.

$\boxed{\text{Big Step}}$

$$\begin{array}{lr}
vdmrt_1 = commitPendingValuesAndUpdateTime(vdmrt, \tau) & (1) \\
vdmrt_1 \xrightarrow{busses} vdmrt_2 & (2) \\
vdmrt_3 = createPeriodicThreads(vdmrt_2) & (3) \\
vdmrt_4 = doContextSwitches(vdmrt_3) & (4) \\
vdmrt_4 \xrightarrow{exec} (vdmrt_5, \tau_b) & (5) \\
\tau_b' = min(\tau_b, minPendingCommitTime(vdmrt_5)) & (6) \\
\hline
(vdmrt, \tau) \xrightarrow{vdmrt} (vdmrt_5, \tau_b') &
\end{array}$$

Figure 4.1: Definition of the Big Step rule.

$\boxed{\text{Init}}$

$$\begin{array}{lr}
cpus = createBareCPUs(demodel) & (1) \\
busses = createBusses(cpus, demodel) & (2) \\
classes = createClasses(demodel) & (3) \\
cpus' = createInitialInstances(cpus, classes, demodel) & (4) \\
\hline
(vdmmodel) \xrightarrow{init} mk\text{-}VDMRT(cpus', busses, 0, classes) &
\end{array}$$

Figure 4.2: Definition of the VDM-RT initialization rule.

An execution trace based on the Big Step rule shown above and the Init from Section 4.3 would look like this:

$$\begin{array}{l}
(VDMRTModel) \xrightarrow{init} (vdmrt) \\
(vdmrt, 0) \xrightarrow{vdmrt} (vdmrt_1, \tau_1) \xrightarrow{vdmrt}* (vdmrt_n, \tau_n)
\end{array}$$

4.3 Initialization

Before the main portion of the VDM-RT semantics applies, we must deal with the creation of a $VDMRT$ construct based on an input model and contract. The initialization inference rule, Init, has the type

$$\xrightarrow{init}: \mathcal{P}((VDMRTModel) \times (VDMRT))$$

where the $VDMRTModel$ is the input model (the sources of the model).

The first two hypotheses of the Init rule in Figure 4.2 deal with bare CPU creation ($createBareCPUs$) and the bus creation ($createBusses$); the CPUs are given as an additional argument along with the input model as the busses record the links between CPUs. The class mapping is created in the third hypothesis ($createClasses$) which includes the copying of all definitions from any parent classes to the actual classes and thus resolving any inherited definitions. The $classes$ is used as a parameter in the fourth hypothesis to populate the mapping of bare CPUs with the initial instances of objects defined in the input model ($createInitialInstances$); this populated mapping is stored within a new CPU map. Finally, in the conclusion these components are combined into a $VDMRT$ construct (with its initial time set to zero) that is ready for use in the main portion of the VDM-RT semantics.

4.4 Operation Calls

This semantics describes four types of operation calls: the combination of synchronous/asynchronous and local/remote. Synchronous calls require that the caller wait for the called operation to complete before it continues, whereas the caller of an asynchronous operation continues execution without waiting for the call to complete. Local calls take place completely on a single CPU, whereas remote calls require the use of the busses in the model to cause the operation to execute on a different CPU, and the caller may wait for a return message indicating that the called operation has completed.

Same-CPU synchronous calls continue execution in the same thread by inserting the content of the called operation at the head of that thread's body; this is done to avoid the non-deterministic properties of a thread context switch. All other calls create a new thread (on the appropriate CPU) to execute the content of the called operation; the original thread (eventually) continues execution as it is. In the semantic model calls are represented as a type union of $\mathit{SyncCall}$ and $\mathit{AsyncCall}$:

$$\mathit{Call} = \mathit{SyncCall} \mid \mathit{AsyncCall}$$

Both $\mathit{SyncCall}$ and $\mathit{AsyncCall}$ have nearly the same structure: since asynchronous calls do not return a value, those operations do not need a target for any such value. Shown here is the definition of the $\mathit{SyncCall}$ construct; the $\mathit{AsyncCall}$ construct omits the target field:

$$\begin{aligned} \mathit{SyncCall} :: \mathit{target} &: \left[\mathit{Id}_v \mid (\mathit{Id}_c \times \mathit{Id}_t)\right] \\ \mathit{name} &: \mathit{Id}_o \times \mathit{Id}_{op} \mid \mathit{Id}_c \times \mathit{Id}_o \times \mathit{Id}_{op} \\ \mathit{args} &: \mathit{Expr}^* \end{aligned}$$

where the target field records the destination for the return value from the call in the calling thread's $\mathit{pending}$ map, if it is to be kept; the name field identifies the operation or function which should be called and finally args are the argument expressions of the call.

It is notable both the target and name fields of the $\mathit{SyncCall}$ construct are actually union types. For the name field, the reference operation may not be on the same CPU as the calling thread; hence the need for the union type. When the object which has the operation to be called is not on the current CPU we must also record the CPU identifier referencing the remote CPU. The values for the target field are similar: in the simple case the field may simply be a variable identifier; in the complex case the field takes on the pair of a CPU and a thread identifier that references the thread that originally invoked the operation, as this case corresponds to a remote synchronous operation call.

To handle return values the semantics makes use of two constructs, the first of which is for use in the operation bodies, Return:

$$\mathit{Return} :: \mathit{exp} : \left[\mathit{Exp}\right]$$

It has a single expression that is evaluated to a $\mathit{VDMValue}$ in the context of the call body. The $\mathit{VDMValue}$ type represents all literals and is a subset of syntactic expressions. The Return construct is used to evaluate the expression in the correct context where it is rewritten by replacing the original return expression with a $\mathit{VDMValue}$ that then later can be matched up with a Wait record.

The second record used to link the calling thread with the return of the call body is Wait:

$$\mathit{Wait} :: \mathit{target} : \mathit{Id}_v \mid (\mathit{Id}_c \times \mathit{Id}_t)$$

where the target field holds the identifier that will be assigned the eventual return value. Note that the target field of the Wait construct takes on the same union type as the target field of the $\mathit{SyncCall}$ construct. This allows us to use a Wait construct in the calling thread regardless of whether the call was local or remote. When a synchronous remote operation call reaches its target CPU, it is instantiated as a $\mathit{SyncCall}$ construct with the identifiers of the calling CPU and thread, and as this is now a local synchronous call, the eventual Wait construct uses those identifiers as its target. This allows the semantics to determine that a return message must be sent across the bus to the calling CPU, where a new Return construct will be created and ultimately resolved with the local Wait construct.

The initial setup resulting from a $\mathit{SyncCall}$ on single CPU is defined by the inference rule Stmt Call Op Local Sync shown in Figure 4.4. The whole process of execution is illustrated abstractly in Figure 4.3, where the body of a thread contains a $\mathit{SyncCall}$ at the head. The first step of the execution is to match

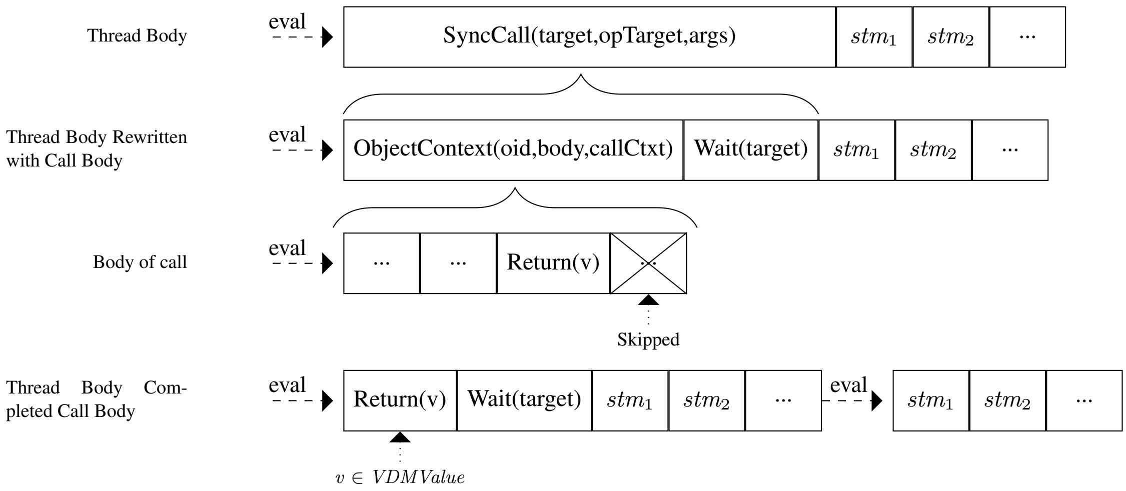

Figure 4.3: Illustration of the semantic evaluation of a synchronous local call.

the $SyncCall$ and rewrite it as an $ObjectContext$ followed by a $Wait$ construct; the $Wait$ can then later be used to match the $Return$ of the body. The $ObjectContext$ construct has the form:

$$\begin{aligned} ObjectContext \;::\; object &: Id_o \\ body &: Stm \\ callCtx &: CallContext \end{aligned}$$

where the $object$ field refers to the object in which the $body$ must be executed, and $callCtx$ contains a $CallContext$ construct that records information about the call. The $CallContext$ construct has the form:

$$\begin{aligned} CallContext \;::\; pending &: Pending \\ state &: \Sigma \\ post &: [Expr] \end{aligned}$$

The Stmt Call Op Local Sync rule shown in Figure 4.4 applies to $SyncCall$ constructs. It evaluates the call's arguments in the calling context (line 2), and records the time taken to evaluate the arguments at line 13, where the total time of the operation is summarized. The pre-condition is checked in the calling context at line 6 using the actual values for the arguments: this is followed by the creation of a new $ObjectContext$ in lines 7–9; this is then concatenated with a $Wait$ statement in line 10, where the $Wait$ statement specifies the target of the $SyncCall$ used to store the operation's return value. The $callBlock$ defined in line 10 is prefixed to the remainder of the current thread's body in line 11, and in line 12 the remainder of the thread's body is executed.

The purpose of the $CallContext$ structure is to record state information for a call such that post-conditions can be evaluated at the completion of a call. The $pending$ field holds the $pending$ state of the calling thread at the moment the operation is called; $state$ is a map holding the evaluated arguments; and the optional post-condition comes from the operation being called. Then the body of the call is executed, and is stored in the $ObjectContext$. The inference rules Stmt ObjectContext Step (see the Appendix) and Stmt ObjectContext Complete (Figure 4.7) execute all of the statements in the body of the object context until a $Return$ statement has been fully evaluated by the Stmt Return Eval rule shown in Figure 4.5. The Stmt Return Eval rule checks that return value is an expression (line 1), and then evaluates this in the current context (line 2). This is followed by a rewrite of the $Return$ where the expression is replaced with the actual value (line 3). The evaluation is continued in line 4 and finally the total time used is calculated in line 5.

Any remaining statements present in the sequence after the $Return$ construct will be removed by

$\boxed{\text{Stmt Call Op Local Sync}}$

$$
\begin{array}{ll}
\mathit{opTarget} = (\mathit{oid}, \mathit{op}) & (1)\\
\mathit{argsTimed} = [(\mathit{value}, \delta_e) \mid \mathit{arg} \in \mathit{args} \wedge (\mathit{classes}, \mathit{cpus}, \mathit{pending}, o \vdash \llbracket e \rrbracket = (\mathit{value}, \delta_e))] & (2)\\
\mathit{args} = [\mathit{value} \mid (\mathit{value}, \text{-}) \in \mathit{argsTimed}] & (3)\\
\mathit{mk\text{-}Op}(\text{-}, \mathit{params}, \mathit{ret}, \mathit{body}, \mathit{pre}, \mathit{post}) = \mathit{classes}(\mathit{cpu}.\mathit{objects}(\mathit{oid}).\mathit{class}).\mathit{ops}(\mathit{op}) & (4)\\
\sigma = \{p \mapsto a \mid i \in \textbf{inds}\, \mathit{args} \wedge a = \mathit{args}(i) \wedge \mathit{params}(i) = (p, \text{-})\} & (5)\\
\mathit{checkCallPre}(\mathit{classes}, \mathit{cpus}, \mathit{pending}, \mathit{args}, \mathit{params}, \mathit{oid}, \mathit{pre}) = \textbf{true} & (6)\\
\mathit{callContext} = \mathit{mk\text{-}CallContext}(\mathit{pending}, \sigma, \mathit{post}) & (7)\\
\mathit{partialLetDef} = \mathit{mk\text{-}PartialLetDef}(\sigma, [\,], \mathit{mk\text{-}SimpleBlock}(\mathit{body})) & (8)\\
\mathit{objContext} = \mathit{mk\text{-}ObjectContext}(\mathit{oid}, \mathit{partialLetDef}, \mathit{callContext}) & (9)\\
\mathit{callBlock} = [\mathit{objContext}, \mathit{mk\text{-}Wait}(\mathit{target})] & (10)\\
\mathit{stms} = \mathit{callBlock} \curvearrowright \mathit{rest} & (11)\\
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash & \\
\qquad (\mathit{stms}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta_{\mathit{rest}}) & (12)\\
\delta' = \mathit{sum}([\delta_e \mid (\text{-}, \delta_e) \in \mathit{argsTimed}]) + \delta_{\mathit{rest}} + \mathit{LocalSyncCallTime} & (13)\\
\hline
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash & \\
([\mathit{mk\text{-}SyncCall}(\mathit{target}, \mathit{opTarget}, \mathit{args})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} & \\
\quad (\mathit{rest}'', \mathit{pending}'', \mathit{cpu}'', \mathit{busses}'', \delta') &
\end{array}
$$

Figure 4.4: Definition of the Stmt Call Op Local Sync rule.

$\boxed{\text{Stmt Return Eval}}$

$$
\begin{array}{ll}
\mathit{exp} \notin \mathit{VDMValue} & (1)\\
\mathit{classes}, \mathit{cpus}, \mathit{pending}, o \vdash \llbracket \mathit{exp} \rrbracket = (\mathit{retValue}, \delta_e) & (2)\\
\mathit{rest}' = [\mathit{mk\text{-}Return}(\mathit{retValue})] \curvearrowright \mathit{rest} & (3)\\
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash & \\
\qquad (\mathit{rest}', \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) & (4)\\
\delta' = \delta_e + \mathit{ReturnTime} & (5)\\
\hline
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash & \\
([\mathit{mk\text{-}Return}(\mathit{exp})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta') &
\end{array}
$$

Figure 4.5: Definition of the Stmt Return Eval rule.

$\boxed{\text{Stmt Return Eat}}$

$$
\begin{array}{ll}
\mathit{rest} \neq [\,] & (1)\\
\textbf{hd}\, \mathit{rest} \notin \mathit{Wait} & (2)\\
\mathit{rest}' = [\mathit{mk\text{-}Return}(v)] \curvearrowright \textbf{tl}\, \mathit{rest} & (3)\\
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash & \\
\qquad (\mathit{rest}', \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) & (4)\\
\hline
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash & \\
([\mathit{mk\text{-}Return}(v)] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) &
\end{array}
$$

Figure 4.6: Definition of the Stmt Return Eat rule.

$$\boxed{\text{Stmt ObjectContext Complete}}$$

$$
\begin{array}{lr}
\tau, \mathit{classes}, \mathit{cpus}, c, t, \mathit{oid} \vdash & \\
\qquad ([\mathit{body}], \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) & (1)\\
\mathit{rest}' = [\,] \vee (\textbf{hd}\, \mathit{rest}' = [\mathit{mk\text{-}Return}(v)] \wedge v \in \mathit{VDMValue}) & (2)\\
\mathit{cctx} = \mathit{mk\text{-}CallContext}(\mathit{prepending}, \mathit{args}, \mathit{post}) & (3)\\
\mathit{checkCallPost}(\mathit{classes}, \mathit{cpus}, \mathit{oid}, \mathit{prepending}, \mathit{pending}', \mathit{args}, v, \mathit{post}) = \textbf{true} & (4)\\
\mathit{rest}'' = \mathit{rest}' \curvearrowright \mathit{rest} & (5)\\
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash & \\
\qquad (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}') \xrightarrow{\mathit{stmt}} (\mathit{rest}''', \mathit{pending}'', \mathit{cpu}'', \mathit{busses}'', \delta') & (6)\\
\delta'' = \delta + \delta' & (7)\\
\hline
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash & \\
([\mathit{mk\text{-}ObjectContext}(\mathit{oid}, \mathit{body}, \mathit{cctx})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} & \\
\quad (\mathit{rest}''', \mathit{pending}'', \mathit{cpu}'', \mathit{busses}'', \delta'') &
\end{array}
$$

Figure 4.7: Definition of the Stmt ObjectContext Complete rule.

$$\boxed{\text{Stmt Return Wait}}$$

$$
\begin{array}{l}
\mathit{target} \in \mathit{Id}_v\\
\sigma' = \mathit{pending}(o) \dagger \{\mathit{target} \mapsto v\}\\
\mathit{pending}' = \mathit{pending} \dagger \{o \mapsto \sigma'\}\\
\tau, \mathit{classes}, \mathit{cpus}, c, t, \mathit{obj} \vdash (\mathit{rest}, \mathit{pending}', \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}'', \mathit{cpu}', \mathit{busses}', \delta)\\
\hline
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash\\
([\mathit{mk\text{-}Return}(v), \mathit{mk\text{-}Wait}(\mathit{target})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}}\\
\quad (\mathit{rest}', \mathit{pending}'', \mathit{cpu}', \mathit{busses}', \delta)
\end{array}
$$

Figure 4.8: Definition of the Stmt Return Wait rule.

the Stmt Return Eat rule as shown in Figure 4.6. The rule checks that the head is a *Return* statement, and that the first statement of the remaining statements (the *rest* meta-variable) is not a *Wait* statement. If this is satisfied then the *Return* statement and the tail of the rest (line 3) is used to continue execution (line 4). When the *ObjectContext* is fully evaluated –containing only a *Return* construct that has a *VDMValue*– the post-condition is checked (line 4) and the entire *ObjectContext* is removed (line 5) in the Stmt ObjectContext Complete rule shown in Figure 4.7, leaving in its place just the contained *Return* construct.

The last step is to match that *Return* construct with the corresponding *Wait* construct that was created when the operation was called, and then insert the value returned into the thread's *pending* field; this is defined in the inference rule Stmt Return Wait as shown in Figure 4.8. Where the return value is mapped by the *target* in the executing object.

The evaluation of *AsyncCall* on the same CPU differs from the *SyncCall* in that it creates a new thread to execute the called operation's body and simply removes the *AsyncCall* from the head of the active thread's body.

Remote, cross-CPU calls –where the target operation is in an object on a different CPU– differ from local-CPU calls in that they must use a bus to communicate the call request and return values. The communications are queued on the bus as messages, where call requests are recorded as *CMessage* constructs and return messages are *RMessage* constructs. The *CMessage* construct is defined as:

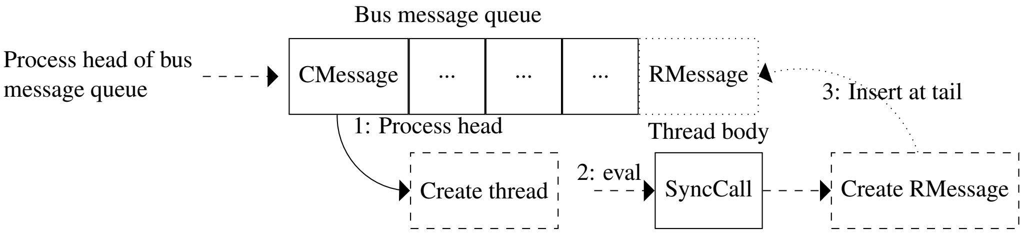

Figure 4.9: Illustration of the semantic evaluation of a $CMessage$ from the bus queue.

$$\begin{array}{lrcl} CMessage :: & obj & : & Id_o \\ & op & : & Id_{op} \\ & args & : & VDMValue^* \\ & replyto & : & [Id_c \times Id_t] \\ & sendTime & : & Time \end{array}$$

where obj is the target object; op the target operation in that object; $args$ is a sequence of arguments evaluated in the sender's context to $VDMValue$s; $replyTo$ identifies the CPU and thread of the caller and $sendTime$ is the time when the message was placed on the bus. The $sendTime$ field is used to calculate the time at which the message arrived at the remote CPU. The $RMessage$ construct is defined as:

$$\begin{array}{lrcl} RMessage :: & value & : & VDMValue^* \\ & replyto & : & Id_c \times Id_t \\ & sendTime & : & Time \end{array}$$

where $value$ is the sequence of $VDMValue$s being returned. The fields $replyTo$ and $sendTime$ have the same purpose as in the $CMessage$ construct.

A remote $SyncCall$ places a $CMessage$ on the appropriate bus and changes the thread status to WAITING (thus excluding the thread from normal scheduling), instead of evaluating the call body locally; this is described in the Stmt Call Op Remote Sync rule. The $CMessage$ is matched by the inference rule Bus Call, during the next step of the interpreter. This is illustrated abstractly in Figure 4.9 where processing of the head of the bus queue checks first that the message has arrived (by the use of message's $sendTime$ field) and then, for a $CMessage$ construct, a new thread is created in the appropriate object where the call will be executed. Eventually, the return value is enqueued back on the bus queue in a $RMessage$ construct, targeted at the calling CPU. When an $RMessage$ is processed, a $Return(v)$ statement is pushed onto the head of the original calling thread's body, immediately in front of the existing $Wait$ construct. This allows the Stmt Return Wait rule to complete the call and handle the return value in the same way as a local sync call would. Asynchronous remote calls are initiated in the same way as synchronous calls, except that the thread status of the calling thread is not changed, and no value is returned.

4.5 Periodic Threads

One of the phases in a step of the execution involves handling the periodic threads. This appeared in the Big Step rule as the hypothesis

$$vdmrt_3 = createPeriodicThreads(vdmrt_2)$$

where $vdmrt_3$ represents the state of the execution after new threads have been created.

The creation of periodic threads is given by the $createPeriodicThreads$ function in Figure 4.10. The function's post-condition applies to every object for which its $periodicCountdown$ field has reached

```
$createPeriodicThreads\colon VDMRT \to VDMRT$
$createPeriodicThreads(vdm)\,vdm' ==$
**post**
  $\forall (id_c, cpu) \in vdm.cpus \cdot$
    $\forall (id_o, obj) \in cpu.objects \cdot$
      **let** $periodic = vdm.classes(obj.class).initial$ **in**
      **let** $cpu' = vdm'.cpus(id_c)$ **in**
      **let** $obj' = cpu'.objects(id_o)$ **in**
        $(obj.periodicCountdown = 0 \;\Rightarrow$
          $obj'.periodicCountdown = precalculateNextPeriodicCountdown(periodic) \wedge$
          **let** $stm = mk\text{-}SyncCall(\textbf{nil}, (id_o, periodic.op), [\,]),$
              $body = [mk\text{-}Duration(\textsc{ExecTime}, [stm])]$ **in**
            $\exists !\; id_t \in Id_t \cdot$
              $(id_t \notin \textbf{dom}\; cpu.threads) \wedge$
              $(cpu'.threads(id_t) = mk\text{-}Thread(\textsc{Runnable}, \{\,\}, id_o, body))) \wedge$
              $(obj.periodicCountdown \neq 0 \;\Rightarrow\; obj = obj')$
```

Figure 4.10: Definition of *createPeriodicThreads*

zero; i.e. all those objects that are periodic and are ready to start a new periodic thread. Those objects will have a new thread created that is ready to invoke the operation defined in the object's class's periodic construct, and they will also have their *periodicCountdown* field recalculated for the next time their periodic thread should happen. Note that this calculation may be non-deterministic due to the fields in the *Periodic* construct.

The *Periodic* construct is defined as

$$\begin{array}{lrcl} Periodic :: & op & : & Id_{op} \\ & period & : & Expr \\ & jitter & : & Expr \\ & delay & : & Expr \\ & offset & : & Expr \end{array}$$

where the fields are defined as explained in Section 2.7.

4.6 Committing Pending Values

During an execution step, threads may change the values of instance variables in objects. However, such changes are not committed to the object state immediately: instead they are stored in the *pending* field of the *Thread* constructs, hiding the values from other threads until time has progressed sufficiently to cover the time specified by the active *PartialDuration* of the *Thread*.

The resolution of pending values and durations are handled in the Big Step rule by the

$$vdmrt_1 = commitPendingValuesAndUpdateTime(vdmrt, \tau)$$

hypothesis, which considers the prior state of the executing model and the time that the model will be updated to. The $vdmrt_1$ object represents the state of the interpreter after the time is set to τ and all pending values are committed for threads currently ready to commit, i.e. those with PENDING status and with remaining elapsed time equal to the time delta between the old and updated times.

Furthermore, all pending threads that have their values committed are returned to RUNNABLE status if they have remaining work to do, and the remaining pending threads have their *elapsed* time field

```
$commitPendingValuesAndUpdateTime\colon VDMRT \times Time \to VDMRT$
$commitPendingValuesAndUpdateTime(vdm, \tau)\,vdm' ==$
**pre** $\tau \geq vdm.\tau$
**post** $vdm'.\tau = \tau\ \wedge$
  $\forall id_c \in$ **dom** $vdm.cpus \cdot$
  $\forall id_t \in$ **dom** $vdm.cpus(id_c).threads \cdot$
    **let** $(thr, thr') = (vdm.cpus(id_c).threads(id_t), vdm'.cpus(id_c).threads(id_t))$ **in**
      $(thr.body = [\,]\ \Rightarrow\ thr'.status = \textsc{Completed}) \wedge$
      $(thr.status = \textsc{Pending}\ \Rightarrow$
        **let** $step_t = \tau - vdm.\tau$ **in**
        **let** $(d_t, d_t') = (($**hd** $thr.body).elapsed, ($**hd** $thr'.body).elapsed)$ **in**
          $(d_t = step_t\ \Rightarrow$
            $(thr'.pending = \{\,\} \wedge$
            $thr'.body =$ **tl** $thr.body \wedge$
            $(thr'.body \neq [\,]\ \Rightarrow\ thr'.status = \textsc{Runnable}) \wedge$
            $vdm'.cpus(id_c).objects = mergePending(vdm.cpus(id_c).objects, thr.pending))) \wedge$
          $(d_t \neq step_t\ \Rightarrow\ d_t' = d_t - step_t))$
```

Figure 4.11: Definition of *commitPendingValuesAndUpdateTime*

decremented by the time delta between the old and updated times. Note that those threads whose new *elapsed* time is zero are precisely those threads that have their values committed; those threads with a new, non-zero *elapsed* time must still wait before they commit their pending values. All threads with empty bodies are checked to ensure they have COMPLETED status and altered if necessary.

The behaviour of value commit and time update is contained in the semantic function shown in Figure 4.11. Note that the post-condition of the function depends on the *mergePending* function, which is a helper function that merges the pending values of a thread into the object states that those values are associated with.

4.7 Dealing with Durations and Context Switching

The execution cycle of the VDM-RT semantics is centred around *Duration* statements that are used to indicate execution times for blocks of statements. These *Duration* statements hide all changes made to the containing object's state until sufficient time has passed in the simulation for the time value of the *Duration* to reach zero, at which point the changes become visible. Also, *Duration* statements and *PartialDuration* constructs –the partially-executed form of a *Duration* statement– have the effect of blocking other threads from executing on that CPU.

It is important to note that the time value in a *Duration* statement represents information from the user about the temporal characteristics of the eventual implementation. During the execution of the body of a *Duration* statement the durations of each contained instruction are tallied and (eventually) compared to the user-specified time value. At present the semantics requires that the tally is less than the given time value, otherwise the semantics reaches a state for which there are no possible transitions. Future development of the semantics will transition to an error state in the case where the semantics presently halts.

The only exception to this behaviour is if the time value is set to the EXECTIME constant (the user would specify a 0 duration value for this), in which case there is no constraint on the execution duration. The time value of a duration can be interpreted as a strict deadline on the execution of the contained statements.

$$
\begin{array}{l}
\mathit{doContextSwitches}\colon \mathit{VDMRT} \to \mathit{VDMRT}\\
\mathit{doContextSwitches}(\mathit{vdm})\,\mathit{vdm}' ==\\
\textbf{pre}\\
\quad \exists \mathit{id}_c \in \textbf{dom}\, \mathit{vdm}.\mathit{cpus} \cdot \mathit{vdm}.\mathit{cpus}(\mathit{id}_c).\mathit{threads} \neq \{\,\}\\
\textbf{post}\\
\quad \forall \mathit{id}_c \in \textbf{dom}\, \mathit{vdm}.\mathit{cpus} \cdot\\
\quad\quad \forall \mathit{id}_t \in \textbf{dom}\, \mathit{vdm}.\mathit{cpus}(\mathit{id}_c).\mathit{threads} \cdot\\
\quad\quad\quad \textbf{let}\ (\mathit{thr}, \mathit{thr}') = (\mathit{vdm}.\mathit{cpus}(\mathit{id}_c).\mathit{threads}(\mathit{id}_t), \mathit{vdm}'.\mathit{cpus}(\mathit{id}_c).\mathit{threads}(\mathit{id}_t))\ \textbf{in}\\
\quad\quad\quad\quad \mathit{thr}.\mathit{status} = \textsc{Running} \;\Rightarrow\; \mathit{thr}'.\mathit{status} = \textsc{Running} \wedge\\
\quad\quad (\exists \mathit{thr} \in \textbf{rng}\, \mathit{vdm}.\mathit{cpus}(\mathit{id}_c).\mathit{threads} \cdot \mathit{thr}.\mathit{status} \in \{\textsc{Running}, \textsc{Runnable}\}) \;\Rightarrow\\
\quad\quad\quad (\exists! \mathit{thr}' \in \textbf{rng}\, \mathit{vdm}'.\mathit{cpus}(\mathit{id}_c).\mathit{threads} \cdot \mathit{thr}'.\mathit{status} = \textsc{Running})
\end{array}
$$

Figure 4.12: Definition of *doContextSwitches*

The checking of time values in a *Duration* is handled by the *commitPendingValuesAndUpdateTime* function, used in the Big Step. In addition to updating the current time value in the execution, the function also performs the necessary bookkeeping on the time values in (*Partial*)*Duration* constructs. First, the function will decrement all positive-valued *duration* fields in completed (*Partial*)*Duration*s by the amount of time passed since the previous step. Then, the function will commit the values held in the containing *Thread*'s *pending* field for those (*Partial*)*Duration*s that have complete and have a zero- or EXECTIME-valued *duration* field. Once the containing *Thread*'s held values have been committed, the *pending* field is cleared, the completed (*Partial*)*Duration* is removed from the *Thread*'s *body* field, and the *Thread*'s *status* field is set to RUNNABLE.

The change of thread status allows the interpreter to later switch to a different thread on that CPU; unless there is no RUNNING thread on a CPU, the *doContextSwitches* function (see Figure 4.12) will not change the state of that CPU. A context switch selects a thread on a CPU for execution from the set of threads in the RUNNABLE state. This selection is specified in a non-deterministic manner.[2]

[2] Note that the Overture tool implementation, as described in [LLB11], does a deterministic selection. This is one of the points on which the tool is a refinement of the semantics.

Chapter 5

Concluding Remarks

In this technical report we have provided a semantic model for VDM-RT using structural operational semantics. We have clarified some aspects of the VDM-RT semantic model that were not covered in sufficient detail in earlier work. The focus of the work presented here has been on the executable subset of VDM-RT, as is supported by the Overture interpreter. However, the relatively narrow nature of our focus means that there remains further work to be done for a complete independent semantics of the whole VDM-RT language.

The semantics presented here of the VDM-RT notation can also be seen in a larger context where VDM-RT is used in a collaborative simulation (co-simulation) setting. Here [CLL13] provides a semantics of a generic co-simulation framework enabling semantically well-founded co-simulation of models in different notations. In [LCL13] the VDM-RT semantics in this report is slightly adjusted to match the co-simulation framework. Note that the adjustments necessary here are very small.

Acknowledgement

The authors gratefully acknowledge funding from the European Commission through the FP7 DESTECS project (grant agreement number 248134) and the FP7 COMPASS project (grant agreement number 287829).

We thank our collaborators in the DESTECS project for feedback on elements of this work. We would also like to give special thanks to Nick Battle and Stefan Hallerstede for feedback on earlier drafts of this report.

Bibliography

[AL88] Michael Meincke Arentoft and Peter Gorm Larsen. The dynamic semantics of the bsi/vdm specification language. Master's thesis, Technical University of Denmark, DK-2800 Lyngby, Denmark, August 1988.

[AL97] Bernhard K. Aichernig and Peter Gorm Larsen. A Proof Obligation Generator for VDM-SL. In John S. Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, September 1997. ISBN 3-540-63533-5.

[BCJ84] H. Barringer, J.H. Cheng, and C.B. Jones. A Logic Covering Undefinedness in Program Proofs. *Acta Informatica*, 21:251–269, 1984.

[BFL$^+$94] 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.

[BJ78] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[CLL13] Joey W. Coleman, Kenneth Lausdahl, and Peter Gorm Larsen. Semantics for generic co-simulation of heterogenous models. Submitted for publication to the Formal Aspects of Computing journal, April 2013.

[De95] E.H. Dürr and N. Plat (editor). VDM++ Language Reference Manual. Afrodite (esprit-iii project number 6500) document, Cap Volmac, August 1995.

[DGP94] Eugène Dürr, Stephen Goldsack, and Nico Plat. Rigorous Development of Concurrent and Real-Time Object-oriented Systems, March 1994. Tutorial presented at TOOLS Europe '94, Versailles, France.

[DK92] E. Dürr and J.v. Katwijk. VDM++, A Formal Specification Language for Object Oriented Designs. In *COMP EURO 92*, pages 214–219. IEEE, May 1992.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[FL09] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[FLM$^+$05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[FLS08] 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.

[Fuc92] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.

[HJ89] I.J. Hayes and C.B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, pages 330–338, November 1989.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of teh ACM*, 12(10):576–581, October 1969.

[JM93] Cliff B. Jones and Kees Middelburg. A typed logic of partial functions reconstructed classically. Technical Report 89, Department of Philosophy, Utrecht University, April 1993.

[Jon96] Cliff Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[KM93] Stuart Kent and Richard Moore. An Axiomatic Semantics for VDM++: OO Aspects, 1993.

[Kra07] Jeff Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):37–42, 2007.

[LAMB89] Peter Gorm Larsen, Michael Meincke Arentoft, Brian Monahan, and Stephen Bear. Towards a Formal Semantics of The BSI/VDM Specification Language. In Ritter, editor, *Information Processing 89*, pages 95–100. IFIP, North-Holland, August 1989.

[Lan94] K. Lano. Expressing the Semantics of VDM++ in RTL, 1994.

[Lar94] Peter Gorm Larsen. Evaluation of Underdetermined Explicit Expressions. In M. Bertran M. Naftalin, T. Denvir, editor, *FME'94: Industrial Benefit of Formal Methods*, pages 233–250. Springer-Verlag, October 1994.

[LBF$^+$10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[LCL13] Kenneth Lausdahl, Joey W. Coleman, and Peter Gorm Larsen. The Execution Semantics of VDM Real-Time in a Co-Simulation Environment. Submitted for publication to the Science of Computer Programming journal, June 2013.

[Lei69] A.C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach Science Publishers, New York, 1969.

[LHB$^+$96] P. G. Larsen, B. S. Hansen, H. Brunn, N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.

[LL91] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods*. VDM Europe, Springer-Verlag, March 1991.

[LLB11] Kenneth Lausdahl, Peter Gorm Larsen, and Nick Battle. A Deterministic Interpreter Simulating A Distributed real time system using VDM. In *Proceedings of the 13th international conference on Formal methods and software engineering*, ICFEM'11, pages 179–194, Berlin, Heidelberg, October 2011. Springer-Verlag. ISBN 978-3-642-24558-9.

[LP95] Peter Gorm Larsen and Wiesław Pawłowski. The Formal Semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5–6):585–602, September 1995.

[LVLW10] Kenneth Lausdahl, Marcel Verhoef, Peter Gorm Larsen, and Sune Wolff. Overview of VDM-RT Constructs and Semantic Issues. In Ken Pierce, Nico Plat, and Sune Wolf, editors, *Proceedings of the 8th Overture Workshop*, number CS-TR-1224 in Technical Report Series, pages 57–67, September 2010.

[MBD$^+$00] 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.

[Mok91] A.K. Mok. Towards mechanization of real-time system design. In A.M. van Tilborg and G.M. Koob, editors, *Foundations of Real-Time Computing. Formal Specifications and Methods*. Kluwer Academic Publishers, 1991.

[Mon85] Brian Q. Monahan. A Semantic Definition of the STC VDM Reference Language. Doc. no. 9, November 1985.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, July–December 2004.

[RL10] Augusto Ribeiro and Peter Gorm Larsen. Proof Obligation Generation and Discharging for Recursive Definitions in VDM. In Jin Song and Huibiao, editors, *The 12th International Conference on Formal Engineering Methods (ICFEM 2010)*. Springer-Verlag, November 2010.

[Sch86] D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn & Bacon, 1986.

[Sco82] Dana S. Scott. Domains for Denotational Semantics. *ICALP '82*, July 1982.

[SS87] Harald Søndergaard and Peter Sestoft. Non-Determinacy and Its Semantics. Technical Report 86/12, DIKU, Datalogisk Institut, Københavns Universitet, Sigurdsgade 41, DK-2200 København N, 1987.

[SS88] Harald Søndergaard and Peter Setoft. Referential Transparancy and Allied Notions. Technical Report 88/7, DIKU, Datalogisk Institut, Københavns Universitet, Sigurdsgade 41, DK-2200 København N, 1988.

[SS90] Harald Søndergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990.

[SS92] Harald Søndergaard and Peter Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, October 1992.

[Sto77] Joseph E. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[Str67] Christopher Strachey. Fundamental concepts in programming languages. In *Lecture Notes for the International School in Computer Programming*, 1967.

[TW90] Andrzej Tarlecki and Morten Wieth. A Naive Domain Universe for VDM. In Dines Bjørner, C.A.R. Hoare, and Hans Langmaack, editors, *VDM '90 VDM and Z – Formal Methods in Software Development*, pages 552–579. VDM Europe, Springer-Verlag, April 1990.

[Ver09] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2009.

[VLH06] 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*, Lecture Notes in Computer Science 4085, pages 147–162. Springer-Verlag, 2006.

[Wie89] Morten Wieth. Loose Specification and its Semantics. In G.X. Ritter, editor, *Information Processing 89*, pages 1115–1120. IFIP, North-Holland, August 1989.

Appendix A

Complete VDM-RT Semantics

A.1 VDM-RT Abstract Syntax

The VDM-RT abstract syntax only considers a subset of the full VDM++ language dialect. It includes classes and concurrency but excludes inheritance and static access.

A.1.1 Structure

The VDM-RT expressions ($Expr$) and types ($Type$) are imported from the VDM-SL denotational semantics [LHB⁺96].

$Type = \ldots$
$Expr = \ldots$

and where the type $VDMRTModel$ represents a textual representation of a specification:

$VDMRTModel = \cdots$

Toplevel

$$\Sigma = Id_v \xrightarrow{m} VDMValue$$

$$Classes = Id_{cl} \xrightarrow{m} Class$$

$$Busses = Id_b \xrightarrow{m} Bus$$

$$CPUs = Id_c \xrightarrow{m} CPU$$

$$Pending = Id_o \xrightarrow{m} \Sigma$$

$$\begin{array}{lrcl} VDMRT :: & cpus & : & CPUs \\ & busses & : & Busses \\ & time & : & Time \\ & classes & : & Classes \end{array}$$

where

$$\begin{array}{l}
inv\text{-}VDMRT(mk\text{-}VDMRT(cpus, busses, time, classes)) \quad \triangleq \\
\qquad \forall cpu \in \mathbf{rng}\, cpus \cdot \\
\qquad\quad \forall obj \in \mathbf{rng}\, cpu.objects \cdot \\
\qquad\qquad (classes(obj.class).initial \in Period \;\Rightarrow\; obj.periodicCountdown \neq \mathbf{nil}) \\
\qquad\qquad \wedge\; obj.class \in \mathbf{dom}\, classes
\end{array}$$

Definitions

$$\begin{array}{lrcl}
Bus \;:: & cpus & : & Id_c\text{-}\mathbf{set} \\
& speed & : & \mathbb{N}_1 \\
& queue & : & (Id_c \times (CMessage \mid RMessage))^*
\end{array}$$

$$\begin{array}{lrcl}
CMessage \;:: & obj & : & Id_o \\
& op & : & Id_{op} \\
& args & : & VDMValue^* \\
& replyto & : & \left[Id_c \times Id_t\right] \\
& sendTime & : & Time
\end{array}$$

$$\begin{array}{lrcl}
RMessage \;:: & value & : & VDMValue^* \\
& replyto & : & Id_c \times Id_t \\
& sendTime & : & Time
\end{array}$$

$$\begin{array}{lrcl}
CPU \;:: & objects & : & Id_o \xrightarrow{m} Object \\
& threads & : & Id_t \xrightarrow{m} Thread \\
& speed & : & \mathbb{N}_1
\end{array}$$

where

$$\begin{array}{l}
inv\text{-}CPU(mk\text{-}CPU(objects, threads, speed)) \quad \triangleq \\
\qquad \forall t \in \mathbf{dom}\, threads \cdot \mathbf{let}\; pending = threads(t).pending \;\; \mathbf{in} \\
\quad \mathbf{dom}\, pending \subseteq \mathbf{dom}\, objects \wedge \\
\qquad\quad \forall obj \in \mathbf{dom}\, pending \cdot \\
\qquad\qquad pending(obj) \subseteq \mathbf{dom}\, objects(obj).\sigma
\end{array}$$

$$\begin{array}{lrcl}
Thread \;:: & status & : & \textsc{Running} \mid \textsc{Runnable} \mid \textsc{Waiting} \mid \textsc{Pending} \mid \textsc{Completed} \\
& pending & : & Pending \\
& context & : & Id_o \\
& body & : & (Duration \mid PartialDuration)^*
\end{array}$$

$$\begin{array}{lrcl}
Class \;:: & parents & : & Id_c^* \\
& values & : & \Sigma \\
& vars & : & Id_v \xrightarrow{m} Type \times Expr \\
& ops & : & Id_{op} \xrightarrow{m} Op \\
& funs & : & Id_f \xrightarrow{m} Fun \\
& invs & : & Fun\text{-}\mathbf{set} \\
& initial & : & Duration^* \mid Periodic
\end{array}$$

$$\begin{array}{lrcl}
Op \;:: & async & : & \mathbb{B} \\
& args & : & (Id_v \times Type)^* \\
& ret & : & Type^* \\
& body & : & Duration^* \\
& pre & : & \left[Expr\right] \\
& post & : & \left[Expr\right]
\end{array}$$

where

$$\begin{array}{l}
inv\text{-}Op(mk\text{-}Op(async, args, ret, body, pre, post)) \quad \triangleq \\
\qquad (async \;\Rightarrow\; ret = [\,]) \wedge \\
\qquad \mathbf{let}\ arguments = \mathbf{elems}\,[i \mid (i, t) \in args]\ \ \mathbf{in} \\
\qquad\quad \mathbf{len}\ args = \mathbf{card}\ arguments \wedge \\
\qquad\quad FV(pre) \subseteq arguments \wedge \\
\qquad\quad collapseOld(FV(post)) \subseteq argumentssure.args = args \wedge measure.ret = \mathbb{N}
\end{array}$$

$$\begin{array}{lrcl}
Fun \;::\; & args & : & (Id_v \times Type)^* \\
& ret & : & Type^* \\
& body & : & Expr \\
& pre & : & Expr \\
& post & : & Expr
\end{array}$$

where

$$\begin{array}{l}
inv\text{-}Fun(mk\text{-}Fun(args, ret, body, pre, post)) \quad \triangleq \\
\qquad \mathbf{let}\ arguments = \mathbf{elems}\,[i \mid (i, t) \in args]\ \ \mathbf{in} \\
\\
\qquad\quad \wedge\, FV(body) \subseteq arguments \\
\qquad\quad \wedge\, FV(pre) \subseteq arguments \\
\qquad\quad \wedge\, FV(post) \subseteq arguments
\end{array}$$

$$\begin{array}{lrcl}
Periodic \;::\; & op & : & Id_{op} \\
& period & : & Expr \\
& jitter & : & Expr \\
& delay & : & Expr \\
& offset & : & Expr
\end{array}$$

where

$$\begin{array}{l}
inv\text{-}Periodic(mk\text{-}Periodic(op, period, jitter, delay, offset)) \quad \triangleq \\
\qquad \{period, jitter, delay, offset\} \text{ must all evaluate to } Time
\end{array}$$

$$\begin{array}{lrcl}
Object \;::\; & class & : & Id_{cl} \\
& state & : & \Sigma \\
& periodicCountdown & : & [Time]
\end{array}$$

Pattern and Bind

$$\begin{array}{lrcl}
Pattern \;::\; & type & : & Type \\
& names & : & [Id^*] \\
& cTypes & : & (Type \mid Pattern)^*
\end{array}$$

$$PatternBind = Pattern \mid Bind$$

$$Bind = SetBind \mid TypeBind$$

$$\begin{array}{lrcl}
SetBind \;::\; & pattern & : & Pattern^+ \\
& exp & : & Exp
\end{array}$$

$$\begin{array}{lrcl}
TypeBind \;::\; & pattern & : & Pattern^+ \\
& type & : & Type
\end{array}$$

Statements

$$\begin{aligned}
&\mathit{Stm} = \textsc{skip}\\
&\quad \mid \mathit{Assignment} \mid \mathit{Atomic} \mid \mathit{Call} \mid \mathit{Cases} \mid \mathit{Cycles} \mid \mathit{Duration}\\
&\quad \mid \mathit{ForSet} \mid \mathit{ForSeq} \mid \mathit{ForIndex} \mid \mathit{If} \mid \mathit{LetBeStm} \mid \mathit{LetDef} \mid \mathit{New} \mid \mathit{Return}\\
&\quad \mid \mathit{SemanticStm} \mid \mathit{SimpleBlock} \mid \mathit{Start} \mid \mathit{While}
\end{aligned}$$

Note that $\mathit{SemanticStm}$ does not have syntax and is only included in the semantics to handle return of a call and durations.

$$\mathit{SemanticStm} = \mathit{PartialLetDef} \mid \mathit{ObjectContext} \mid \mathit{Wait} \mid \mathit{PartialDuration} \mid \mathit{PartialAtomic}$$

$$\begin{aligned}
\mathit{Assignment} \;::\; \mathit{target} &: \mathit{Id}_v \mid (\mathit{Id}_o \times \mathit{Id}_v)\\
\mathit{exp} &: \mathit{Exp}
\end{aligned}$$

$$\mathit{Atomic} \;::\; \mathit{assignments} : \mathit{Assignment}^*$$

$$\begin{aligned}
\mathit{PartialAtomic} \;::\; \mathit{assignments} &: \mathit{Assignment}^*\\
\mathit{oids} &: \mathit{Id}_o\text{-}\mathbf{set}
\end{aligned}$$

$$\mathit{Call} = \mathit{SyncCall} \mid \mathit{AsyncCall}$$

$$\begin{aligned}
\mathit{SyncCall} \;::\; \mathit{target} &: \big[\mathit{Id}_v \mid (\mathit{Id}_c \times \mathit{Id}_t)\big]\\
\mathit{name} &: \mathit{Id}_o \times \mathit{Id}_{op} \mid \mathit{Id}_c \times \mathit{Id}_o \times \mathit{Id}_{op}\\
\mathit{args} &: \mathit{Expr}^*
\end{aligned}$$

$$\begin{aligned}
\mathit{AsyncCall} \;::\; \mathit{name} &: \mathit{Id}_o \times \mathit{Id}_{op} \mid \mathit{Id}_c \times \mathit{Id}_o \times \mathit{Id}_{op}\\
\mathit{args} &: \mathit{Expr}^*
\end{aligned}$$

$$\mathit{Wait} \;::\; \mathit{target} : \mathit{Id}_v \mid (\mathit{Id}_c \times \mathit{Id}_t)$$

$$\mathit{Return} \;::\; \mathit{exp} : \big[\mathit{Exp}\big]$$

$$\begin{aligned}
\mathit{ObjectContext} \;::\; \mathit{object} &: \mathit{Id}_o\\
\mathit{body} &: \mathit{Stm}\\
\mathit{callCtx} &: \mathit{CallContext}
\end{aligned}$$

$$\begin{aligned}
\mathit{CallContext} \;::\; \mathit{pending} &: \mathit{Pending}\\
\mathit{state} &: \Sigma\\
\mathit{post} &: \big[\mathit{Expr}\big]
\end{aligned}$$

$$\begin{aligned}
\mathit{Cases} \;::\; \mathit{exp} &: \mathit{Exp}\\
\mathit{cases} &: (\mathit{Pattern} \times \mathit{Stm})^*\\
\mathit{others} &: \big[\mathit{Stm}\big]
\end{aligned}$$

$$\begin{aligned}
\mathit{Cycles} \;::\; \mathit{cycles} &: \mathit{Exp}\\
\mathit{body} &: \mathit{Stm}
\end{aligned}$$

$$\begin{aligned}
\mathit{Duration} \;::\; \mathit{duration} &: \textsc{ExecTime} \mid \mathit{Exp}\\
\mathit{body} &: \mathit{Stm}
\end{aligned}$$

$$\begin{aligned}
\mathit{PartialDuration} \;::\; \mathit{duration} &: \textsc{ExecTime} \mid \mathit{Time}\\
\mathit{elapsed} &: \mathit{Time}\\
\mathit{body} &: \mathit{Stm}
\end{aligned}$$

$$\begin{array}{lrcl}
\mathit{ForIndex} :: & \mathit{var} & : & \mathit{Id}_v \\
 & \mathit{from} & : & \mathit{Exp} \\
 & \mathit{to} & : & \mathit{Exp} \\
 & \mathit{by} & : & \mathit{Exp} \\
 & \mathit{body} & : & \mathit{Stm}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{ForSet} :: & \mathit{pattern} & : & \mathit{Pattern} \\
 & \mathit{setExp} & : & \mathit{Exp} \\
 & \mathit{body} & : & \mathit{Stm}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{ForSeq} :: & \mathit{pattern} & : & \mathit{Pattern} \\
 & \mathit{seqExp} & : & \mathit{Exp} \\
 & \mathit{body} & : & \mathit{Stm}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{If} :: & \mathit{exp} & : & \mathit{Exp} \\
 & \mathit{then} & : & \mathit{Stm} \\
 & \mathit{else} & : & \mathit{Stm}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{LetBe} :: & \mathit{bind} & : & \mathit{MultipleBind} \\
 & \mathit{suchThat} & : & \mathit{Exp} \\
 & \mathit{body} & : & \mathit{Stm}
\end{array}$$

$$\mathit{Definition} = \mathit{Id}_v \xrightarrow{m} \mathit{Expr}$$

$$\begin{array}{lrcl}
\mathit{LetDef} :: & \mathit{localDefs} & : & \mathit{Definition}^{+} \\
 & \mathit{body} & : & \mathit{Stm}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{PartialLetDef} :: & \mathit{context} & : & \Sigma \\
 & \mathit{localDefs} & : & \mathit{Definition}^{*} \\
 & \mathit{body} & : & \mathit{Stm}
\end{array}$$

$$\mathit{SimpleBlock} :: \mathit{body} : \mathit{Stm}^{*}$$

$$\mathit{Start} :: \mathit{obj} : \mathit{Id}_o$$

$$\begin{array}{lrcl}
\mathit{While} :: & \mathit{exp} & : & \mathit{Exp} \\
 & \mathit{body} & : & \mathit{Stm}
\end{array}$$

$$\begin{array}{lrcl}
\mathit{New} :: & \mathit{class} & : & \mathit{Id}_c \\
 & \mathit{target} & : & \mathit{Id}_v
\end{array}$$

A.2 Context Conditions/Typechecking

This section lists the top level context conditions for a subset of the types specified in Section A.1.1.

$$\mathit{TypeMap} = (\mathit{Id}_{cl} \times \mathit{Id}_v) \xrightarrow{m} \mathit{Type}$$

$$\begin{array}{l}
\mathit{wf\text{-}VDMRT}\colon \mathit{VDMRT} \times \mathit{TypeMap} \to \mathbb{B} \\
\mathit{wf\text{-}VDMRT}(\mathit{mk\text{-}VDMRT}(\mathit{cpus}, \mathit{busses}, 0, \mathit{classes}), \mathit{types}) == \\
\qquad \forall \mathit{bid} \in \mathbf{dom}\, \mathit{busses} \cdot \mathit{wf\text{-}Bus}(\mathit{busses}(\mathit{bid}), \mathit{classes}, \mathit{cpus}, \mathit{types}) \\
\quad\ \wedge \forall \mathit{cid} \in \mathbf{dom}\, \mathit{cpus} \cdot \mathit{wf\text{-}CPU}(\mathit{cpus}(\mathit{cid}), \mathit{classes}, \mathit{cpus}, \mathit{types}) \\
\quad\ \wedge \forall \mathit{clid} \in \mathbf{dom}\, \mathit{classes} \cdot (\mathit{wf\text{-}Class}(\mathit{clid}, \mathit{classes}(\mathit{clid}), \mathit{classes}, \mathit{cpus}, \mathit{types}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \forall \mathit{clid}_p \in \mathit{classes}(\mathit{clid}).\mathit{parents} \cdot \mathit{clid}_p \in \mathbf{dom}\, \mathit{classes})
\end{array}$$

Definitions

$$\begin{array}{l}
\mathit{wf\text{-}Bus}\colon \mathit{BUS} \times \mathit{CPUs} \to \mathbb{B} \\
\mathit{wf\text{-}Bus}(\mathit{mk\text{-}BUS}(\mathit{cpus}, \mathit{speed}, \mathit{queue}), \mathit{cpus}) == \forall \mathit{id}_c \in \mathit{cpus} \cdot \mathit{id}_c \in \mathbf{dom}\, \mathit{cpus}
\end{array}$$

$$\begin{array}{l}
\mathit{wf\text{-}CPU}\colon \mathit{CPU} \times \mathit{Classes} \times \mathit{CPUs} \times \mathit{TypeMap} \to \mathbb{B} \\
\mathit{wf\text{-}CPU}(\mathit{mk\text{-}CPU}(\mathit{objects}, \mathit{threads}, \mathit{speed}), \mathit{classes}, \mathit{cpus}, \mathit{types}) == \\
\quad \mathit{objects} = \{\,\} \wedge \mathit{threads} = \{\,\}
\end{array}$$

$$\begin{array}{l}
\mathit{wf\text{-}Class}\colon \mathit{Id}_{cl} \times \mathit{Class} \times \mathit{Classes} \times \mathit{CPUs} \times \mathit{TypeMap} \to \mathbb{B} \\
\mathit{wf\text{-}Class}(\mathit{clid}, \mathit{mk\text{-}Class}(\mathit{parents}, \mathit{values}, \mathit{vars}, \mathit{ops}, \mathit{funs}, \mathit{invs}, \mathit{initial}), \mathit{classes}, \mathit{cpus}, \mathit{types}) == \\
\qquad \mathit{cid} \in \mathit{Id}_c \\
\quad \wedge\ \mathit{oid} \in \mathit{Id}_o \\
\quad \wedge\ \neg\, \exists \mathit{any} \in \mathit{Id}_v \cdot (\mathit{cid}, \mathit{oid}, \mathit{any}) \in \mathbf{dom}\, \mathit{types} \\
\quad \wedge\ \mathit{types}' = \mathit{types} \dagger \{(\mathit{cid}, \mathit{oid}, \mathit{id}_v) \mapsto t \mid \mathit{id}_v \in \mathbf{dom}\, \mathit{values} \wedge t = \mathit{typeOf}(\mathit{values}(\mathit{id}_v))\} \\
\quad \wedge\ \mathit{types}'' = \mathit{types}' \dagger \{(\mathit{cid}, \mathit{oid}, \mathit{id}_v) \mapsto t \mid \mathit{id}_v \in \mathbf{dom}\, \mathit{vars} \wedge (t, \text{-}) = \mathit{vars}(\mathit{id}_v)\} \\
\quad \wedge\ \forall \mathit{id}_v \in \mathbf{dom}\, \mathit{vars} \cdot (\mathit{type}, \mathit{exp}) = \mathit{vars}(\mathit{id}_v) \wedge \\
\qquad\qquad\qquad\qquad\qquad \mathit{contextTypeOf}(\mathit{exp}, \mathit{classes}, \mathit{types}') = \mathit{vars}(\mathit{id}_v).\#1 \\
\quad \wedge\ \forall \mathit{id}_{op} \in \mathbf{dom}\, \mathit{ops} \cdot \mathit{wf\text{-}Op}(\mathit{clid}, \mathit{id}_{op}, \mathit{cid}, \mathit{oid}, \mathit{classes}, \mathit{cpus}, \mathit{types}'') \\
\quad \wedge\ \forall \mathit{id}_f \in \mathbf{dom}\, \mathit{funs} \cdot \mathit{wf\text{-}Fun}(\mathit{clid}, \mathit{funs}(\mathit{id}_f), \mathit{cid}, \mathit{oid}, \mathit{classes}, \mathit{cpus}, \mathit{types}'') \\
\quad \wedge\ \forall \mathit{fun} \in \mathit{invs} \cdot \mathit{wf\text{-}Fun}(\mathit{clid}, \mathit{fun}, \mathit{cid}, \mathit{oid}, \mathit{classes}, \mathit{cpus}, \mathit{types}'') \wedge \mathit{fun}.\mathit{ret} = [\textsc{Bool}] \\
\quad \wedge \left(\begin{array}{l} \mathit{initial} \in \mathit{Duration}^* \;\Rightarrow\; \mathit{wf\text{-}StatementSeq}(\mathit{initial}, \mathit{cid}, \mathit{clidclasses}, \mathit{cpus}, \mathit{types}'') \\ \vee\ \mathit{initial} \in \mathit{Periodic} \;\Rightarrow\; \mathit{wf\text{-}Periodic}(\mathit{initial}, \mathit{cid}, \mathit{clid}, \mathit{classes}, \mathit{cpus}, \mathit{types}'') \end{array} \right)
\end{array}$$

$$\begin{array}{l}
\mathit{wf\text{-}Op}\colon \mathit{Id}_{cl} \times \mathit{Id}_{op} \times \mathit{Id}_c \times \mathit{Id}_o \times \mathit{Classes} \times \mathit{CPUs} \times \mathit{TypeMap} \to \mathbb{B} \\
\mathit{wf\text{-}Op}(\mathit{clid}, \mathit{id}_{op}, \mathit{cid}, \mathit{oid}, \mathit{classes}, \mathit{cpus}, \mathit{types}) == \\
\quad \mathit{mk\text{-}Op}(\mathit{async}, \mathit{args}, \mathit{ret}, \mathit{body}, \mathit{pre}, \mathit{post}) = \mathit{classes}(\mathit{clid}).\mathit{ops}(\mathit{id}_{op}) \wedge \\
\quad \mathit{types}' = \mathit{types} \dagger \{(\mathit{clid}, \mathit{idv}) \mapsto \mathit{type} \mid (\mathit{idv}, \mathit{type}) \in \mathit{args}\} \wedge \\
\quad \mathit{async} = \mathbf{true} \;\Rightarrow\; \mathbf{len}\, \mathit{ret} = 0 \wedge \\
\quad \mathit{wf\text{-}Statement}(\mathit{body}, \mathit{cid}, \mathit{clid}, \mathit{classes}, \mathit{cpus}, \mathit{types}') \wedge \\
\quad \forall \mathit{exp} \in \{\mathit{pre}, \mathit{post}\} \cdot \mathit{exp} \neq \mathbf{nil} \;\Rightarrow\; \mathit{contextTypeOf}(\mathit{exp}, \mathit{classes}, \mathit{types}') = \textsc{Bool}
\end{array}$$

$$\begin{array}{l}
\mathit{wf\text{-}Fun}\colon \mathit{Id}_{cl} \times \mathit{Fun} \times \mathit{Id}_c \times \mathit{Id}_o \times \mathit{Classes} \times \mathit{CPUs} \times \mathit{TypeMap} \to \mathbb{B} \\
\mathit{wf\text{-}Fun}(\mathit{clid}, \mathit{mk\text{-}Fun}(\mathit{args}, \mathit{ret}, \mathit{body}, \mathit{pre}, \mathit{post}), \mathit{cid}, \mathit{oid}, \mathit{classes}, \mathit{cpus}, \mathit{types}) == \\
\quad \mathit{types}' = \mathit{types} \dagger \{(\mathit{clid}, \mathit{idv}) \mapsto \mathit{type} \mid (\mathit{idv}, \mathit{type}) \in \mathit{args}\} \wedge \\
\quad \mathit{contextTypeOf}(\mathit{body}, \mathit{classes}, \mathit{types}') = \mathit{ret} \wedge \\
\quad \forall \mathit{exp} \in \{\mathit{pre}, \mathit{post}\} \cdot \mathit{exp} \neq \mathbf{nil} \;\Rightarrow\; \mathit{contextTypeOf}(\mathit{exp}, \mathit{classes}, \mathit{types}') = \textsc{Bool}
\end{array}$$

$$\begin{array}{l}
\mathit{wf\text{-}Periodic}\colon \mathit{Periodic} \times \mathit{Id}_c \times \mathit{Id}_{cl} \times \mathit{Classes} \times \mathit{CPUs} \times \mathit{TypeMap} \to \mathbb{B} \\
\mathit{wf\text{-}Periodic}(\mathit{mk\text{-}Periodic}(\mathit{op}, \mathit{period}, \mathit{jitter}, \mathit{delay}, \mathit{offset}), \mathit{cid}, \mathit{clid}, \mathit{classes}, \mathit{cpus}, \mathit{types}) == \\
\quad \mathit{op} \in \mathbf{rng}\, \mathit{classes}(\mathit{clid}).\mathit{ops} \wedge \\
\quad (\forall \mathit{exp} \in \{\mathit{period}, \mathit{jitter}, \mathit{delay}.\mathit{offset}\} \cdot \mathit{contextTypeOf}(\mathit{exp}, \mathit{classes}, \mathit{types}) = \textsc{Time} \wedge \mathit{exp} \geq 0) \wedge \\
\quad \mathit{period} > 0 \wedge \mathit{delay} < \mathit{period}
\end{array}$$

Statements

$$\begin{array}{l}
\textit{wf-StatementSeq}\colon \textit{Stm}^* \times \textit{Id}_c \times \textit{Id}_{cl} \times \textit{Classes} \times \textit{CPUs} \times \textit{TypeMap} \to \mathbb{B} \\
\textit{wf-StatementSeq}(\textit{stms}, \textit{cid}, \textit{clid}, \textit{classes}, \textit{cpus}, \textit{types}) == \\
\quad \textbf{if } \textit{stms} = [\,] \\
\quad \textbf{then true} \\
\quad \textbf{else } \; \textit{wf-Statement}(\textbf{hd } \textit{stms}, \textit{cid}, \textit{clid}, \textit{classes}, \textit{cpus}, \textit{types}) \wedge \\
\qquad\qquad \textit{wf-StatementSeq}(\textbf{tl } \textit{stms}, \textit{cid}, \textit{clid}, \textit{classes}, \textit{cpus}, \textit{types})
\end{array}$$

$$\textit{wf-Statement}\colon \textit{Stm} \times \textit{Id}_c \times \textit{Id}_{cl} \times \textit{Classes} \times \textit{CPUs} \times \textit{TypeMap} \to \mathbb{B}$$

The context conditions for statements have been omitted, but follow the general pattern of ensuring that all variable references are valid, all operation calls are to objects of the correct type, and so on.

A.3 Rules

This section describes all inference rules used in this work including the inference rule signatures.

$$\begin{array}{l}
\{\textit{SkipTime}, \textit{IfTime}, \textit{WhileTime}, \textit{CasesTime}, \textit{NewTime}, \textit{ForIndexTime}, \\
\textit{ForSeqTime}, \textit{ForSetTime}, \textit{LetDefTime}, \textit{LetBeTime}, \textit{LocalAssignmentTime}, \\
\textit{RemoteAssignmentTime}, \textit{AtomicTime}, \textit{StartTime}, \textit{LocalSyncCallTime}, \\
\textit{LocalAsyncCallTime}, \textit{RemoteSyncCallTime}, \textit{RemoteAsyncCallTime}, \\
\textit{ReturnTime}\} \subseteq \textit{Time}
\end{array}$$

A.3.1 Signatures

$$\xrightarrow{vdmrt}\colon (\textit{VDMRT} \times \textit{Time}) \times (\textit{VDMRT} \times \textit{Time})$$

$$\xrightarrow{init}\colon (\textit{VDMRTModel}) \times (\textit{VDMRT})$$

$$\xrightarrow{exec}\colon (\textit{VDMRT}) \times (\textit{VDMRT} \times \textit{Time})$$

$$\xrightarrow{busses}\colon (\textit{VDMRT}) \times (\textit{VDMRT})$$

$$\begin{array}{l}
\textit{Time} \times \textit{Classes} \vdash \\
\qquad \xrightarrow{cpus}\colon (\textit{Cpus} \times \textit{Busses}) \times (\textit{Cpus} \times \textit{Busses} \times \textit{Time})
\end{array}$$

$$\begin{array}{l}
\textit{Time} \times \textit{Classes} \vdash \\
\qquad \xrightarrow{bus}\colon (\textit{BUS} \times \textit{Cpus}) \times (\textit{BUS} \times \textit{Cpus})
\end{array}$$

$$\begin{array}{l}
\textit{Time} \times \textit{Classes} \times \textit{Cpus} \times \textit{Id}_c \vdash \\
\qquad \xrightarrow{cpu}\colon (\textit{CPU} \times \textit{Busses}) \times (\textit{CPU} \times \textit{Busses} \times \textit{Time})
\end{array}$$

$$\begin{array}{l}
\textit{Time} \times \textit{Classes} \times \textit{Cpus} \times \textit{Id}_c \times \textit{Id}_t \times \textit{Id}_o \vdash \\
\qquad \xrightarrow{dur}\colon (\textit{Duration}^* \times \textit{Pending} \times \textit{CPU} \times \textit{Busses}) \times \\
\qquad\qquad (\textit{Duration}^* \times \textit{Pending} \times \textit{CPU} \times \textit{Busses} \times \textit{Time})
\end{array}$$

$$\begin{array}{l}
\textit{Time} \times \textit{Classes} \times \textit{Cpus} \times \textit{Id}_c \times \textit{Id}_t \times \textit{Id}_o \vdash \\
\qquad \xrightarrow{stmt}\colon (\textit{Stm}^* \times \textit{Pending} \times \textit{CPU} \times \textit{Busses}) \times (\textit{Stm}^* \times \textit{Pending} \times \textit{CPU} \times \textit{Busses} \times \textit{Time})
\end{array}$$

$$\xrightarrow{bind} : (Bind \times Pending \times CPU) \times (Pattern \times \Sigma)$$

$$Classes, Cpus, Pending, Id_o \vdash [\![Exp]\!] \to (VDMValue \times Time)$$

A.3.2 Top level rules

The Init rule initializes the *vdmrt* record from a source *model* by creating the CPUs, busses and classes.

$\boxed{\text{Init}}$

$$\frac{\begin{array}{l} cpus = createCpus(demodel) \\ busses = createBusses(cpus, demodel) \\ classes = createClasses(demodel) \\ cpus' = createInitialInstances(cpus, classes, demodel) \\ vdmrt = mk\text{-}VDMRT(cpus', busses, 0, classes) \end{array}}{(model) \xrightarrow{init} (vdmrt)}$$

The Big Step rule is the top level rule of the semantics. It is a big step rule that first deals with the creation of a new execution context by committing ready pending variables and updating time, and then handles the activity of the busses, creation of periodic threads and context switches. The $\xrightarrow{exec}$ transition relation is used to execute the model in the updated context. Lastly, this rule calculates the minimum time until next pending commit.

$\boxed{\text{Big Step}}$

$$\frac{\begin{array}{l} vdmrt^1 = commitPendingValuesAndUpdateTime(vdmrt, \tau) \\ vdmrt^1 \xrightarrow{busses} vdmrt^2 \\ vdmrt^3 = createPeriodicThreads(vdmrt^2) \\ vdmrt^4 = doContextSwitches(vdmrt^3) \\ vdmrt^4 \xrightarrow{exec} (vdmrt^5, \tau_b) \\ \tau_b' = min(\tau_b, minPendingCommitTime(vdmrt^5)) \end{array}}{(vdmrt, \tau) \xrightarrow{vdmrt} (vdmrt^5, \tau_b')}$$

The Exec rule uses the $\xrightarrow{cpus}$ rule to execute the CPUs and calculate a new time bound.

$\boxed{\text{Exec}}$

$$\frac{\tau, classes \vdash (cpus, busses) \xrightarrow{cpus} (cpus', busses', \tau_b)}{mk\text{-}VDMRT(cpus, busses, \tau, cls) \xrightarrow{exec} (mk\text{-}VDMRT(cpus', busses', \tau, cls), \tau_b)}$$

Bus rules

The two rules, Busses and Busses Base, deal with bus activity. The Busses rule selects a bus from the set of all busses and uses the *Rdebus* transition relation to execute a single bus' activity and continues with the remaining busses using down with the *Rdebusses* transition relation recursively. The recursive base case is handled by the Busses Base rule.

$\boxed{\text{Busses}}$

$$\frac{\begin{array}{l} busses(b) = bus \\ \tau, classes \vdash (bus, cpus) \xrightarrow{bus} (bus', cpus') \\ busses' = \{bus\} \mathbin{\lhd\!\!\!\!-} busses \\ mk\text{-}VDMRT(cpus', busses', \tau, cls) \xrightarrow{busses} mk\text{-}VDMRT(cpus'', busses'', \tau, cls) \\ busses''' = busses \dagger \{b \to bus'\} \end{array}}{mk\text{-}VDMRT(cpus, busses, \tau, cls) \xrightarrow{busses} mk\text{-}VDMRT(cpus'', busses''', \tau, cls)}$$

Busses Base

$$\frac{}{\textit{mk-VDMRT}(\textit{cpus}, \{\,\}, \tau, \textit{cls}) \xrightarrow{\textit{busses}} \textit{mk-VDMRT}(\textit{cpus}, \{\,\}, \tau, \textit{cls})}$$

There are three rules that deal with individual bus activity: Bus Base, Bus Call and Bus Return. The rules assume that the queue on the bus is sorted by the time that the messages were added to the bus queue. The Bus Base rule serves as the base case when the message at the top of the bus queue has an arrival time later than the current time.

Bus Base

$$\frac{\begin{array}{l} \textit{bus.queue} = [(c, \textit{msg})] \curvearrowright \textit{queue}' \\ \tau < \textit{arrivalTime}(\textit{bus.speed}, \textit{msg}) \end{array}}{\tau, \textit{classes} \vdash (\textit{bus}, \textit{cpus}) \xrightarrow{\textit{bus}} (\textit{bus}, \textit{cpus})}$$

The Bus Call rule reduces the bus queue if the queue has a $\textit{CMessage}$ at the queue head and the arrival time of that message is earlier than or equal to the current time τ. The message is removed from the queue and a new thread is created on the receiving CPU. The body of the newly created thread is created with a $\textit{SyncCall}$ matching the requested call from the message. Finally, the rule makes a recursive call to $\xrightarrow{\textit{bus}}$ to further process the bus queue.

Bus Call

$$\frac{\begin{array}{l} \textit{bus.queue} = [(c, \textit{msg})] \curvearrowright \textit{queue}' \\ \tau \geq \textit{arrivalTime}(\textit{bus.speed}, \textit{msg}) \\ \textit{mk-CMessage}(\textit{id}_o, \textit{op}, \textit{args}, \textit{replyto}, \textit{sendTime}) = \textit{msg} \\ \textit{id}_o \in \mathbf{dom}\, \textit{cpus}(c).\textit{objects} \\ \textit{cpu}' = \textit{createThread}(\textit{cpu}, \textit{id}_o, [\textit{mk-Duration}(0, [\textit{mk-SyncCall}(\textit{replyto}, (\textit{id}_o, \textit{op}), \textit{args})]]) \\ \textit{cpus}' = \textit{cpus} \dagger \{c \mapsto \textit{cpu}'\} \\ \textit{bus}' = \textit{mk-Bus}(\textit{bus.cpus}, \textit{bus.speed}, \textit{queue}') \\ \tau, \textit{classes} \vdash (\textit{bus}', \textit{cpus}') \xrightarrow{\textit{bus}} (\textit{bus}'', \textit{cpus}'') \end{array}}{\tau, \textit{classes} \vdash (\textit{bus}, \textit{cpus}) \xrightarrow{\textit{bus}} (\textit{bus}'', \textit{cpus}'')}$$

The Bus Return rule is similar to Bus Call but handles return messages ($\textit{RMessage}$). The return message is transformed into a $\textit{Return}$ statement in the target thread which must have the status WAITING. The $\textit{Return}$ is inserted into the body of the duration at the head of the thread body. When the new return is inserted the thread status is changed into RUNNABLE allowing its execution to resume.

Bus Return

$$\frac{\begin{array}{l} \textit{bus.queue} = [(c, \textit{msg})] \curvearrowright \textit{queue}' \\ \tau \geq \textit{arrivalTime}(\textit{bus.speed}, \textit{msg}) \\ \textit{mk-RMessage}(\textit{values}, \textit{replyto}, \textit{sendTime}) = \textit{msg} \\ \textit{replyto} = (c, t) \\ \textit{cpus}(c) = \textit{mk-CPU}(\textit{objects}, \textit{threads}, \textit{speed}) \\ \textit{threads}(t) = \textit{mk-Thread}(\text{WAITING}, \textit{pending}, \textit{context}, \textit{body}) \\ \textit{body} = [\textit{mk-Duration}(\tau_d, \textit{stms})] \curvearrowright \textit{remainder} \\ \textit{smts}' = \textit{insertReturn}(\textit{stms}, \textit{mk-Return}(\textit{values})) \\ \textit{body}' = [\textit{mk-Duration}(\tau_d, \textit{stms}')] \curvearrowright \textit{remainder} \\ \textit{thread}' = \textit{mk-Thread}(\text{RUNNABLE}, \textit{pending}, \textit{context}, \textit{body}') \\ \textit{threads}' = \textit{threads} \dagger \{t \to \textit{thread}'\} \\ \textit{cpus}' = \textit{cpus} \dagger \{c \to \textit{mk-CPU}(\textit{objects}, \textit{threads}', \textit{speed})\} \\ \textit{bus}' = \textit{mk-Bus}(\textit{bus.cpus}, \textit{bus.speed}, \textit{queue}') \\ \tau, \textit{classes} \vdash (\textit{bus}', \textit{cpus}') \xrightarrow{\textit{bus}} (\textit{bus}'', \textit{cpus}'') \end{array}}{\tau, \textit{classes} \vdash (\textit{bus}, \textit{cpus}) \xrightarrow{\textit{bus}} (\textit{bus}'', \textit{cpus}'')}$$

CPU rules

Rules for all CPUs The following two inference rules describe how all CPUs are executed. The rule CPUs Step selects a single CPU and uses the *Rdecpu* transition relation to execute a single CPU before it uses the $\xrightarrow{cpus}$ transition relation to recursively execute the rest of the CPUs, where the CPUs Base rule serves as the base case.

$\boxed{\text{CPUs Base}}$

$$\frac{}{\tau, classes \vdash (\{\,\}, busses) \xrightarrow{cpus} (\{\,\}, busses, \infty)}$$

$\boxed{\text{CPUs Step}}$

$$\frac{\begin{array}{l} cpus(c) = cpu \\ \tau, classes, cpus, c \vdash (cpu, busses) \xrightarrow{cpu} (cpu', busses', elapsed) \\ \tau, classes \vdash (\{c\} \mathbin{\triangleleft\!\!\!\!-} cpus, busses') \xrightarrow{cpus} (cpus'', busses'', \tau_b) \\ cpus''' = cpus'' \dagger \{c \mapsto cpu'\} \\ \tau_b' = min(elapsed, \tau_b) \end{array}}{\tau, classes \vdash (cpus, busses) \xrightarrow{cpus} (cpus''', busses'', \tau_b')}$$

Rules for single CPUs The following two inference rules, CPU Pending and CPU Running, describe how a single CPU schedules execution. Both rules select a thread, execute its body with the $\xrightarrow{dur}$ transition relation and updates the thread *status*, *pending* and *body*. The new thread status is handled differently in the two rules. The CPU Pending rule sets the thread status to PENDING if the new thread body is empty after applying the $\xrightarrow{dur}$ transition relation to the thread body. Whereas the CPU Running sets the status to RUNNING indicating that it can continue to execute if the new thread body is not empty. Note that a thread with status PENDING is handled in the *commitPendingValuesAndUpdateTime* function of the top level rule Big Step.

$\boxed{\text{CPU Pending}}$

$$\frac{\begin{array}{l} cpu = mk\text{-}CPU(objs, thrs, spd) \\ thrs(t) = mk\text{-}Thread(\textsc{Running}, o, pending, body) \\ \tau, classes, cpus, c, t, o \vdash (body, pending, cpu, busses) \xrightarrow{dur} (body', pending', cpu', busses', \delta) \\ (\mathbf{hd}\, body').body = [\,] \\ thrs' = thrs \dagger \{t \mapsto mk\text{-}Thread(\textsc{Pending}, o, pending', body')\} \\ cpu'' = mk\text{-}CPU(objs, thrs', spd) \end{array}}{\tau, classes, cpus, c \vdash (cpu, busses) \xrightarrow{cpu} (cpu'', busses', \delta)}$$

$\boxed{\text{CPU Running}}$

$$\frac{\begin{array}{l} cpu = mk\text{-}CPU(objs, thrs, spd) \\ thrs(t) = mk\text{-}Thread(\textsc{Running}, o, pending, body) \\ \tau, classes, cpus, c, t, o \vdash (body, pending, cpu, busses) \xrightarrow{dur} (body', pending', cpu', busses', \delta) \\ (\mathbf{hd}\, body').body \neq [\,] \\ thrs' = thrs \dagger \{t \mapsto mk\text{-}Thread(\textsc{Running}, o, pending', body')\} \\ cpu'' = mk\text{-}CPU(objs, thrs', spd) \end{array}}{\tau, classes, cpus, c \vdash (cpu, busses) \xrightarrow{cpu} (cpu'', busses', \delta)}$$

Durations

The duration rules are split up into three steps starting with the rule Duration Eval which evaluates the time of the duration and replaces the expression in the duration with the calculated value. The second rule, Duration Step to PartialDuration, converts a duration into a partial duration, executing at least part of the duration's body in the process. Finally, the Duration Step PartialDuration takes a partial duration and executes it; the rest in the result of the $\xrightarrow{stmt}$ transition relation is then encapsulated into a new partial duration that replaces the original partial duration. While executing, it uses the partial duration to record the total time taken to execute and the remaining statements that still require execution. The CPU Pending rule removes those partial durations that have empty *body* fields, and it also changes the thread status to PENDING enabling the top level rule Big Step to commit the changes made in the duration.

Duration Eval

$$\dfrac{\begin{array}{l} exp \notin (\mathit{Time} \cup \{\textrm{EXECTIME}\}) \\ classes, cpus, pending, o \vdash [\![exp]\!] = (value, \text{-}) \\ body = [\mathit{mk\text{-}Duration}(exp, stmts)] \curvearrowright rest \\ body' = [\mathit{mk\text{-}Duration}(value, stmts)] \curvearrowright rest \\ \tau, classes, cpus, c, t, o \vdash (body', pending, cpu, busses) \xrightarrow{stmt} (body', pending', cpu', busses', \delta) \end{array}}{\tau, classes, cpus, c, t, o \vdash (body, pending, cpu, busses) \xrightarrow{dur} (body', pending', cpu', busses', \delta)}$$

Duration Step to PartialDuration

$$\dfrac{\begin{array}{l} n \neq \textrm{EXECTIME} \;\Rightarrow\; \delta \leq n \\ body = [\mathit{mk\text{-}Duration}(n, stmts)] \curvearrowright tail \\ \tau, classes, cpus, c, t, o \vdash (stmts, pending, cpu, busses) \xrightarrow{stmt} (rest, pending', cpu', busses', \delta) \\ body' = [\mathit{mk\text{-}PartialDuration}(n, \delta, rest)] \curvearrowright tail \end{array}}{\tau, classes, cpus, c, t, o \vdash (body, pending, cpu, busses) \xrightarrow{dur} (body', pending', cpu', busses', \delta)}$$

Duration Step PartialDuration

$$\dfrac{\begin{array}{l} n \neq \textrm{EXECTIME} \;\Rightarrow\; \delta \leq (n - \delta_{elapsed}) \\ body = [\mathit{mk\text{-}PartialDuration}(n, \delta_{elapsed}, stmts)] \curvearrowright tail \\ \tau, classes, cpus, c, t, o \vdash (stmts, pending, cpu, busses) \xrightarrow{stmt} (rest, pending', cpu', busses', \delta) \\ body' = [\mathit{mk\text{-}PartialDuration}(n, \delta_{elapsed} + \delta, rest)] \curvearrowright tail \end{array}}{\tau, classes, cpus, c, t, o \vdash (body, pending, cpu, busses) \xrightarrow{dur} (body', pending', cpu', busses', \delta)}$$

General Statements

The inference rules for statements use a small step semantics to step statements through recursive application of the $\xrightarrow{stmt}$ rule until the time bound is reached. When the time bound is reached the rules return as a big step. The Stmt Base is the base case of the recursive application and stops the statement execution and returns with an empty set of statements. The $\xrightarrow{stmt}$ rule strips a statement from the sequence of statements which is either evaluated and removed or rewritten due to a partial evaluation. The rule always returns the combined time it took to execute the statements and inner expressions.

Stmt Base

$$\dfrac{}{\tau, classes, cpus, c, t, o \vdash ([\,], pending, cpu, busses) \xrightarrow{stmt} ([\,], pending, cpu, busses, 0)}$$

The Stmt Skip rule removes the skip statement from the head of a sequence of statements and increments time accordingly.

Stmt Skip

$$\dfrac{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash (rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending, cpu, busses, \delta) \\ \delta' = \mathit{SkipTime} + \delta \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([\textrm{SKIP}] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending, cpu, busses, \delta') \end{array}}$$

The Stmt SimpleBlock executes the statements of the block and removes it from the sequence of statements.

Stmt SimpleBlock

$$\frac{\begin{array}{l}\tau, classes, cpus, c, t, o \vdash \\ (stms \curvearrowright rest, pending', cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta)\end{array}}{\begin{array}{l}\tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}SimpleBlock(stms)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta)\end{array}}$$

The Stmt If True and Stmt If False both removes the *if-statement* from the sequence of statements and replaces it with either with the $then$ or $else$ depending on the evaluated value of the test expression in the *if-statement*.

Stmt If True

$$\frac{\begin{array}{l}classes, cpus, pending, o \vdash [\![e]\!] = (\mathbf{true}, \delta_e) \\ \tau, classes, cpus, c, t, o \vdash ([th] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \delta' = \delta_e + \delta + IfTime\end{array}}{\begin{array}{l}\tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}If(e, th, el)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta')\end{array}}$$

Stmt If False

$$\frac{\begin{array}{l}classes, cpus, pending, o \vdash [\![e]\!] = (\mathbf{false}, \delta_e) \\ \tau, classes, cpus, c, t, o \vdash ([el] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \delta' = \delta_e + \delta + IfTime\end{array}}{\begin{array}{l}\tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}If(e, th, el)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta')\end{array}}$$

The Stmt While True rule prefixes the sequence of statements with the *while statement*'s $body$ if the test expression is true, whereas the Stmt While False simply removes the *while statement* when the test expression is false.

Stmt While True

$$\frac{\begin{array}{l}classes, cpus, pending, o \vdash [\![e]\!] = (\mathbf{true}, \delta_e) \\ stms = [body, mk\text{-}While(e, body)] \curvearrowright rest \\ \tau, classes, cpus, c, t, o \vdash (stms, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \delta' = \delta_e + \delta + WhileTime\end{array}}{\begin{array}{l}\tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}While(e, body)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta')\end{array}}$$

Stmt While False

$$\frac{\begin{array}{l}classes, cpus, pending, o \vdash [\![e]\!] = (\mathbf{false}, \delta_e) \\ \tau, classes, cpus, c, t, o \vdash (rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \delta' = \delta_e + \delta + WhileTime\end{array}}{\begin{array}{l}\tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}While(e, body)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses')\end{array}}$$

The Stmt Cases rule starts by evaluating the expression of the *cases statement*, and then it constructs a list ($alts$) of cases where the pattern matched combined with the $others$ statement if it is given and a skip statement in case no cases matched or no $others$ statement were given. Then a $PartialLetDef$ is created with the state and statement from the head of the list *alts* which then is appended to the rest and executed.

Stmt Cases

$$\begin{array}{l}
classes, cpus, pending, o \vdash [\![e]\!] = (value, \delta_e) \\
alts = [(\sigma, stm) \mid i \in \textbf{inds}\, cases \bullet (p, stm) = cases(i) \wedge \sigma = match(p, value) \wedge \sigma \neq \{\,\}] \\
\qquad\quad \curvearrowright [(\{\,\}, others) \mid others \neq \textbf{nil}] \curvearrowright [(\{\,\}, \textsc{skip})] \\
(\sigma, stm) = \textbf{hd}\, alts \\
let = mk\text{-}PartialLetDef(\sigma, [\,], stm) \\
\tau, classes, cpus, c, t, o \vdash ([let] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\
\delta' = \delta_e + \delta + CasesTime \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}Cases(e, cases, others)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta')
\end{array}$$

The Stmt New rule creates a new object of the *cl* class and both adds the object to the CPU but also updates pending with *target* pointing to the new object. The class instantiated must not be an active thread, therefore, the initial field must not be periodic or contain a set of durations. When the new object is created a fresh *oid* is selected and all initial values and variables are evaluated to the new object state σ used in the new object. The object is then added to the CPU and the *target* is pointed at the new object in the pending map using in further evaluation. It is important to note that this rule is not type correct since the *pending* map does not allow for Id_o but only Id_v, further work is needed to update the pending lookup in the cases where an object id is encountered since this requires the state of the object to be made accessible.

Stmt New

$$\begin{array}{l}
classes(cl).initial \notin Periodic \cup \{\textbf{nil}\} \\
oid \in Id_o \\
oid \notin \bigcup\{\textbf{dom}\, acpu.objects \mid acpu \in (\textbf{rng}\, cpus \cup \{cpu\})\} \\
initVals = classes(cl).values \\
initVars = classes(cl).vars \\
inits = \{id \mapsto (v, \delta_e) \mid id \in \textbf{dom}\, initVars \wedge classes, cpus, initVals \vdash [\![e]\!] = (v, \delta_e)\} \\
\sigma = initVals \dagger \{id \mapsto v \mid id \in \textbf{dom}\, inits \wedge (v, \text{-}) = inits(id)\} \\
\delta_e = sumMapRange(id \mapsto \delta \mid id \in \textbf{dom}\, inits \wedge inits(id) = (\text{-}, \delta)) \\
obj = mk\text{-}Object(cl, \sigma, \textbf{nil}) \\
cpu' = mk\text{-}CPU(cpu.objects \dagger \{oid \mapsto obj\}, cpu.threads, cpu.speed) \\
pending' = pending \dagger \{o \mapsto (pending(o) \dagger \{target \mapsto oid\})\} \\
\tau, classes, cpus, c, t, o \vdash (rest, pending', cpu', busses) \xrightarrow{stmt} (rest', pending'', cpu'', busses', \delta) \\
\delta' = \delta + \delta_e + NewTime \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}New(cl, target)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu'', busses', \delta')
\end{array}$$

Duration

The duration statements always start with the Stmt Duration Eval which evaluates the time expression and replaces it with the actual value. Then either the duration is executed by Stmt Duration Complete, and fully completes with an empty rest or a return statement fully evaluated which results in the duration being removed or the if a rest exists then the duration is replaced with a partial duration and execution is contained.

Stmt Duration Eval

$$\frac{\begin{array}{l} exp \notin Time \cup \{\textsc{ExecTime}\} \\ classes, cpus, pending, o \vdash [\![exp]\!] = (value, \text{-}) \\ rest' = [mk\text{-}Duration(value, stm)] \curvearrowright rest \\ \tau, classes, cpus, c, t, o \vdash (rest', pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta') \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}Duration(exp, stm)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest'', pending', cpu', busses', \delta') \end{array}}$$

Stmt Duration Complete

$$\frac{\begin{array}{l} dur = mk\text{-}Duration(value, stm) \\ value \in Time \cup \{\textsc{ExecTime}\} \\ \tau, classes, cpus, c, t, o \vdash ([stm], pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ rest' = [\,] \vee (rest' = [mk\text{-}Return(v)] \wedge v \in VDMValue) \\ rest'' = rest' \curvearrowright rest \\ value \neq \textsc{ExecTime} \;\Rightarrow\; \delta \leq value \\ \tau, classes, cpus, c, t, o \vdash (rest'', pending', cpu', busses') \xrightarrow{stmt} (rest''', pending'', cpu'', busses'', \delta') \\ value \neq \textsc{ExecTime} \;\Rightarrow\; \delta'' = value + \delta' \\ value = \textsc{ExecTime} \;\Rightarrow\; \delta'' = \delta + \delta' \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([dur] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest''', pending'', cpu'', busses'', \delta'') \end{array}}$$

The three rules Stmt Duration to PartialDuration wrap the rest from a duration step that did not complete, and the Stmt Duration Step PartialDuration executes the statements of the partial duration in the case where the head of the statements is different from a return statement.

Stmt Duration to PartialDuration

$$\frac{\begin{array}{l} value \in Time \cup \{\textsc{ExecTime}\} \\ \tau, classes, cpus, c, t, o \vdash ([stm], pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \mathbf{hd}\; rest \notin Return \\ value \neq \textsc{ExecTime} \;\Rightarrow\; \delta \leq value \\ rest'' = [mk\text{-}PartialDuration(value, \delta, mk\text{-}SimpleBlock(rest')] \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}Duration(value, stm)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest'', pending', cpu', busses', \delta) \end{array}}$$

Stmt Duration Step PartialDuration

$$\frac{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash ([stm], pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \mathbf{hd}\; rest \notin Return \\ value \neq \textsc{ExecTime} \;\Rightarrow\; \delta \leq (value - \delta_{elapsed}) \\ rest'' = [mk\text{-}PartialDuration(value, \delta_{elapsed} + \delta, mk\text{-}SimpleBlock(rest')] \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}PartialDuration(value, \delta_{elapsed}, stm)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} \\ \quad (rest'', pending', cpu', busses', \delta) \end{array}}$$

Finally, the Stmt Duration Complete PartialDuration combined the rest of the evaluation of statements from the partial duration with the rest of the current thread body and continues execution. The two rests can be combined because either the rest is an empty sequence or it contains a fully evaluated return statement.

$\boxed{\text{Stmt Duration Complete PartialDuration}}$

$$\begin{array}{l}
partialduration = mk\text{-}PartialDuration(value, \delta_{elapsed}, stm) \\
\tau, classes, cpus, c, t, o \vdash ([stm], pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\
rest' = [\,] \vee (rest' = [mk\text{-}Return(v)] \wedge v \in VDMValue) \\
rest'' = rest' \curvearrowright rest \\
value \neq \textsc{ExecTime} \;\Rightarrow\; \delta \leq (value - \delta_{elapsed}) \\
\tau, classes, cpus, c, t, o \vdash (rest'', pending', cpu', busses') \xrightarrow{stmt} (rest''', pending'', cpu'', busses'', \delta') \\
value \neq \textsc{ExecTime} \;\Rightarrow\; \delta'' = value + \delta' \\
value = \textsc{ExecTime} \;\Rightarrow\; \delta'' = \delta + \delta' \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([partialduration] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} \\
\quad (rest''', pending'', cpu'', busses'', \delta'')
\end{array}$$

For

The for statements are divided into three groups: Stmt ForIndex, Stmt ForSeq and Stmt ForSet. They are all unfolded to a sequence of partial let defs representing the loops of the for statement. The Stmt ForIndex starts out by evaluating the $from$, to and by, and then calculates the number of partial let defs needed to unfold the loop. The Stmt ForSeq starts by evaluating the $seqExp$ to a set value, and then unfolding the loop where the state in each partial let def has the pattern p bound to an element of the sequence represented by $seqExp$. The Stmt ForSet converts the set represented by $setExp$ to a sequence, and then follows the same procedure as Stmt ForSeq.

$\boxed{\text{Stmt ForIndex}}$

$$\begin{array}{l}
forindex = mk\text{-}ForIndex(id_v, e_{from}, e_{to}, e_{by}, stm) \\
classes, cpus, pending, o \vdash [\![e_{from}]\!] = (v_{from}, \delta_{from}) \\
classes, cpus, pending, o \vdash [\![e_{to}]\!] = (v_{to}, \delta_{to}) \\
classes, cpus, pending, o \vdash [\![e_{by}]\!] = (v_{by}, \delta_{by}) \\
stms = [mk\text{-}PartialLetDef(\{id_v \mapsto v\}, [\,], stm) \\
\qquad\qquad \mid n \in \mathbb{N} \wedge v = v_{from} + n \cdot v_{by} \wedge ((v_{from} < v < v_{to}) \vee (v_{to} < v < v_{from}))] \\
\tau, classes, cpus, c, t, o \vdash (stms \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\
\delta' = \delta_{from} + \delta_{to} + \delta_{by} + \delta + ForIndexTime \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([forindex] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta')
\end{array}$$

$\boxed{\text{Stmt ForSeq}}$

$$\begin{array}{l}
classes, cpus, pending, o \vdash [\![seqExp]\!] = (seq, \delta_e) \\
stms = [mk\text{-}PartialLetDef(\sigma, [\,], stm) \mid i \in \textbf{inds}\, seq \wedge \sigma = match(p, seq(i))] \\
\tau, classes, cpus, c, t, o \vdash (stms \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\
\delta' = \delta_e + \delta + ForSeqTime \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}ForSeq(p, seqExp, stm)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta')
\end{array}$$

Stmt ForSet

$$\frac{\begin{array}{l} classes, cpus, pending, o \vdash [\![setExp]\!] = (set, \delta_e) \\ stms = [mk\text{-}PartialLetDef(\sigma, [\,], stm) \mid i \in \mathbf{inds}\, set2seq(set) \wedge \sigma = match(p, seq(i))] \\ \tau, classes, cpus, c, t, o \vdash (stms \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \delta' = \delta_e + \delta + ForSetTime \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}ForSet(p, setExp, stm)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta') \end{array}}$$

Other VDM-RT specific

The Stmt Cycle rule converts the cycle statement into a duration where the time is calculated based on the current CPU speed.

Stmt Cycle

$$\frac{\begin{array}{l} classes, cpus, pending, o \vdash [\![e]\!] = (value, \text{-}) \\ value \geq 0 \\ time = convertCyclesToTime(value, cpu.speed) \\ dur = mk\text{-}Duration(time, stm) \\ \tau, classes, cpus, c, t, o \vdash ([dur] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}Cycles(e, stm)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \end{array}}$$

Lets

The let-statements are divided into two groups: Stmt LetDef and Stmt LetBe. Both are converted into partial let defs. The Stmt LetDef rule directly rewrites the let def statement into a partial let def whereas the Stmt LetBe rules first bind the patterns and then tests if the expression evaluates to true.

Stmt LetDef

$$\frac{\begin{array}{l} stms = [mk\text{-}PartialLetDef(\{\,\}, defs, body)] \curvearrowright rest \\ \tau, classes, cpus, c, t, o \vdash (stms, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \delta' = \delta + LetDefTime \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}LetDef(defs, body)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta') \end{array}}$$

Stmt LetBe

$$\frac{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash (bind, pending, cpu) \xrightarrow{bind} (ps, \sigma') \\ pending' = pending \dagger \{o \mapsto (pending(o) \dagger \sigma')\} \\ classes, cpus, pending', o \vdash [\![e]\!] = (\mathbf{true}, \delta_e) \\ stms = [mk\text{-}PartialLetDef(\sigma', [\,], body)] \curvearrowright rest \\ \tau, classes, cpus, c, t, o \vdash (stms, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta) \\ \delta' = \delta_e + \delta + LetBeTime \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}LetBe(bind, e, body)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta') \end{array}}$$

The Stmt PartialLetDef Step rule steps through the definitions and evaluates each definition until the sequence of definitions becomes empty.

Stmt PartialLetDef Step

$$\frac{\begin{array}{l}
partialletdef = \mathit{mk\text{-}PartialLetDef}(\sigma, defs, body) \\
(id_v, e) = \mathbf{hd}\, defs \\
pending' = pending \dagger \sigma \\
classes, cpus, pending', o \vdash [\![e]\!] = (v, \delta_e) \\
\sigma' = \sigma \dagger \{id_v \mapsto v\} \\
stms = [\mathit{mk\text{-}PartialLetDef}(\sigma', \mathbf{tl}\, defs, body)] \curvearrowright rest \\
\tau, classes, cpus, c, t, o \vdash (stms, pending, cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta) \\
\delta' = \delta_e + \delta
\end{array}}{\begin{array}{l}
\tau, classes, cpus, c, t, o \vdash \\
([partialletdef] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta')
\end{array}}$$

The Stmt PartialLetDef Eval Complete does a full evaluation of the partial let def in a way that either the sequence of statements becomes empty or is a single, fully evaluated return statement. In either case, the sequence of statements resulting from the partial let def is concatenated with the sequence of statements from the thread body and evaluation is continued.

Stmt PartialLetDef Eval Complete

$$\frac{\begin{array}{l}
partialletdef = \mathit{mk\text{-}PartialLetDef}(\sigma, [\,], body) \\
pending' = pending \dagger \sigma \\
\tau, classes, cpus, c, t, o \vdash ([body], pending', cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta_{body}) \\
rest' = [\,] \vee (rest' = [\mathit{mk\text{-}Return}(v)] \wedge v \in \mathit{VDMValue}) \\
rest'' = rest' \curvearrowright rest \\
pending''' = (\mathbf{dom}\, \sigma \mathbin{\triangleleft\!\!\!-} pending'') \dagger (\mathbf{dom}\, \sigma \triangleleft pending) \\
\tau, classes, cpus, c, t, o \vdash (rest'', pending''', cpu', busses') \xrightarrow{stmt} (rest''', pending'''', cpu'', busses'', \delta) \\
\delta' = \delta_{body} + \delta
\end{array}}{\begin{array}{l}
\tau, classes, cpus, c, t, o \vdash \\
([partialletdef] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest''', pending'''', cpu'', busses'', \delta')
\end{array}}$$

The Stmt PartialLetDef Eval Waiting rule evaluates a part of the statements in the partial let def. It wraps the rest in a new partial let def and adds the new partial let def at the head of the thread body.

Stmt PartialLetDef Eval Waiting

$$\frac{\begin{array}{l}
partialletdef = \mathit{mk\text{-}PartialLetDef}(\sigma, [\,], body) \\
pending' = pending \dagger \sigma \\
\tau, classes, cpus, c, t, o \vdash ([body], pending', cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta) \\
rest' \neq [\,] \wedge rest' \neq [\mathit{mk\text{-}Return}(\text{-})] \\
pending''' = (\mathbf{dom}\, \sigma \mathbin{\triangleleft\!\!\!-} pending'') \dagger (\mathbf{dom}\, \sigma \triangleleft pending) \\
\sigma' = \mathbf{dom}\, \sigma \triangleleft pending'' \\
rest'' = [\mathit{mk\text{-}PartialLetDef}(\sigma', [\,], \mathit{mk\text{-}SimpleBlock}(rest'))] \curvearrowright rest
\end{array}}{\begin{array}{l}
\tau, classes, cpus, c, t, o \vdash \\
([partialletdef] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest'', pending''', cpu', busses', \delta)
\end{array}}$$

Assignments

There are two kinds of assignment/atomic statements, either the target has a Id_v indicating a local assignment or it has a $Id_o \times Id_v$ indicating that it is a remote assignment. The assignment evaluates the expression and stores the result in pending for the object specified followed by an invariant check.

$\boxed{\text{Stmt Assign Local}}$

$$\begin{array}{l}
target \in Id_v \\
classes, cpus, pending, o \vdash [\![e]\!] = (value, \delta_e) \\
\sigma' = pending(o) \dagger \{target \mapsto value\} \\
pending' = pending \dagger \{o \mapsto \sigma'\} \\
obj = cpu.objects(o) \\
checkInvs(classes(obj.class).invs, obj.state \dagger \sigma') \\
\tau, classes, cpus, c, t, o \vdash (rest, pending', cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta) \\
\delta' = \delta_e + \delta + LocalAssignmentTime \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}Assignment(target, e)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta')
\end{array}$$

$\boxed{\text{Stmt Assign Remote}}$

$$\begin{array}{l}
target = (oid, v) \\
classes, cpus, pending, o \vdash [\![e]\!] = (value, \delta_e) \\
\sigma' = pending(oid) \dagger \{v \mapsto value\} \\
pending' = pending \dagger \{oid \mapsto \sigma'\} \\
obj = cpu.objects(oid) \\
checkInvs(classes(obj.class).invs, obj.state \dagger \sigma') \\
\tau, classes, cpus, c, t, o \vdash (rest, pending', cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta) \\
\delta' = \delta_e + \delta + RemoteAssignmentTime \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}Assignment(target, e)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta')
\end{array}$$

The atomic statements are defined into three steps. First of all, the Stmt Atomic Start rule converts the atomic statement into a partial atomic statement. Secondly, either the Stmt Atomic Local or Stmt Atomic Remote performs the actual assignment. And finally, the Stmt Atomic Base is the recursive base case for the atomic assignments that when all assignments are done checks that the invariants still hold.

$\boxed{\text{Stmt Atomic Start}}$

$$\begin{array}{l}
stmts = [mk\text{-}PartialAtomic(assigns, \{\,\})] \curvearrowright rest \\
\tau, classes, cpus, c, t, o \vdash (stmts, pending, cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta') \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}Atomic(assigns)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending'', cpu', busses', \delta')
\end{array}$$

$\boxed{\text{Stmt PartialAtomic Base}}$

$$\begin{array}{l}
states = \{oid \mapsto (cpu.objects(oid).state \dagger pending(oid)) \mid (oid, \sigma) \in cpu.objects\} \\
invs = \{oid \mapsto (classes(cpu.objects(oid).class).invs) \mid oid \in \mathbf{dom}\; cpu.objects\} \\
\forall oid \in \mathbf{dom}\; cpu.objects \cdot checkInvs(invs(oid), states(oid)) \\
\delta = AtomicTime \\
\hline
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}PartialAtomic([\,], oids)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} ([\,], pending, cpu, busses, \delta)
\end{array}$$

$\boxed{\text{Stmt PartialAtomic Local}}$

$$\frac{\begin{array}{l}
\mathbf{hd}\ assigns = mk\text{-}Assignment(target, exp) \\
target \in Id_v \\
classes, cpus, pending, o \vdash \llbracket exp \rrbracket = (value, \delta_e) \\
\sigma' = pending(o) \dagger \{target \mapsto value\} \\
pending' = pending \dagger \{o \mapsto \sigma'\} \\
oids' = oids' \cup \{o\} \\
rest' = [mk\text{-}PartialAtomic(\mathbf{tl}\ assigns, oids')] \curvearrowright rest \\
\tau, classes, cpus, c, t, o \vdash (rest', pending', cpu, busses) \xrightarrow{stmt} (rest'', pending'', cpu', busses', \delta) \\
\delta' = \delta_e + \delta + LocalAssignmentTime
\end{array}}{\begin{array}{l}
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}PartialAtomic(assigns, oids)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} \\
\quad (rest'', pending'', cpu', busses', \delta')
\end{array}}$$

$\boxed{\text{Stmt PartialAtomic Remote}}$

$$\frac{\begin{array}{l}
\mathbf{hd}\ assigns = mk\text{-}Assignment(target, exp) \\
target = (id_o, id_v) \\
classes, cpus, pending, o \vdash \llbracket exp \rrbracket = (value, \delta_e) \\
\sigma' = pending(id_o) \dagger \{id_v \mapsto value\} \\
pending' = pending \dagger \{id_o \mapsto \sigma'\} \\
oids' = oids' \cup \{id_o\} \\
rest' = [mk\text{-}PartialAtomic(\mathbf{tl}\ assigns, oids')] \curvearrowright rest \\
\tau, classes, cpus, c, t, o \vdash (rest', pending', cpu, busses) \xrightarrow{stmt} (rest'', pending'', cpu', busses', \delta) \\
\delta' = \delta_e + \delta + LocalAssignmentTime
\end{array}}{\begin{array}{l}
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}PartialAtomic(assigns, oids)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} \\
\quad (rest'', pending'', cpu', busses', \delta')
\end{array}}$$

Threads

The Stmt Start rule starts a new thread for an object and updates the CPU with the new thread. The rule requires that no other thread exists for the *obj* in the start statement. The new thread will be created with the set to the initial field of the its class.

$\boxed{\text{Stmt Start}}$

$$\frac{\begin{array}{l}
\forall thread \in \mathbf{rng}\ cpu.threads \cdot thread.context \neq obj \\
body = classes(cpu.objects(obj).class).initial \\
body \in (Duration \mid PartialDuration)^* \\
cpu' = createThread(cpu, obj, body) \\
\tau, classes, cpus, c, t, o \vdash (rest, pending, cpu', busses) \xrightarrow{stmt} (rest', pending', cpu'', busses', \delta) \\
\delta' = \delta + StartTime
\end{array}}{\begin{array}{l}
\tau, classes, cpus, c, t, o \vdash \\
([mk\text{-}Start(obj)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu'', busses', \delta')
\end{array}}$$

Object Context Switch

The two object context rules are used to control under what object the execution takes place. The *ObjectContext* record contains the *oid* that should be used to execute the *body* and the *cctx* contains the call context that should be used to check any post-conditions.

The Stmt ObjectContext Step rule executes a part of the body of the object context and creates a new object context with the remainder of the body.

Stmt ObjectContext Step

$$\frac{\begin{array}{l}\tau, \mathit{classes}, \mathit{cpus}, c, t, \mathit{oid} \vdash ([\mathit{body}], \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) \\ \mathit{rest}' \neq [\,] \wedge \textbf{hd}\, \mathit{rest}' \notin \mathit{Return} \\ \mathit{rest}'' = [\mathit{mk\text{-}ObjectContext}(\mathit{oid}, \mathit{mk\text{-}SimpleBlock}(\mathit{rest}'), \mathit{cctx})] \curvearrowright \mathit{rest}\end{array}}{\begin{array}{l}\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ ([\mathit{mk\text{-}ObjectContext}(\mathit{oid}, \mathit{body}, \mathit{cctx})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} \\ \quad (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta)\end{array}}$$

The Stmt ObjectContext Complete rule completes the execution of the body until it is empty or contains a fully evaluated return statement. Then it uses the call context to check the post condition of the call.

Stmt ObjectContext Complete

$$\frac{\begin{array}{l}\tau, \mathit{classes}, \mathit{cpus}, c, t, \mathit{oid} \vdash ([\mathit{body}], \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) \\ \mathit{rest}' = [\,] \vee (\textbf{hd}\, \mathit{rest}' = [\mathit{mk\text{-}Return}(v)] \wedge v \in \mathit{VDMValue}) \\ \mathit{cctx} = \mathit{mk\text{-}CallContext}(\mathit{prepending}, \mathit{args}, \mathit{post}) \\ \mathit{checkCallPost}(\mathit{classes}, \mathit{cpus}, \mathit{oid}, \mathit{prepending}, \mathit{pending}', \mathit{args}, v, \mathit{post}) = \textbf{true} \\ \mathit{rest}'' = \mathit{rest}' \curvearrowright \mathit{rest} \\ \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}') \xrightarrow{\mathit{stmt}} (\mathit{rest}''', \mathit{pending}'', \mathit{cpu}'', \mathit{busses}'', \delta') \\ \delta'' = \delta + \delta'\end{array}}{\begin{array}{l}\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ ([\mathit{mk\text{-}ObjectContext}(\mathit{oid}, \mathit{body}, \mathit{cctx})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} \\ \quad (\mathit{rest}''', \mathit{pending}'', \mathit{cpu}'', \mathit{busses}'', \delta'')\end{array}}$$

Bindings

Type-Bind

$$\frac{\begin{array}{l}\textbf{len}\, p = 1 \\ x \in t \\ \sigma' = \mathit{match}(p(1), x)\end{array}}{\begin{array}{l}\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ (\mathit{mk\text{-}TypeBind}(p, t), \mathit{pending}, \mathit{cpu}) \xrightarrow{\mathit{bind}} (p, \sigma')\end{array}}$$

Multi-Type-Bind

$$\frac{\sigma' = \textbf{merge}\, \{\mathit{match}(p, x) \mid p \in \mathit{ps} \bullet x \in t\}}{\begin{array}{l}\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ (\mathit{mk\text{-}TypeBind}(\mathit{ps}, t), \mathit{pending}, \mathit{cpu}) \xrightarrow{\mathit{bind}} (\mathit{ps}, \sigma')\end{array}}$$

Set-Bind

$$\frac{\begin{array}{l}\textbf{len}\, p = 1 \\ \mathit{classes}, \mathit{cpus}, \mathit{pending}, o \vdash [\![e]\!] = \mathit{valueSet} \\ x \in \mathit{valueSet} \\ \sigma' = \mathit{match}(p(1), x)\end{array}}{\begin{array}{l}\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ (\mathit{mk\text{-}SetBind}(p, \mathit{set}), \mathit{pending}, \mathit{cpu}) \xrightarrow{\mathit{bind}} (p, \sigma')\end{array}}$$

Multi-Set-Bind

$$\frac{\begin{array}{l} classes, cpus, pending, o \vdash \llbracket e \rrbracket = valueSet \\ \sigma' = \textbf{merge}\,\{match(p, x) \mid p \in ps \wedge x \in valueSet\} \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ (mk\text{-}SetBind(p, set), pending, cpu) \xrightarrow{bind} (ps, \sigma') \end{array}}$$

Calls

There are four types of call rules dealing with a combination of synchronous/asynchronous and local/remote but common to all is that they all evaluate the arguments for the call and lookup the operation that should be executed. The Stmt Call Op Local Sync rule then performs the execution of the operation body through a $CallContext$ and $ObjectContext$. The Stmt Call Op Local Async rule instead creates a new thread with a synchronous call to the operation.

Stmt Call Op Local Sync

$$\frac{\begin{array}{l} opTarget = (oid, op) \\ argsTimed = [(value, \delta_e) \mid arg \in args \wedge (classes, cpus, pending, o \vdash \llbracket e \rrbracket = (value, \delta_e))] \\ args = [value \mid (value, \text{-}) \in argsTimed] \\ mk\text{-}Op(\text{-}, params, ret, body, pre, post) = classes(cpu.objects(oid).class).ops(op) \\ \sigma = \{p \mapsto a \mid i \in \textbf{inds}\, args \wedge a = args(i) \wedge params(i) = (p, \text{-})\} \\ checkCallPre(classes, cpus, pending, args, params, oid, pre) = \textbf{true} \\ callContext = mk\text{-}CallContext(pending, \sigma, post) \\ partialLetDef = mk\text{-}PartialLetDef(\sigma, [\,], mk\text{-}SimpleBlock(body)) \\ objContext = mk\text{-}ObjectContext(oid, partialLetDef, callContext) \\ callBlock = [objContext, mk\text{-}Wait(target)] \\ stms = callBlock \curvearrowright rest \\ \tau, classes, cpus, c, t, o \vdash (stms, pending, cpu, busses) \xrightarrow{stmt} (rest', pending', cpu', busses', \delta_{rest}) \\ \delta' = sum([\delta_e \mid (\text{-}, \delta_e) \in argsTimed]) + \delta_{rest} + LocalSyncCallTime \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}SyncCall(target, opTarget, args)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} \\ \quad (rest'', pending'', cpu'', busses'', \delta') \end{array}}$$

Stmt Call Op Local Async

$$\frac{\begin{array}{l} opTarget = (oid, op) \\ argsTimed = [(value, \delta_e) \mid arg \in args \wedge (classes, cpus, pending, o \vdash \llbracket e \rrbracket = (value, \delta_e))] \\ args = [value \mid (value, \text{-}) \in argsTimed] \\ mk\text{-}Op(\textbf{true}, params, ret, body, pre, post) = classes(cpu.objects(oid).class).ops(op) \\ cpu' = createThread(cpu, oid, mk\text{-}Duration(\textsc{ExecTime}, [mk\text{-}SyncCall(\textbf{nil}, opTarget, args)])) \\ \tau, classes, cpus, c, t, o \vdash (rest, pending, cpu', busses) \xrightarrow{stmt} (rest', pending', cpu'', busses', \delta) \\ \delta' = sum([\delta_e \mid (\text{-}, \delta_e) \in argsTimed]) + \delta + LocalAsyncCallTime \end{array}}{\begin{array}{l} \tau, classes, cpus, c, t, o \vdash \\ ([mk\text{-}AsyncCall(opTarget, args)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} \\ \quad (rest', pending', cpu'', busses', \delta') \end{array}}$$

The two remote call rules use the bus to communicate the call to the receiver instead of directly creating a synchronous call with the intended object identifier. The Stmt Call Op Remote Async is the simplest rule since it just adds a new $CMessage$ to the bus connecting the current CPU with the receiving CPU.

Stmt Call Op Remote Async

$$\frac{\begin{array}{l} \mathit{opTarget} = (\mathit{ccpu}, \mathit{oid}, \mathit{op}) \\ \mathit{argsTimed} = [(\mathit{value}, \delta_e) \mid \mathit{arg} \in \mathit{args} \land (\mathit{classes}, \mathit{cpus}, \mathit{pending}, o \vdash \llbracket e \rrbracket = (\mathit{value}, \delta_e))] \\ \mathit{args} = [\mathit{value} \mid (\mathit{value}, \text{-}) \in \mathit{argsTimed}] \\ \mathit{mk\text{-}Op}(\mathbf{true}, \mathit{params}, \mathit{ret}, \mathit{body}, \mathit{pre}, \mathit{post}) = \mathit{classes}(\mathit{cpu}.\mathit{objects}(\mathit{oid}).\mathit{class}).\mathit{ops}(\mathit{op}) \\ \mathit{busses}(\mathit{bus}) = \mathit{mk\text{-}Bus}(\{\mathit{ccpu}, c\} \cup \mathit{connected}, \mathit{speed}, \mathit{queue}) \\ \mathit{cmsg} = \mathit{mk\text{-}CMessage}(\mathit{oid}, \mathit{op}, \mathit{args}, \mathbf{nil}, \tau) \\ \mathit{busses}' = \mathit{busses} \dagger \{\mathit{bus} \to \mathit{mk\text{-}Bus}(\{\mathit{ccpu}, c\} \cup \mathit{connected}, \mathit{speed}, \mathit{queue} \curvearrowright [(\mathit{ccpu}, \mathit{cmsg})])\} \\ \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash (\mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}') \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}'', \delta) \\ \delta' = \mathit{sum}([\delta_e \mid (\text{-}, \delta_e) \in \mathit{argsTimed}]) + \delta + \mathit{RemoteAsyncCallTime} \end{array}}{\begin{array}{l} \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ ([\mathit{mk\text{-}AsyncCall}(\mathit{opTarget}, \mathit{args})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} \\ \quad (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}'', \delta') \end{array}}$$

The Stmt Call Op Remote Sync also adds a $\mathit{CMessage}$ to the bus, but instead of continuing with the execution it changes the current thread status to WAITING and stops the recursive execution.

Stmt Call Op Remote Sync

$$\frac{\begin{array}{l} \mathit{opTarget} = (\mathit{ccpu}, \mathit{oid}, \mathit{op}) \\ \mathit{argsTimed} = [(\mathit{value}, \delta_e) \mid \mathit{arg} \in \mathit{args} \land (\mathit{classes}, \mathit{cpus}, \mathit{pending}, o \vdash \llbracket e \rrbracket = (\mathit{value}, \delta_e))] \\ \mathit{args} = [\mathit{value} \mid (\mathit{value}, \text{-}) \in \mathit{argsTimed}] \\ \mathit{mk\text{-}Op}(\text{-}, \mathit{params}, \mathit{ret}, \mathit{body}, \mathit{pre}, \mathit{post}) = \mathit{classes}(\mathit{cpu}.\mathit{objects}(\mathit{oid}).\mathit{class}).\mathit{ops}(\mathit{op}) \\ \mathit{rest}' = [\mathit{mk\text{-}Wait}(\mathit{target})] \curvearrowright \mathit{rest} \\ \mathit{busses}(\mathit{bus}) = \mathit{mk\text{-}Bus}(\{\mathit{ccpu}, c\} \cup \mathit{connected}, \mathit{speed}, \mathit{queue}) \\ \mathit{cmsg} = \mathit{mk\text{-}CMessage}(\mathit{oid}, \mathit{op}, \mathit{args}, (c, t), \tau) \\ \mathit{busses}' = \mathit{busses} \dagger \{\mathit{bus} \to \mathit{mk\text{-}Bus}(\{\mathit{ccpu}, c\} \cup \mathit{connected}, \mathit{speed}, \mathit{queue} \curvearrowright [(\mathit{ccpu}, \mathit{cmsg})])\} \\ \mathit{cpu}' = \mathit{changeThreadStatus}(\mathit{cpu}, t, \text{WAITING}) \\ \delta' = \mathit{sum}([\delta_e \mid (\text{-}, \delta_e) \in \mathit{argsTimed}]) + \mathit{RemoteSyncCallTime} \end{array}}{\begin{array}{l} \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ ([\mathit{mk\text{-}SyncCall}(\mathit{target}, \mathit{opTarget}, \mathit{args})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} \\ \quad (\mathit{rest}', \mathit{pending}, \mathit{cpu}', \mathit{busses}', \delta') \end{array}}$$

Return

There are three rules that handle return statement evaluation. The Stmt Return Eval just evaluates the return expression in the current context and replaces the expression with its value.

Stmt Return Eval

$$\frac{\begin{array}{l} \mathit{exp} \notin \mathit{VDMValue} \\ \mathit{classes}, \mathit{cpus}, \mathit{pending}, o \vdash \llbracket \mathit{exp} \rrbracket = (\mathit{retValue}, \delta_e) \\ \mathit{rest}' = [\mathit{mk\text{-}Return}(\mathit{retValue})] \curvearrowright \mathit{rest} \\ \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash (\mathit{rest}', \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) \\ \delta' = \delta_e + \mathit{ReturnTime} \end{array}}{\begin{array}{l} \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ ([\mathit{mk\text{-}Return}(\mathit{exp})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta') \end{array}}$$

The Stmt Return Eat rule activates if the head of the statement is a return statement. The rule then removes the subsequent statement if it is not a wait statement.

$\boxed{\text{Stmt Return Eat}}$

$$\frac{\begin{array}{l} \mathit{rest} \neq [\,] \\ \textbf{hd}\, \mathit{rest} \notin \mathit{Wait} \\ \mathit{rest}' = [\mathit{mk\text{-}Return}(v)] \curvearrowright \textbf{tl}\, \mathit{rest} \\ \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash (\mathit{rest}', \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) \end{array}}{\begin{array}{l} \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ ([\mathit{mk\text{-}Return}(v)] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}'', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) \end{array}}$$

The Stmt Return Base is the base case for the recursive calls and stops the recursion when the statement being executed only contains a fully evaluated return statement.

$\boxed{\text{Stmt Return Base}}$

$$\frac{\begin{array}{l} v \in \mathit{VDMValue} \\ \mathit{stms} = [\mathit{mk\text{-}Return}(v)] \end{array}}{\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash (\mathit{stms}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{stms}, \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta)}$$

Completely evaluation of $\mathit{Return}, \mathit{Wait}$ The following rules deal with the final evaluation of the return from a function. The Stmt Wait Nil rule handles the cases where an async call was made and thus no thread is waiting for a reply; in this case the return and wait statements are removed from the rest.

$\boxed{\text{Stmt Wait Nil}}$

$$\frac{\tau, \mathit{classes}, \mathit{cpus}, c, t, \mathit{obj} \vdash (\mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta)}{\begin{array}{l} \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ ([\mathit{mk\text{-}Return}(\text{-}), \mathit{mk\text{-}Wait}(\textbf{nil})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} \\ \quad (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}', \delta) \end{array}}$$

The Stmt Return Wait rule handles local calls and updates the pending map with the return value and removed the return and wait from the rest.

$\boxed{\text{Stmt Return Wait}}$

$$\frac{\begin{array}{l} \mathit{target} \in \mathit{Id}_v \\ \sigma' = \mathit{pending}(o) \dagger \{\mathit{target} \mapsto v\} \\ \mathit{pending}' = \mathit{pending} \dagger \{o \mapsto \sigma'\} \\ \tau, \mathit{classes}, \mathit{cpus}, c, t, \mathit{obj} \vdash (\mathit{rest}, \mathit{pending}', \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}'', \mathit{cpu}', \mathit{busses}', \delta) \end{array}}{\begin{array}{l} \tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\ ([\mathit{mk\text{-}Return}(v), \mathit{mk\text{-}Wait}(\mathit{target})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} \\ \quad (\mathit{rest}', \mathit{pending}'', \mathit{cpu}', \mathit{busses}', \delta) \end{array}}$$

The Stmt Wait Message Return handles remove sync calls where a thread on another CPU is waiting for a reply. The return value is added to a $\mathit{RMessage}$ where the destination is defined by the target of the wait statement. The new message is then added to the bus and both the return and wait statement is removed from the sequence of statements.

Stmt Wait Message Return

$$\begin{array}{l}
\mathit{target} = (r_c, r_t) \\
\mathit{rmsg} = \mathit{mk\text{-}RMessage}(v, \mathit{target}, \tau) \\
\mathit{busses}(\mathit{bus}) = \mathit{mk\text{-}Bus}(\{r_c, c\} \cup \mathit{connected}, \mathit{speed}, \mathit{queue}) \\
\mathit{busses}' = \mathit{busses} \dagger \{\mathit{bus} \to \mathit{mk\text{-}Bus}(\{r_c, c\} \cup \mathit{connected}, \mathit{speed}, \mathit{queue} \curvearrowright [(r_c, \mathit{rmsg})])\} \\
\tau, \mathit{classes}, \mathit{cpus}, c, t, \mathit{obj} \vdash (\mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}') \xrightarrow{\mathit{stmt}} (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}'', \delta) \\
\hline
\tau, \mathit{classes}, \mathit{cpus}, c, t, o \vdash \\
([\mathit{mk\text{-}Return}(v), \mathit{mk\text{-}Wait}(\mathit{target})] \curvearrowright \mathit{rest}, \mathit{pending}, \mathit{cpu}, \mathit{busses}) \xrightarrow{\mathit{stmt}} \\
\quad (\mathit{rest}', \mathit{pending}', \mathit{cpu}', \mathit{busses}'', \delta')
\end{array}$$

A.4 Utility Functions

$\mathit{match}\colon \mathit{Pattern} \times \mathit{VDMValue} \to \Sigma$
$\mathit{match}(p, v)\sigma' ==$
Returns a new map with all variables that could be bound to the pattern

Figure A.1: Pattern matching function

$\mathit{FV}\colon \mathit{Pattern} \to \mathit{Id}$-**set**
$\mathit{FV}(p)\mathit{ids}' ==$
Returns a set of all free variables in the pattern

Figure A.2: Returns a set of free variables from a Pattern

$\mathit{collapseOld}\colon \mathit{Id}$-**set** $\to \mathit{Id}$-**set**
$\mathit{collapseOld}(p)\mathit{ids}' ==$
Replaces all old ids in the set with the original id

Figure A.3: Returns a set of ids consisting of all ids from the input set, old ids are resolved to the original id

$\mathit{Old}\colon \mathit{Id}_o$-**set** $\to \mathit{Id}_o$-**set**
$\mathit{Old}(p)\mathit{ids}' ==$
returns the old id for a given object

Figure A.4: Returns a set of old ids for the set of Id_o given

$\mathit{checkInvs}\colon \mathit{Fun}$-**set** $\times \Sigma \to \mathbb{B}$
$\mathit{checkInvs}(\mathit{invs}, \sigma)b' ==$
$\forall f \in \mathit{invs} \cdot \sigma \vdash [\![f.\mathit{body}]\!] = \textbf{true}$

Figure A.5: Checks the set of boolean functions against the current state

$\mathit{convertCyclesToTime}\colon \mathit{VDMValue} \times \mathit{CPU} \to \Sigma$
$\mathit{convertCyclesToTime}(v, c)\mathit{time}' == \frac{v}{c.\mathit{speed}}$
Returns a time based on the number of cycles and the speed of the cpu

Figure A.6: Cycles conversion function

$checkCallPre\colon Classes \times Cpus \times Pending \times VDMValue^* \times (Id_v \times Type)^* \times Id_o \times Exp \to \mathbb{B}$
$checkCallPre(classes, cpus, pending, args, params, obj, pre)result' ==$
$typeCheck(params, args)$
$\wedge\ classes, cpus, pending, obj \vdash [\![pre]\!] = \textbf{true}$

Figure A.7: Check pre-conditions for operation calls

$checkCallPost\colon Classes \times Cpus \times Id_o \times Pending \times Pending \times \Sigma \times VDMValue^* \times Exp \to \mathbb{B}$
$checkCallPost(classes, cpus, oid, prepending, pending, args, post)result' ==$

```
  let $oldPending = \{old(v) \mapsto prepending(v) \mid v \in \textbf{dom}\, prepending\}$,
      $state = pending \dagger pending$  in
          $classes, cpus, state, obj \vdash [\![post]\!] = \textbf{true}$
```

Figure A.8: Check post-conditions for operation calls

$typeCheck\colon (Id_v \times Type)^* \times VDMValue^* \to \mathbb{B}$
$typeCheck(params, args)result' ==$
$\textbf{len}\, params = \textbf{len}\, args \wedge$
$\textbf{false} \notin [typeOf(v) = type \mid i \in \textbf{inds}\, params \wedge (\text{-}, type) = params(i) \wedge v = args(i)]$

Figure A.9: Type check operation call arguments

$typeOf\colon VDMValue \to Type$
$typeOf(v)type' ==$
Returns the type of a VDMValue

Figure A.10: Gives the type of a VDMValue

$changeThreadStatus\colon CPU \times Id_t \times (\textsc{Running} \mid \textsc{Runnable} \mid \textsc{Waiting} \mid \textsc{Pending} \mid \textsc{Completed}) \to CPU$
$changeThreadStatus(cpu, t, newStatus)cpu' ==$

```
let $mk\text{-}CPU(objects, threads, speed) = cpu$,
    $mk\text{-}Thread(\text{-}, pending, context, body) = threads(t)$,
    $threads' = threads \dagger \{t \to mk\text{-}Thread(newStatus, pending, context, body)\}$
    in
$mk\text{-}CPU(objects, threads', speed)$
```

Figure A.11: Change the status of a thread

$createThread\colon CPU \times Id_o \times Duration^* \to CPU$
$createThread(cpu, oid, body)cpu' ==$

```
let $t \in Thread$
    $t \notin \textbf{dom}\, cpu.threads$
    $thread = mk\text{-}Thread(\textsc{Runnable}, \{\,\}, oid, body, \textbf{nil})$
    $threads = cpu.threads \dagger \{t \to thread\}$
    in
$mk\text{-}CPU(cpu.objects, threads, cpu.speed)$
```

Figure A.12: Creates a new thread

```
$insertReturn \colon Stm^* \times Return \to [Stm]$
$insertReturn(stms, return) stms' ==$
cases hd $stms$ of
                    $mk\text{-}Wait(target) \to [return, mk\text{-}Wait(target)] \curvearrowright$ tl $rest$
$mk\text{-}ObjectContext(id_o, body) \to$ let $b = insertReturn(body, return)$
                                          $r = [mk\text{-}ObjectContext(id_o, b)]$ in
                                      $r \curvearrowright$ tl $rest$
$mk\text{-}PartialLetDef(\sigma, [\,], stm) \to$ let $sb = mk\text{-}SimpleBlock(insertReturn([stm], return))$ in
                                      $[mk\text{-}PartialLetDef(\sigma, [\,], sb)] \curvearrowright$ tl $rest$
end
```

Figure A.13: Inserts a return next to a inner Wait statement

```
$sumMapRange \colon VDMValue \xrightarrow{m} Number \to Number$
$sumMapRange(map) ==$
  if $map = \{\,\}$
  then $0$
  else let $key \in$ dom $map$ in $map(key) + sumMapRange(\{key\} \lessdot map)$
```

Figure A.14: Sums the range of a map structure

```
$sum \colon Number^* \to Number$
$sum(seq) ==$
  if $seq = [\,]$
  then $0$
  else hd $seq + sum($tl $seq)$
```

Figure A.15: Sums a sequence of numbers

```
$set2seq \colon VDMValue$-set $\to VDMValue^*$
$set2seq(set) ==$
  if $set = \{\,\}$
  then $[\,]$
  else let $item \in set$ in $[item] \curvearrowright set2seq(set - \{item\})$
```

Figure A.16: Sums the range of a map structure

Index of Rules and Definitions

Kenneth Lausdahl, Joey W. Coleman and Peter Gorm Larsen ,
Semantics of the VDM Real-Time Dialect, 2013

Department of Engineering
Aarhus University
Edison, Finlandsgade 22
8200 Aarhus N
Denmark

Tel.: +45 4189 3000