Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

FMI-Compliant Code Generation in the INTO-CPS Tool Chain

Deliverable Number: D5.3d

Version: 1.0

Date: December, 2017

Public Document

http://into-cps.au.dk

Contributors:

Victor Bandur, AU
Miran Hasanagić, AU
Adrian Pop, LIU
Marcel Groothuis, CLP

Editors:

Victor Bandur, AU

Reviewers:

Julien Ouy, CLE
Etienne Brosse, ST
Frederik Foldager, AI

Consortium:

Aarhus University	AU	Newcastle University	UNEW
University of York	UY	Linköping University	LIU
Verified Systems International GmbH	VSI	Controllab Products	CLP
ClearSy	CLE	TWT GmbH	TWT
Agro Intelligence	AI	United Technologies	UTRC
Softeam	ST		

Document History

Ver	Date	Author	Description
0.01	11-01-2017	Victor Bandur	Initial document version.
1.0	19-12-2017	Victor Bandur	Final document version.

Abstract

This deliverable details the state of the INTO-CPS tool chain code generation capability at the end of project Year 3. Code generation is spread across the three tools OpenModelica, 20-sim and Overture. With respect to FMI, all tools have the ability to export standalone FMUs.

Contents

1 Introduction

This deliverable describes the maturing code generation capability of the INTO-CPS tool chain at the end of Year 3 of the INTO-CPS project. Code generation is spread across the three tools OpenModelica [Lin15], 20-sim [Con13] and Overture [LBF+10]. In comparison to the code generation capabilities of OpenModelica and 20-sim, which rely on code generation for simulation, Overture's C code generator was developed from scratch for INTO-CPS. This has resulted in a corresponding emphasis on Overture's code generator in this deliverable.

1.1 Summary of Changes in Year 3

Both the OpenModelica and 20-sim code generators have been improved since project Year 2, but these changes are only minor. The new status of these code generators is captured in Sections 3 and 4 with only minor differences from what is reported in the previous version of this deliverable. These minor changes are as follows:

- **OpenModelica** New support for embedded platforms through the Modelica_DeviceDrivers library. This is reported briefly in Section 3.
- **20-sim** New support for the `timeevent()` function and for getting and setting complete FMU state. Support for the 3D visualisation FMU has been moved out of 20-sim, as previously reported, and is now described in the INTO-CPS user manual, Deliverable D4.3a [BLL+17]. This is reported briefly in Section 4.

The most substantial update over the Year 2 version of this deliverable is contained in Section 5.4, where new code generator support in Overture for distributed VDM-RT architectures is described. Nevertheless, as most of this deliverable is dedicated to describing code generator support in Overture, the reader would benefit from reading all of Section 5, as minor updates appear throughout.

2 Background and Related Work

This section introduces the three modelling notations Modelica, bond graphs and VDM, and some existing work on code generation for each. Citations

for these formalisms are given herein.

2.1 Modelica

Modelica [FE98], [Fri04] is an object-oriented, equation-based language for conveniently modelling complex physical systems containing, *e.g.*, mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents. The Modelica language supports continuous, discrete and hybrid time simulations.

The Modelica language has been designed to allow tools to automatically generate efficient simulation code with the main objective of facilitating exchange of models, model libraries, and simulation specifications. The definition of simulation models is expressed in a declarative manner, modularly and hierarchically. Various formalisms can be expressed in the more general Modelica formalism. In this respect Modelica has a multi-domain modeling capability which gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic *etc.* model components within the same application model.

Several tools exist that support code generation from the Modelica language. These are the commercial tools Dymola[1], SimulationX[2] and MapleSim[3]; and the open-source tools OpenModelica[4] and JModelica[5]. Most of these tools generate C or C++ code and can also generate FMUs.

2.2 SIDOPS+ and Bond Graphs

SIDOPS (Structured Interdisciplinary Description Of Physical Systems) is a computer language developed for the description of models and submodels of physical systems [Bro90]. It is designed to express bond-graph models that describe domain-independent engineering systems. 20-sim uses the SIDOPS+ version of the language, the key features of which are discussed below.

[1] `http://http://www.modelon.com/products/dymola/`.
[2] `http://www.simulationx.com/`.
[3] `http://www.maplesoft.com/products/maplesim/`.
[4] `http://www.openmodelica.org/`.
[5] `http://jmodelica.org/`.

SIDOPS+ [BB97] enhances support for organizing complex systems as a hierarchy of submodels by separating the interface of a model from its specification. This enables the creation of different specifications for one interface. In addition, SIDOPS+ supports different representations of model descriptions within three abstraction levels [BB97]:

- At the *technical component* level, models describe networks of devices which are represented by component graphs.
- At the *physical concept* level, models capture the physical processes of a system and can be expressed using graphical formalisms.
- At the *mathematical* level, models provide the quantitative description of the physical processes, written in the form of acausal equations or sequential statements (computer code) that calculate output variable values from the input variables.

All representations are port-based networks, meaning that the connection points between model elements is the location where exchange of information (signals) or power takes place. As a result, it is possible to map one representation to another without losing consistency. Similar to Modelica, the SIDOPS+ language supports continuous, discrete and hybrid time simulations by offering special functions for determining the sample interval for discrete-time variables that are linked through equations, and for creating continuous signals out of discrete input signals.

In 20-sim a model can be defined graphically, similar to drawing an engineering schematic, or using equations based on the SIDOPS+ language. Such models can be used to simulate and analyze the behaviour of multi-domain dynamic systems using, *e.g.*, mechanical, electrical, hydraulic, thermal and control components.

Systems can be modelled in 20-sim using a variety of modelling formalisms:

- Block diagrams
- Bond graphs
- Iconic diagrams
- Mathematical equations
- System descriptions (state space, transfer function)

Different formalisms can be freely combined within one model (mixed model).

Graphical models in 20-sim are built from pre-built library blocks or custom-made blocks. These blocks are called "submodels". They are implemented using either a graphical representation or equations.

Using graphical implementations inside submodels allows for hierarchical modelling. 20-sim supports unlimited levels of hierarchy in the model. The highest hierarchical levels in the model typically consist of graphical models (state space models, block diagrams, bond graphs or components). The lowest level in the hierarchy is always formed by equation models written in the SIDOPS+ language.

20-sim supports ANSI-C and C++ code generation for a large part of the SIDOPS+ language. The focus for the code generator is on generating code with real-time capabilities. Other tools that can generate code from bond graphs include CAMP-G[6], which generates MATLAB code; MS1[7] which can generate C and MATLAB code; and PSM++[8] which can generate Pascal code.

2.3 VDM

The Vienna Development Method (VDM) [Bjø79] is a formal software development notation and method based on formal proof of specification properties. The core specification language of VDM is called VDM-SL. Specifications written in VDM-SL are based on a central system state. Modifications to the state define the overall behaviour of the system being specified. The facilities of the language include fundamental types such as $\mathbb{R}$ and $\mathbb{N}^+$, function and operation pre- and post-conditions, state invariants, user-defined types with invariants *etc.*

The first level of development of VDM sees a move from VDM-SL to the language VDM++, an object-oriented extension. In VDM++ it is possible to make use of class-based structuring of specifications such that portions can be reused across specifications, just as encapsulation can be exploited for reuse in object-oriented programming languages. Indeed, object-orientation in VDM++ was inspired by object-orientation in programming languages.

The second expansion of the language results in the dialect VDM-RT. Beside all the object-oriented features of VDM++, VDM-RT adds facilities

[6] `http://www.bondgraph.com`.
[7] `http://www.lorsim.be`.
[8] `http://www.raczynski.com/pn/pn.htm`.

for capturing timing behaviour and specifying distributed system architectures.

Code generation for the various dialects of the (VDM) is implemented in two VDM support tools. The original VDMTools [CSK07, FLS08] implements Java and C++ code generators for VDM++. The follow-up open-source alternative, Overture, provides a Java code generator and a C code generator that is currently under active development. The target language for the Overture code generators is VDM-RT, specifically its features for distributed architectures.

3 FMI Code Generation with OpenModelica

OpenModelica is an open-source Modelica-based modeling and simulation environment. Modelica is an object-oriented, equation based language to conveniently model and simulate complex multi-domain physical systems. The OpenModelica environment supports graphical composition of Modelica models. Models are simulated via translation to FMU, C or C++ code. Compilation of Modelica models in OpenModelica happens in several phases

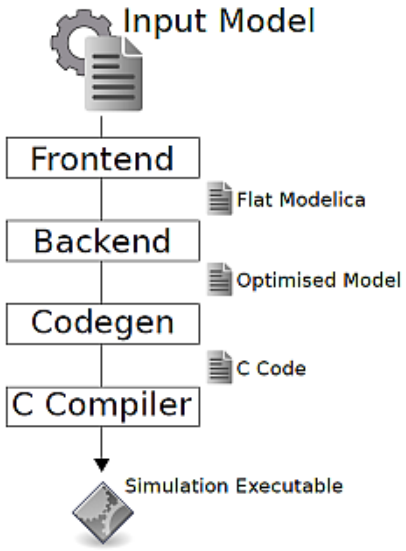

Figure 1: OpenModelica compilation phases.

[Sjö15] (see also Figure 1):

- Frontend - removes object orientation structures and builds the hybrid Differential Algebraic Equations (DAE) system to be solved.
- Backend - the hybrid DAE system is index reduced, transformed to causal form (sorted), and optimized.
- Codegen - the optimized system of equation is transformed to FMU, C or C++ code using a template language.

We now describe briefly the design principles behind code generation in the OpenModelica simulator. The OpenModelica simulator transforms Modelica code into different lower level languages that can be compiled into executable code. Currently OpenModelica can generate C, C++ and JavaScript code. Additionally, the OpenModelica simulator can generate FMUs compliant with both FMI 1.0 and 2.0 for model exchange and co-simulation.

The transformation from Modelica into executable code consists of several phases (see also Figure 2):

- Flattening - removal of object orientation from the Modelica language and creation of a hybrid DAE system.
- Basic Optimization - optimization of the hybrid DAE system, index reduction, matching, equation sorting, causalization.
- Advanced Optimization - more optimization of the system of equations, alias elimination, tearing, common sub-expression elimination, *etc.*
- Independent Simulation Code - the final system of equations is transformed into an independent simulation code structure.
- Code Generation - the Independent Simulation Code structure is given to several templates which can generate code in different languages, currently C, C++, JavaScript. The templates can also package FMUs.
- Simulation - the code is compiled into a standalone executable from the generated code and executed.

The code generation templates are written in the OpenModelica template language Susan [FPSP09].

As with the other INTO-CPS tools, OpenModelica's code generation capability is employed in generating FMI 1.0 and 2.0 FMUs for co-simulation. Only the forward Euler method of integration is available in the generated FMUs. These are source code FMUs which contain all the necessary source code files to be compiled for any target. With INTO-CPS, specifically for embedded targets, this process is facilitated by 20-sim 4C.

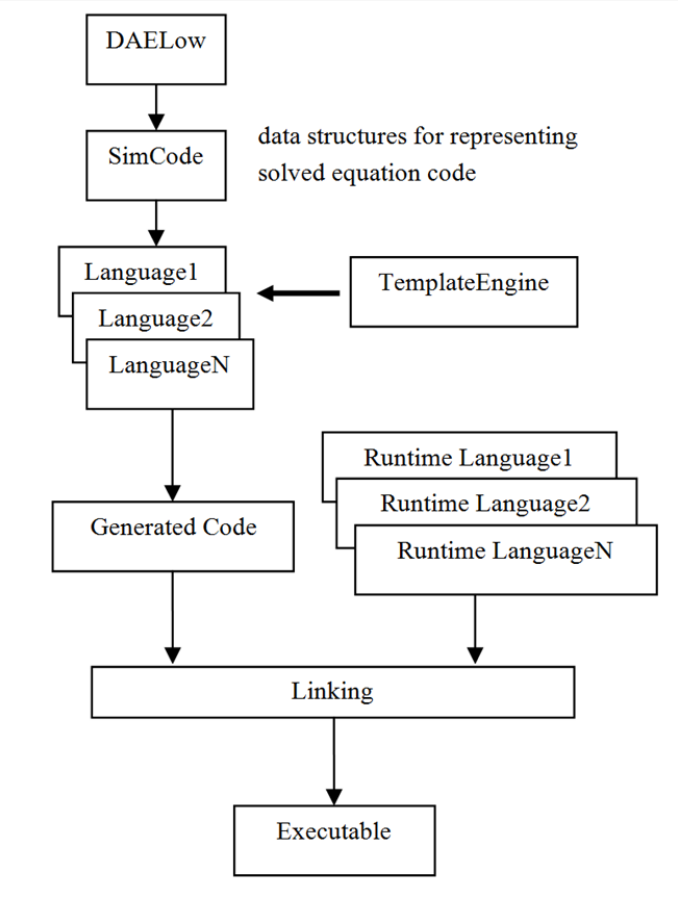

Figure 2: OpenModelica code generation using templates.

A direct way to compile to embedded target has also been implemented in OpenModelica [TBW+17] using a new restricted code generator and the Modelica_DeviceDrivers library.

4 FMI Code Generation with 20-sim

FMI code generation from 20-sim builds upon the application's existing code generation toolbox. 20-sim can generate ANSI-C and C++ code from a graphical or equation model. The generated code is passed to 20-sim 4C, a rapid prototyping application that takes the generated model code as input, combines it with target-specific code and prepares it for execution on a real-time target.

The main design principle behind the separation between 20-sim and 20-sim 4C is that a model should be independent of the actual target on which it should run. A model should contain only the necessary information of the target relevant for the simulation and no details specific or relevant to code generation. A typical 20-sim model contains no information about the intended target. The model can contain behavioural details about the target, such as the accuracy of an analog-to-digital converter, but detailed knowledge about the actual chip used and how to read values from this converter is not necessary for the simulation and is therefore not part of the model. As a consequence, 20-sim is not able, on its own, to produce standalone C code that can access specific hardware. It can only generate standalone C code that includes the model behaviour. This is enough for generating FMUs but not for running them on actual hardware.

4.1 Code Generation Principles

The 20-sim ANSI-C/C++ code is generated based on all SIDOPS+ equations inside the model. Figure 3 shows the flowchart of the 20-sim code generation process. The processing phase in 20-sim takes the graphical or equation model, flattens it and translates it into a hybrid Differential Algebraic Equation (DAE) system. This DAE system is transformed into a causal form (set of sorted equations). These sorted equations are then further optimized for both simulation and code generation purposes.

20-sim uses code generation templates to generate code for different purposes. One of these templates is the standalone FMU export template (for both FMI 1.0 and 2.0). 20-sim translates the optimized sorted equations into several blocks of ANSI-C code (*e.g.* initialization code, static equations, dynamic equations). These blocks are all stored in a token dictionary. Based on the token dictionary and the selected code generation template, the actual code is produced by means of a token replacement step. An example of the generated code can be found in deliverable D5.1d [HLG+15] (Section 6.2, Listing 3).

The FMU export template contains all functions that are required by the FMI 1.0 and/or 2.0 standard. These functions call the pre-defined 20-sim model functions for initialization, calculate steps and terminate. Besides 20-sim generated model functions, the template contains several pre-defined helper functions that implement ANSI-C versions of the SIDOPS+ language functions not directly supported in ANSI-C. Examples include matrix support functions, Table file read functionality and motion profile calculation

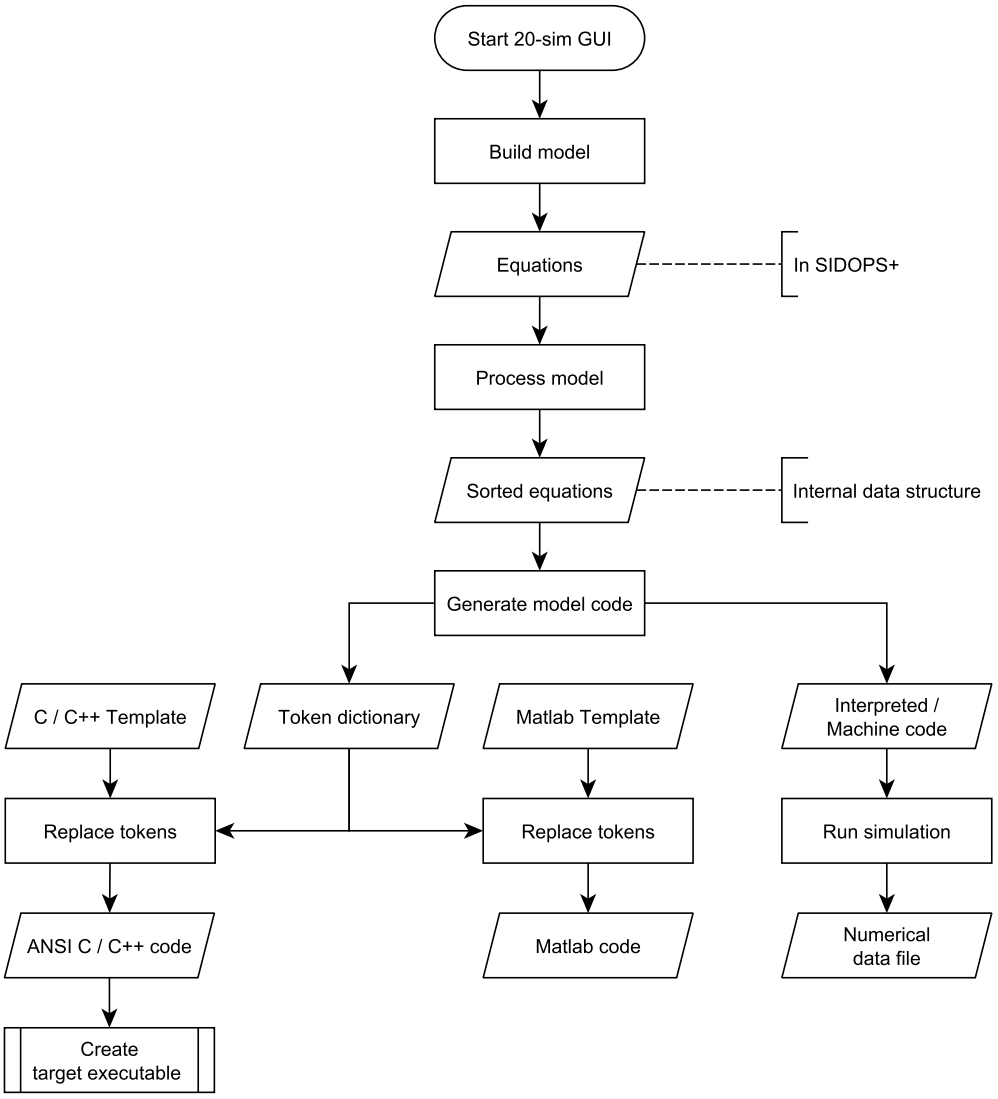

Figure 3: Flowchart of code generation from 20-sim.

functions. The latest version of the 20-sim Standalone FMU template can be found on GitHub:

`https://github.com/controllab/fmi-export-20sim`.

4.2 Code Generation Capabilities

Only a subset of the 20-sim modelling language elements can be exported as ANSI-C or C++ code. The exact supported features depend on the chosen template and its purpose. The main purpose of the 20-sim code generator is to export control systems. Therefore the focus is on executing the generated code on "bare-bone" targets (*i.e.* without operating system support, such as Arduino) or as a real-time task under a real-time operating system.

The following features are not, or are only partially supported for code generation in all templates. The FMI export template has no specific real-time goal, therefore this template supports more features than the other code generation templates.

- **Hybrid models:** Models that contain both discrete- and continuous-time sections cannot be generated at once. However, it is possible to export the continuous and discrete blocks separately.
- **External code:** Calls to external code are not supported. Examples are: `DLL()`, `DLLDynamic()` and the MATLAB functions.
- **Variable delays:** The `tdelay()` function is not supported due to the requirement for dynamic memory allocation.
- **Frequency Event function:** `frequencyevent()` statements are ignored in the generated code.
- **Fixed-step integration methods:** *Euler*, *Runge-Kutta 2* and *Runge-Kutta 4* are supported.
- **Variable-step integration methods:** *Vode-Adams* and *Modified Backward Differential Formula* (MeBDF) are only available in the development FMI export template found on GitHub at `https://github.com/controllab/fmi-export-20sim` (branch: MeBDFi). The variable step-size methods are not supported for all other code-generation templates due to their real-time constraints.
- **Implicit models:** Models that contain unsolved algebraic loops are not supported.
- **File I/O:** The 20-sim "Table2D" block is supported for FMU export. All other file-related functions are not supported.

4.3 FMU Capabilities

The FMI standard allows for many optional features. Here we summarize the most important characteristics of FMUs exported from 20-sim. Feature support includes the following:

- Co-simulation FMUs for both FMI 1.0 and FMI 2.0
- Standalone FMUs
- Real-time capability (when not using file I/O and variable step-size integration methods)
- Multi-instance support
- Support for both fixed-step-size and variable step-size methods
- FMUs which include source code
- Access to all model parameters and internal model variables
- Dynamic memory allocation (`canNotUseMemoryManagementFunctions = false`)
- Structured variable naming support (hierarchy using "`.`" and arrays using "`[ ]`")
- Getting and setting the complete FMU state

The following FMI features are not supported:

- Getting partial derivatives
- Definition of used units

5 FMI Code Generation with Overture

Code generation capability development for Overture was split into two stages. Unlike 20-sim and OpenModelica, which had mature code generation capabilities at the beginning of the INTO-CPS project, code generation to C for Overture was a new feature. The first step was to develop the code generation capability proper, the second to extend this to export of standalone FMUs. This section describes the design and implementation of the C code generator. This discussion is further divided into two parts. The first is concerned with code generation for the basic expression language of

VDM-SL and the object-oriented features of VDM++, whereas the other deals specifically with code generation of the distribution features of VDM-RT.

5.1 VDM and Code Generation

VDM can be used to specify systems at a very abstract level, as well as at a level that is concrete enough to be transliterated to any imperative programming language. Constructs such as pre- and post-conditions, state invariants and non-determinism facilitate the former, whereas constructs such as local variable declarations, assignments and loops facilitate the latter.

For instance, consider the following two specifications of a sorting algorithm, one very abstract, the other very concrete:

```
abstractSort(unsortedList : seq of int) sortedList : seq of int
pre   true
post  permutations(unsortedList, sortedList) and
      forall i, j in set inds sortedList &
          i <= j => sortedList(i) <= sortedList(j);
```

```
concreteSort : (seq of int) ==> (seq of int)
concreteSort(unsortedList) ==
(
  dcl sortedList : seq of int := unsortedList;
  dcl tmp : int;

  for all i in set inds sortedList do
    for all j in set inds sortedList do
      if sortedList(j) >= sortedList(i) then
      (
        tmp := sortedList(i);
        sortedList(i) := sortedList(j);
        sortedList(j) := tmp;
      );

  return sortedList;
)
```

The former assumes the existence of a function that confirms whether two sequences of integers are permutations of each other:

```
permutations : seq of int * seq of int -> bool
```

The abstract specification conveys its meaning very clearly in terms of the relationship that the resulting state must bear to the starting state. It forms a very clear starting point for implementation in any programming language. The meaning of the concrete specification is perhaps not as easily gleaned,

as it has an imperative flavour. But it is intentionally constructed like an imperative program, such that it can be easily transliterated into, say, a Java implementation:

```
private static List<Integer> concreteSort(List<Integer> unsortedList)
{
  List<Integer> sortedList = unsortedList;
  Integer tmp;

  for(int i = 0;  i < sortedList.size();  i++)
  {
    for(int j = 0; j < sortedList.size();  j++)
    {
      if(sortedList.get(j) >= sortedList.get(i))
      {
        tmp = sortedList.get(i);
        sortedList.set(i, sortedList.get(j));
        sortedList.set(j, tmp);
      }
    }
  }

  return sortedList;
}
```

The ability to choose the level of abstraction for any given specification not only facilitates a refinement-based approach to software development, but makes the method easy to use by system developers with very different expertise, from purely mathematical to fully focused on programming.

Refinement is crucial to code generation. Essentially, a code generator must embody a refinement strategy which is applied without human intervention in seemingly one step. The easiest way to implement a code generator is to keep the source language as concrete as possible. The example above demonstrates that, whereas the abstract specification of the sorting procedure can be implemented in any way that is correct *wrt* the post-condition, the concrete specification is a lot more direct, in that it provides unequivocal guidance toward the implementation of a bubble sort. It is important to note that it is the *meaning* of the specification that is essential. Its presentation, whether abstract or concrete, is a matter of practicality. Implementation of any other correct sorting procedure would satisfy the concrete specification, but the bubble sort is easiest to *derive* from the specification. The Overture C code generator starts from a restricted subset of VDM-RT that makes such derivation easiest.

5.2 Semantics of VDM-RT

The semantics of VDM-RT adopted by the Overture C code generator is fully documented in INTO-CPS deliverables D2.1b [FCL+15] and D2.2b [FCC+16]. The semantic work is rooted in Hoare and He's Unifying Theories of Programming (UTP) [HJ98]. Here we give a brief overview of the most important aspects of the semantics.

Features of VDM-SL The expression language of VDM-RT forms the basic mathematical core of the language. The mathematical vocabulary is standard and ranges over basic number domains, the Boolean domain, sets, sequences, maps *etc.* Expressions can refer to state variables, but they do not admit non-deterministic constructs, such as underspecified choice between values (*e.g.* VDM-RT's `let-be-such-that` construct.) Therefore, expression evaluation is deterministic *modulo* specification state, that is, an expression evaluated in the context of an instance of a class will always be deterministic relative to that object's state. The semantics of the expression language of VDM-RT is therefore assumed to be the standard mathematical one, and a direct semantic mapping into UTP expressions is assumed.

Features of VDM++ The feature of VDM++ of primary interest to code generation is object orientation. The semantics defines nine fundamental conditions governing the structure of a valid object-oriented specification in VDM++. Later, we show that the code emitted by the Overture C code generator is in conformance with these conditions, with a few justifiable exceptions. In brief, the conditions are:

- **OO1**: The special class *Object* is always a class of the system.
- **OO2**: Every class of the system has a superclass, except for *Object*.
- **OO2a**: Every class other than *Object* may have multiple direct superclasses.
- **OO3**: Every class has *Object* as a (not necessarily direct) superclass.
- **OO3a**: Cycles are not allowed in the case of multiple inheritance. That is, if class C inherits from classes A and B, then A and B may not themselves be in an inheritance relationship.
- **OO4**: Every class must define at least one attribute.
- **OO4a**: Every class must define an invariant.

- **OO5**: Attribute names are unique across classes.
- **OO6**: Each attribute is either of basic type, or of the type of one of the classes existing in the system.

These healthiness conditions form part of a theory of classes and object-orientation to which the semantics of VDM++ proper must conform. Further to the conditions on the static structure of an object-oriented specification, conditions are placed on object (class instance) behaviour. The effect of these conditions is as follows.

Upon creation of an instance of a class, all the attributes inherited from its superclasses are collected and associated with the instance being created. They are assigned default values nondeterministically for basic types and null references for class types. Access to overridden attributes along an inheritance chain (*e.g.* C inherits from B and B inherits from A) is resolved to the nearest overriding attribute in the chain.

In the presence of multiple inheritance, it is possible for several superclasses to define the same function or operation, creating ambiguity for the class inheriting from both simultaneously. Listing 1 illustrates this situation in VDM++.

Listing 1: Method declaration ambiguity in multiple inheritance context.

```
class A

operations
public op : () ==> bool
op() ==
  return true
end A

class B
operations
public op : () ==> bool
op() ==
  return false
end B

class C is subclass of B, A
end C

class D
instance variables
  obj : C := new C();

operations
public testop : () ==> bool
testop() ==
  return obj.op();
end D
```

The semantics dictates that in this circumstance a choice be made arbitrarily from the multiple definitions of the function or operation. However, Overture does not allow this sort of ambiguity in the specification (Overture considers the specification in Listing 1 invalid), eliminating both this and the more general problem of diamond inheritance. In a code generation context, therefore, this choice does not have to be made. The decision to disallow such specifications in Overture reflects a refinement of the semantics, and so consistency between semantics, existing tool support and tool support under development is maintained.

Features of VDM-RT The two additional features provided by VDM-RT, namely timing information and facilities for distributed architectures, are treated separately in Section 5.4.

5.3 Achieving Translation

The priority of the translation strategy is to remain faithful to the VDM-RT semantics described above. The strategy therefore assumes that VDM-RT specifications have been validated using Overture's various facilities. This section describes the strategy and the two sides of the translation mechanism, the implementation of the strategy and the native C support library. This section focuses specifically on the fundamental features of VDM, those provided by VDM-SL and VDM++. Section 5.4 discusses the translation of the additional features provided by VDM-RT.

5.3.1 Native Support Library

Implementations generated from VDM-RT models consist of two parts, the generated code and a native support library[9]. The native library is fixed and does not change during the code generation process. We illustrate its design here by means of very simple generated VDM models.

[9]The design of the native library is based on the following four sources:
`http://www.pvv.ntnu.no/~hakonhal/main.cgi/c/classes/`, accessed 2016-09-22.
`http://www.eventhelix.com/RealtimeMantra/basics/ComparingCPPAndCPerformance2.htm`, accessed 2016-09-22.
`http://www.go4expert.com/articles/virtual-table-vptr-multiple-inheritance-t16616/`, accessed 2016-09-22.
`http://www.go4expert.com/articles/virtual-table-vptr-t16544/`, accessed 2016-09-22.

The native library provides a single fundamental data structure in support of all the VDM-RT data types, called `TypedValue`. The complete definition is shown in Listing 2 (excerpt from previous work on integrating Overture with the TASTE toolset [FVB+16].) A pointer to `TypedValue` is `#defined` as `TVP`, and is used throughout the implementation.

Listing 2: Fundamental code generator data type.

```
typedef enum {
  VDM_INT, VDM_NAT, VDM_NAT1, VDM_BOOL, VDM_REAL, VDM_RAT, VDM_CHAR, VDM_SET,
  VDM_SEQ, VDM_MAP, VDM_PRODUCT, VDM_QUOTE, VDM_TOKEN, VDM_RECORD, VDM_CLASS
} vdmtype;

typedef union TypedValueType {
  void* ptr;              // VDM_SET, VDM_SEQ, VDM_CLASS, VDM_MAP, VDM_PRODUCT
  int intVal;             // VDM_INT, VDM_INT1 and VDM_TOKEN
  bool boolVal;           // VDM_BOOL
  double doubleVal;       // VDM_REAL
  char charVal;           // VDM_CHAR
  unsigned int quoteVal;// VDM_QUOTE
} TypedValueType;

struct TypedValue {
  vdmtype type;
  TypedValueType value;
};

struct Collection {
  struct TypedValue** value;
  int size;
  int buf_size;
};
```

An element of this type carries information about the type of the VDM value represented and the value proper. For space efficiency, the value storage mechanism is a C `union`.

Members of the basic, unstructured types `int`, `char`, *etc.* are stored directly as values in corresponding fields. Due to subtype relationships between certain VDM types, for instance `nat` and `nat1`, fields in the union can be reused. Functions to construct such basic values are provided:

- `TVP newInt(int)`
- `TVP newBool(bool)`
- `TVP newQuote(unsigned int)`
- *etc.*

All the operations defined by the VDM language manual on basic types are implemented one-to-one. They can be found in the native library header file `VdmBasicTypes.h`.

Members of structured VDM types, such as `seq` and `set`, are stored as references, owing to their variable size. The `ptr` field is dedicated to these. These collections are represented as arrays of `TypedValue` elements, wrapped in the C structure `Collection`. The field `size` of `Collection` records the number of elements in the collection, whereas the field `buf_size` records the length of the pre-allocated buffer used for storage. Naturally, collections can be nested. At the level of VDM these data types are immutable and follow value semantics. But internally they are constructed in various ways. For instance, internally creating a fresh set from known values is different from constructing one value-by-value according to some filter on values. In the former case a new set is created in one shot, whereas in the latter an empty set is created to which values are added. Several functions are provided for constructing collections which accommodate these different situations.

- `newSetVar(size_t, ...)`
- `newSetWithValues(size_t, TVP*)`
- `newSeqWithValues(size_t, TVP*)`
- *etc.*

These rely on two functions for constructing the inner collections of type `struct Collection` at field `ptr`:

- `TVP newCollection(size_t, vdmtype)`
- `TVP newCollectionWithValues(size_t, vdmtype, TVP*)`

The former creates an empty collection that can be grown as needed by memory re-allocation. The latter wraps an array of values for inclusion in a `TVP` value of structured type. All the operations defined in the VDM language manual on structured types are implemented one-to-one. They can be found in the header files `VdmSet.h`, `VdmSeq.h` and `VdmMap.h`.

VDM's object orientation features are fundamentally implemented in the native library using C `structs`. In brief, a class is represented by a `struct` whose fields represent the fields of the class. The functions and operations of the class are implemented as functions associated with the corresponding `struct`.

Consider the following example VDM specification.

Listing 3: Example VDM model.

```
class A
instance variables
  private i : int := 1;
```

```
operations
  public op : () ==> int
  op() ==
    return i;
end A
```

The code generator produces the two files `A.h` and `A.c`, shown below.

Listing 4: Corresponding header file `A.h`.

```
#include "Vdm.h"
#include "A.h"

#define CLASS_ID_A_ID 0

#define ACLASS struct A*

#define CLASS_A__Z2opEV 0

struct A
{
  VDM_CLASS_BASE_DEFINITIONS(A);

  VDM_CLASS_FIELD_DEFINITION(A, i);

};

TVP _Z1AEV(ACLASS this_);

ACLASS A_Constructor(ACLASS);
```

The basic construct is a `struct` containing the class fields and the class virtual function table:

Listing 5: Macro for defining class virtual function tables.

```
#define VDM_CLASS_FIELD_DEFINITION(className, name) \
  TVP m_##className##_##name

#define VDM_CLASS_BASE_DEFINITIONS(className) \
  struct VTable * _##className##_pVTable; \
  int _##className##_id; \
  unsigned int _##className##_refs
```

The virtual function table contains information necessary for resolving a function call in a multiple inheritance context as well as a field which receives a pointer to the implementation of the operation `op`.

Listing 6: Virtual function table.

```
typedef TVP (*VirtualFunctionPointer)(void * self, ...);

struct VTable
{
  //Fields used in the case of multiple inheritance.
```

```
  int d;
  int i;

  VirtualFunctionPointer pFunc;
};
```

The rest of the important parts of the implementation consist of the function implementing `op()`, the definition of the virtual function table into which this slots and the complete constructor mechanism.

Listing 7: Corresponding implementation file `A.c`.

```
#include "A.h"
#include <stdio.h>
#include <string.h>

void A_free_fields(struct A *this)
{
  vdmFree(this->m_A_i);
}

static void A_free(struct A *this)
{
  --this->_A_refs;
  if (this->_A_refs < 1)
  {
    A_free_fields(this);
    free(this);
  }
}

/* A.vdmrt 6:9 */
static  TVP _Z2opEV(ACLASS this)
{
  /* A.vdmrt 8:10 */
  TVP ret_1 = vdmClone(newBool(true));

  /* A.vdmrt 8:3 */
  return ret_1;
}


static struct VTable VTableArrayForA [] =
{
  {0,0,((VirtualFunctionPointer) _Z2opEV),},
};


ACLASS A_Constructor(ACLASS this_ptr)
{
  if(this_ptr==NULL)
  {
    this_ptr = (ACLASS) malloc(sizeof(struct A));
  }

  if(this_ptr!=NULL)
  {
    this_ptr->_A_id = CLASS_ID_A_ID;
    this_ptr->_A_refs = 0;
    this_ptr->_A_pVTable=VTableArrayForA;
```

```
    this_ptr->m_A_i= NULL ;
  }

  return this_ptr;
}

// Method for creating new "class"
static TVP new()
{
  ACLASS ptr=A_Constructor(NULL);

  return newTypeValue(VDM_CLASS,
    (TypedValueType)
      {.ptr=newClassValue(ptr->_A_id,
        &ptr->_A_refs,
        (freeVdmClassFunction)&A_free,
        ptr)});
}


/* A.vdmrt 1:7 */
TVP _Z1AEV(ACLASS this)
{
  TVP __buf = NULL;

  if(this == NULL)
  {
    __buf = new();

    this = TO_CLASS_PTR(__buf, A);
  }

  return __buf;
}
```

`TO_CLASS_PTR` merely unwraps values and can be ignored for now.

Construction of an instance of class `A` starts with a call to `_Z1AEV`. An instance of `struct A` is allocated and its virtual function table is populated with the pointer to the implementation of `op()`, `_Z2opEV`. The latter name is a result of a name mangling scheme implemented in order to avoid name clashes in the presence of inheritance[10]. A header file called `MangledNames.h` provides the mappings between VDM model identifiers and mangled names in the generated code. This mapping aids in writing the `main` function. The scheme used is `ClassName_identifier`. Listing 8 shows the contents of the file for the example model.

Listing 8: File `MangledNames.h`.

```
#define A_op _Z2opEV
```

[10]The name mangling scheme is based on the following sources:
`https://en.wikipedia.org/wiki/Name_mangling`, accessed 2016-09-28.
`http://www.avabodh.com/cxxin/namemangling.html`, accessed 2016-09-28.

```
#define A_A _Z1AEV
```

By default, the code generation process provides an empty `main.c` file such that it is possible to compile the generated code initially. It will, of course, be completely inert. The following example populated `main.c` file illustrates how to make use of the generated code.

Listing 9: Example `main.c` file.

```
#include "A.h"

int main()
{
  TVP a_instance = _Z1AEV(NULL);
  TVP result;

  result = CALL_FUNC(A, A, a_instance, CLASS_A__Z2opEV);

  printf("Operation op returns: %d\n", result->value.intVal);

  vdmFree(result);
  vdmFree(a_instance);

  return 0;
}
```

Had the class `A` contained any `values` or `static` fields, the very first calls into the model would have been to `A_const_init()` and `A_static_init()`. The `main.c` file also contains helper functions that aggregate all these calls into corresponding global initialization and tear-down functions. As this is not the case here, an instance of the class implementation is first created, together with a variable to store the result of `op`. The macro `CALL_FUNC` carries out the necessary calculations for calling the correct version of `_Z2opEV` in the presence of inheritance and overriding (not the case here).

Listing 10: Macros supporting function calls.

```
#define GET_STRUCT_FIELD(tname,ptr,fieldtype,fieldname) \
  (*((fieldtype*)(((unsigned char*)ptr) + \
      offsetof(struct tname, fieldname))))


#define GET_VTABLE_FUNC(thisTypeName,funcTname,ptr,id) \
  GET_STRUCT_FIELD(thisTypeName,ptr,struct VTable*, \
      _##funcTname##_pVTable)[id].pFunc


#define CALL_FUNC(thisTypeName,funcTname,classValue,id, args... ) \
  GET_VTABLE_FUNC( thisTypeName, \
              funcTname, \
              TO_CLASS_PTR(classValue,thisTypeName), \
              id) \
  (CLASS_CAST(TO_CLASS_PTR(classValue,thisTypeName), \
    thisTypeName, \
```

```
    funcTname), ## args)
```

The result is assigned to `result`, which is then accessed according to the structure of `TVP`. The function `vdmFree` is the main memory cleanup function for variables of type `TVP`.

5.3.2 Translating Features of VDM-SL

In this section we discuss how the basic features of VDM-RT, those contained in the subset VDM-SL, are translated to C.

Basic data types Instances of the fundamental data types of VDM-SL (integers, reals, characters *etc.*) translate directly to instances of type `TVP` with the appropriate field of the union structure `TypedValueType` set to the value of the instance. They are instantiated using the corresponding constructor functions `newInt()`, `newBool()` *etc.* introduced above. Operations on fundamental data types preserve value semantics by always allocating new memory for the result `TVP` instance and returning the corresponding pointer.

Structured types. Like basic types, aggregate types such as sets and maps are treated in exactly the same way. The support library provides both the data type infrastructure as well as the operations on aggregate types such that translation is rendered straightforward. For example, the definition

```
a : set of int := {1} union {2};
```

translates directly to

```
TVP a = vdmSetUnion(newSetVar(1, newInt(1)), newSetVar(1, newInt(2)));
```

where `newSetVar()` is one of the several special-purpose internal constructors. The translation strategy is similar for sequences and maps. Value semantics for these immutable data types is maintained in the same way as for the basic data types.

Quote types Quote types such as that shown in Listing 11 are treated at the individual element level. Each element is assigned a unique number via a `#define` directive, as shown in Listing 12.

Listing 11: Quote type example.

```
class QuoteExample

types

  public QuoteType = <Val1> | <Val2> | <Val3>

end QuoteExample
```

Listing 12: Quote type example translation.

```
...

#ifndef QUOTE_VAL1
#define QUOTE_VAL1 2658640
#endif /* QUOTE_VAL1 */

#ifndef QUOTE_VAL2
#define QUOTE_VAL2 2658641
#endif /* QUOTE_VAL2 */

...
```

Union types. The decision to keep run-time type information for every variable of type `TVP` obviates the need for a translation strategy for union types.

5.3.3 Translating Features of VDM++

Classes Earlier we introduced the mechanism of C structures used to represent classes. Translation of a model class is therefore straightforward, with each class receiving its own specific `struct`. As illustrated in Listings 4 and 7 above, each class receives its own pair of C header and implementation files. Most importantly, the header file contains the definition of the corresponding class `struct` and the declarations of the interface functions for this struct. These include the top-level constructor and initialization and cleanup functions for class `values` and `static` field declarations. The implementation (`.c`) file contains the constructor mechanism, the definition of the virtual function table of the class and the implementations of the class's functions and operations. The virtual function table is constructed in accordance with the inheritance hierarchy in which the class belongs (this is discussed below). Class `values` definitions and static fields are implemented as global variables. Their definitions are also inserted in the implementation file, along with initializer and cleanup functions to be called, respectively, when the implementation starts and terminates.

Inheritance The effect of inheritance is to augment the definition of the inheriting class with the features of the parent class, *modulo* overriding. In our `struct`-based implementation of classes and objects, the traits of the base class are copied into the `struct` corresponding to the inheriting class. Therefore, the `struct` of the inheriting class duplicates the fields and virtual function table of the base class. It is important to note here that the meaning of qualifiers such as `protected` is lost when such inheritance hierarchies are translated. However, the generated code is meant to be used as a black box, and access to these definitions should not circumvent the existing infrastructure put in place in the original model (*e.g.* accessing a private field manually rather than through the accessor operations defined in the model. Correct access is ensured by Overture.

Consider the translation of the model with inheritance shown in Listing 13. Despite its cumbersome length, we provide the listing of the complete translation so that the reader may also gain familiarity (at his/her own pace) with all the elements of the generated code.

Listing 13: Inheritance example.

```
class A
instance variables
public field_A : int := 0;

operations
public opA : int ==> int
opA(i) == return i;
end A


class B is subclass of A
operations
public opB : () ==> ()
opB() == skip;
end B


class C
instance variables
b : B := new B();

operations
public op : () ==> int
op() == return b.opA(b.field_A);
end C
```

The six files `A.h`, `A.c`, `B.h`, `B.c`, `C.h` and `C.c` reproduced below make up the complete translation.

Listing 14: File `A.h`.

```
// The template for class header
```

```
#ifndef CLASSES_A_H_
#define CLASSES_A_H_

#define VDM_CG

#include "Vdm.h"

//include types used in the class
#include "A.h"

#define CLASS_ID_A_ID 0
#define ACLASS struct A*

#define CLASS_A__Z3opAEI 0

struct A
{
  VDM_CLASS_BASE_DEFINITIONS(A);

  VDM_CLASS_FIELD_DEFINITION(A, field_A);
  VDM_CLASS_FIELD_DEFINITION(A, numFields);
};

TVP _Z1AEV(ACLASS this_);

void A_const_init();
void A_const_shutdown();
void A_static_init();
void A_static_shutdown();

void A_free_fields(ACLASS);
ACLASS A_Constructor(ACLASS);

#endif /* CLASSES_A_H_ */
```

Listing 15: File `A.c`.

```
#include "A.h"
#include <stdio.h>
#include <string.h>


void A_free_fields(struct A *this)
{
  vdmFree(this->m_A_field_A);
}

static void A_free(struct A *this)
{
  --this->_A_refs;
  if (this->_A_refs < 1)
  {
    A_free_fields(this);
    free(this);
  }
}

static  TVP _Z17fieldInitializer2EV(){
  TVP ret_1 = vdmClone(newInt(0));

  return ret_1;
```

```
}

static  TVP _Z17fieldInitializer1EV(){
  TVP ret_2 = vdmClone(newInt(1));

  return ret_2;
}

static  TVP _Z3opAEI(ACLASS this, TVP i){
  TVP ret_3 = vdmClone(i);

  return ret_3;
}

void A_const_init(){
  numFields_1 = _Z17fieldInitializer1EV();

  return ;
}

void A_const_shutdown(){
  vdmFree(numFields_1);

  return ;
}

void A_static_init(){

  return ;
}

void A_static_shutdown(){

  return ;
}

static  struct VTable VTableArrayForA  [] ={

    {0,0,((VirtualFunctionPointer) _Z3opAEI),},
}  ;

ACLASS A_Constructor(ACLASS this_ptr)
{
  if(this_ptr==NULL)
  {
    this_ptr = (ACLASS) malloc(sizeof(struct A));
  }

  if(this_ptr!=NULL)
  {
    this_ptr->_A_id = CLASS_ID_A_ID;
    this_ptr->_A_refs = 0;
    this_ptr->_A_pVTable=VTableArrayForA;

    this_ptr->m_A_field_A= _Z17fieldInitializer2EV();
  }

  return this_ptr;
}

static TVP new(){
  ACLASS ptr=A_Constructor(NULL);
```

```
  return newTypeValue(VDM_CLASS, (TypedValueType)
      { .ptr=newClassValue(ptr->_A_id, &ptr->_A_refs, \
        (freeVdmClassFunction)&A_free, ptr)});
}

TVP _Z1AEV(ACLASS this){
  TVP __buf = NULL;

  if ( this == NULL )
  {
    __buf = new();

    this = TO_CLASS_PTR(__buf, A);
  }

  return __buf;
}

TVP numFields_1 = NULL ;
```

Listing 16: `File B.h.`

```
#ifndef CLASSES_B_H_
#define CLASSES_B_H_

#define VDM_CG

#include "Vdm.h"
#include "A.h"
#include "B.h"

#define CLASS_ID_B_ID 1

#define BCLASS struct B*

#define CLASS_B__Z3opBEV 0

struct B
{
  VDM_CLASS_BASE_DEFINITIONS(A);

  VDM_CLASS_FIELD_DEFINITION(A, field_A);
  VDM_CLASS_FIELD_DEFINITION(A, numFields);

  VDM_CLASS_BASE_DEFINITIONS(B);

  VDM_CLASS_FIELD_DEFINITION(B, numFields);
};

TVP _Z1BEV(BCLASS this_);

void B_const_init();
void B_const_shutdown();
void B_static_init();
void B_static_shutdown();

void B_free_fields(BCLASS);
BCLASS B_Constructor(BCLASS);

#endif /* CLASSES_B_H_ */
```

Listing 17: File `B.c`.

```
#include "B.h"
#include <stdio.h>
#include <string.h>

void B_free_fields(struct B *this)
{
}

static void B_free(struct B *this)
{
  --this->_B_refs;
  if (this->_B_refs < 1)
  {
    B_free_fields(this);
    free(this);
  }
}

static  void _Z3opBEV(BCLASS this){
  {
    //Skip
  };
}

void B_const_init(){
  return ;
}

void B_const_shutdown(){
  return ;
}

void B_static_init(){
  return ;
}

void B_static_shutdown(){
  return ;
}

static  struct VTable VTableArrayForB  [] ={

    {0,0,((VirtualFunctionPointer) _Z3opBEV),},
}  ;

BCLASS B_Constructor(BCLASS this_ptr)
{
  if(this_ptr==NULL)
  {
    this_ptr = (BCLASS) malloc(sizeof(struct B));
  }

  if(this_ptr!=NULL)
  {
    A_Constructor((ACLASS)CLASS_CAST(this_ptr,B,A));

    this_ptr->_B_id = CLASS_ID_B_ID;
    this_ptr->_B_refs = 0;
    this_ptr->_B_pVTable=VTableArrayForB;
  }
```

```
  return this_ptr;
}

static TVP new(){
  BCLASS ptr=B_Constructor(NULL);

  return newTypeValue(VDM_CLASS, (TypedValueType)
      { .ptr=newClassValue(ptr->_B_id, &ptr->_B_refs, \
          (freeVdmClassFunction)&B_free, ptr)});
}

TVP _Z1BEV(BCLASS this){
  TVP __buf = NULL;

  if ( this == NULL )
  {
    __buf = new();
    this = TO_CLASS_PTR(__buf, B);
  }

  _Z1AEV(((ACLASS) CLASS_CAST(this, B, A)));

  return __buf;
}
```

Listing 18: File `C.h`.

```
#ifndef CLASSES_C_H_
#define CLASSES_C_H_

#define VDM_CG

#include "Vdm.h"
#include "B.h"
#include "C.h"

#define CLASS_ID_C_ID 2

#define CCLASS struct C*

#define CLASS_C__Z2opEV 0

struct C
{
  VDM_CLASS_BASE_DEFINITIONS(C);

  VDM_CLASS_FIELD_DEFINITION(C,b);
  VDM_CLASS_FIELD_DEFINITION(C,numFields);
};

TVP _Z1CEV(CCLASS this_);

void C_const_init();
void C_const_shutdown();
void C_static_init();
void C_static_shutdown();

void C_free_fields(CCLASS);
CCLASS C_Constructor(CCLASS);
```

```
#endif /* CLASSES_C_H_ */
```

Listing 19: File `C.c`.

```
#include "C.h"
#include <stdio.h>
#include <string.h>

void C_free_fields(struct C *this)
{
  vdmFree(this->m_C_b);
}

static void C_free(struct C *this)
{
  --this->_C_refs;
  if (this->_C_refs < 1)
  {
    C_free_fields(this);
    free(this);
  }
}

static  TVP _Z17fieldInitializer4EV(){
  TVP ret_4 = vdmClone(_Z1BEV(NULL));

  return ret_4;
}

static  TVP _Z17fieldInitializer3EV(){
  TVP ret_5 = vdmClone(newInt(1));

  return ret_5;
}

static  TVP _Z2opEV(CCLASS this){
  TVP embeding_1 = GET_FIELD(A, A, GET_FIELD_PTR(C, C, this, b), field_A);

  TVP ret_6 = vdmClone(CALL_FUNC(B, A, GET_FIELD_PTR(C, C, this, b), \
      CLASS_A__Z3opAEI, embeding_1));

  return ret_6;
}

void C_const_init(){
  numFields_2 = _Z17fieldInitializer3EV();

  return ;
}

void C_const_shutdown(){
  vdmFree(numFields_2);

  return ;
}

void C_static_init(){
  return ;
}

void C_static_shutdown(){
```

```
  return ;
}

static  struct VTable VTableArrayForC  [] ={

    {0,0,((VirtualFunctionPointer) _Z2opEV),},
}  ;

CCLASS C_Constructor(CCLASS this_ptr)
{
  if(this_ptr==NULL)
  {
    this_ptr = (CCLASS) malloc(sizeof(struct C));
  }

  if(this_ptr!=NULL)
  {
    this_ptr->_C_id = CLASS_ID_C_ID;
    this_ptr->_C_refs = 0;
    this_ptr->_C_pVTable=VTableArrayForC;

    this_ptr->m_C_b= _Z17fieldInitializer4EV();
  }

  return this_ptr;
}

static TVP new(){
  CCLASS ptr=C_Constructor(NULL);

  return newTypeValue(VDM_CLASS, (TypedValueType)
      { .ptr=newClassValue(ptr->_C_id, &ptr->_C_refs, \
          (freeVdmClassFunction)&C_free, ptr)});
}

TVP _Z1CEV(CCLASS this){
  TVP __buf = NULL;

  if ( this == NULL )
  {
    __buf = new();

    this = TO_CLASS_PTR(__buf, C);
  }

  return __buf;
}

TVP numFields_2 =  NULL ;
```

The duplication of the elements of `A` can be seen in the definition of `struct B` in `B.h`. The listing for `C.c` illustrates the mechanism by which a call to an inherited operation on an instance of `B` is achieved. The macro `CALL_FUNC` is the primary function and method call mechanism. It uses information about the type of the object on which the operation is invoked, as well as the class in which the operation is actually defined, to calculate a function pointer offset in the correct (duplicated) virtual function table of the instance of `B`. The

class in which the operation is originally defined (`A` in this case) is calculated by scanning the chain of superclasses of `B` and choosing the nearest definition. This method satisfies semantics of calls of inherited operations.

Polymorphism Overture limits polymorphism, the overloading of operations and functions. Overloaded operations and functions can only be distinguished by Overture's type system only if they differ in their parameter types. Operations differing only in return type can not be distinguished, rendering the following example definition illegal:

```
class Overloading

operations

public op : bool ==> ()
op(a) == skip;

public op : bool ==> bool
op(a) == return true;

end Overloading
```

Polymorphism is implemented by way of a name mangling scheme, whereby the name generated for any operation or function is augmented with tags representing its parameter types. For instance, the name of the following operation

```
public theOperation : int * bool * char ==> real
```

is generated as `_Z12theOperationEIBC`. The mangled name can be decomposed as follows:

- `_Z`: prepended to all mangled names.
- `12`: number of characters in the original name.
- `theOperation`: the original name.
- `E`: separator between name and parameter type tags.
- `I`: `int` parameter.
- `B`: `bool` parameter.
- `C`: `char` parameter.

Function and operation overriding In single inheritance scenarios, operation/function overriding is achieved in a simple way by choosing the overriding implementation closest in the inheritance chain to the class to which the object on which the operation is invoked belongs. This is in accordance with the corresponding semantics. In multiple inheritance scenarios, Overture does not allow ambiguity leading to a choice of implementation. For instance, the following model is illegal in Overture:

```
class A
operations
public op : () ==> bool
op() ==
  return true
end A

class B
operations
public op : () ==> bool
op() ==
  return false
end B

class C is subclass of B, A
end C
```

This forces the model developer to eliminate all such ambiguity, reducing the scenario to that of single inheritance.

5.3.4 Memory Management

The code generation platform on which the C code generator is based was originally designed to target languages with implicit memory management, such as Java. In the context of C, this poses great difficulty in explicitly freeing allocated memory. For example, the VDM expression

```
1 + 2 + 3
```

translates to the following, independent of context:

```
vdmSum(newInt(1), vdmSum(newInt(2), newInt(3)))
```

Because none of the intermediate values are assigned to `TVP` variables, none of the memory allocated here can be accessed and freed once the outer invocation of `vdmSum()` terminates. This is only a simple illustrative example of the difficulty in dealing with allocated memory explicitly.

We solve this problem using a bespoke garbage collection (GC) strategy that is meant to obviate the need for explicit calls to `vdmFree()` anywhere in

the generated code. All functions that allocate memory on the heap have corresponding GC-aware versions, such that intermediate values allocated as in the example above can be reclaimed in bulk with a call to the GC when it is known to be safe to do so.

The garbage collector is kept simple by the specific structure of models in INTO-CPS. Due to the FMI approach of stepping simulations, an FMI step corresponds, in the VDM world, to one execution of a periodic task. It is known that all variables that are allocated during one execution either update class fields, which are not subject to garbage collection by design, or are otherwise intermediate. In this pattern of execution it is natural to invoke the garbage collector each time the periodic task has finished executing.

The time and memory performance of this prototype garbage collection strategy has been summarily assessed using the VDM model shown in Listing 20.

Listing 20: Collatz conjecture model.

```
class Collatz
instance variables
val : int;

operations
public Collatz : int ==> Collatz
Collatz(v) ==
  val := v;

public run : () ==> ()
run() ==
  if val = 1 then
    return
  elseif val mod 2 = 0 then
    val := val div 2
  else
    val := 3 * val + 1;
end Collatz
```

The Collatz conjecture states that the sequence of natural numbers calculated above always ends in 1, for any starting natural number greater than 1[11]. The model is designed such that the starting number is fixed when the class is instantiated, and each time the method `run()` is invoked it calculates the next number in the Collatz sequence, if the sequence has not yet converged.

The assessment compares the memory performance of the code generator on this model without memory management, its performance using garbage collection, and its performance in the ideal case where all allocated memory

[11] `http://https://en.wikipedia.org/wiki/Collatz_conjecture`.

is freed explicitly. The results are summarized in Figure 4 for initial value of 77,031, which is known to take 350 steps to converge.

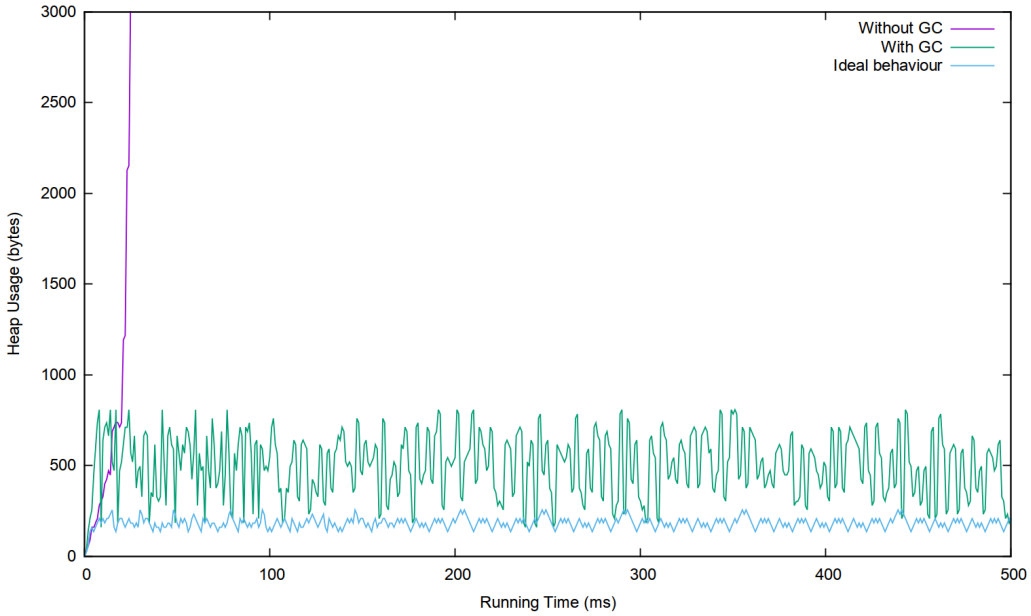

Figure 4: Memory performance of three implementations.

Without any memory management, memory usage increases into the megabyte range by the time the sequence converges. Memory usage in the ideal case tops out at around 300 bytes, whereas with the garbage collection scheme in place it tops out at around 900 bytes. A version of the generated code modified such that its total execution time can be observed (essentially by repeating the procedure thousands of times) yields an increase from 300 ms total execution time in the current and ideal cases, to 310 ms for the implementation with garbage collection.

The garbage collection scheme has been proven to perform adequately for INTO-CPS case studies deployed on embedded microcontrollers, both PIC32 (32-bit) and ATmega (8-bit).

5.3.5 Supported Features

The Overture code generator does not fully support post-condition and invariant checking, lambda expressions, file I/O and full pattern matching.

5.4 Distributed Architectures in VDM-RT

In this section we discuss code generation support for architectures involving distribution modelled in VDM-RT. Section 5.4.1 revisits the principles governing modelling of distributed architectures using VDM-RT. Moreover, it recaps the analysis carried out by the code generator in order to support distributed embedded architectures (DES), as was described in the Year 2 version of this deliverable [BHPG16]. Afterwards, Section 5.4.2 gives concrete guidelines for limiting VDM-RT models as well as motivating them, in order to generate support for distributed embedded systems. Section 5.4.3 provides a minimal example illustrating the main principles for designing distribution into a VDM-RT model. Moreover, we show both the VDM-RT model and highlight the main parts of the distribution-specific C code. Finally, Section 5.4.4 shows how the code generator has been applied in order to realise the INTO-CPS railway interlocking case study from ClearSy (confidential model), which is further described in deliverable D1.3b [OL17]. Additionally, we show various plots that validate the generated C code for the non-public ClearSy model. Support for distributed architectures, discussed below in more detail, is also described in [BTJHL17].

5.4.1 VDM-RT Distribution

In VDM-RT, distribution is modelled inside a special class called a `system` definition. Therefore, distribution is decoupled from the main functionality described by normal classes, and is only introduced by the special `system` definition. This allows easy architectural changes, as well as making a distributed system more holistic. Consequently, the code generator needs to analyse the `system` definition and provide supporting code for distribution based on this analysis. Inside this definition the implicit classes `CPU` and `BUS` are used to model a distributed architecture. The `CPU` class models an independent processor unit or a computation node, and objects can be deployed to them, that is, specified to execute on them. Hence every `CPU` instance corresponds to an individual processor or computation node in the C code implementation of a VDM-RT model. The `BUS` class is used to model a communication channel between `CPU`s. However, in VDM-RT a `BUS` does not specify a hardware bus type, only an abstract representation. So in order to enable the user to interface with a proprietary driver (as is the case with ClearSy's interlocking model, described in Section 5.4.4), the code generator allows the user to provide drivers for a given hardware communication bus, for example a UART (Universal Asynchronous Receiver/Transmitter) or

CAN (Controller Area Network) bus, possibly together with a protocol.

As discussed in Deliverable D5.2c, the code generator automatically generates support for model-level distribution, *i.e.* it provides correct dispatching of remote calls to a concrete user-defined communication protocol/driver (called `send_bus`) as well as providing a dispatcher in order to handle incoming remote calls (called `getRes`). The former enables dispatching with respect to the modelling inside the `system` definition towards a specific bus hardware protocol/driver implementation. The latter enables each CPU to handle possible incoming calls and ensures that the correct method of a local object is invoked. Both functions have standardised interfaces, hence allowing easier interfacing with the low-level aspect of an implementation. Additionally, both functions are generated when the C code for the `system` definition class is generated. Hence the developer can implement the low-level aspects of communication, while the code generator provides the model-level support. During code generation of the architecture modelled in VDM-RT, the relevant information must be extracted from the `system` class definition and used for generation of each computation node. This includes generating each `CPU` as its individual executable with its deployed objects, and generating correct dispatching of remote calls based on the connection between nodes, as specified in the `system` class.

Finally, another aspect is the serialisation and de-serialisation of information to be sent across the network. As described in Deliverable D5.2c, the Abstract Syntax Notation One (ASN.1)[12] notation could be used for serialisation of data types. In particular, research has been carried out as part of `VDM2C` on using the ASN.1 compiler developed by the European Space Agency (ESA), as found at `https://github.com/ttsiodras/asn1scc`. This tool is especially made to target embedded platforms. Subsequently this tool is referred to as `asn1.exe`, and is supported for development under Linux, Mac OSX and Windows platforms. Additionally, work has been carried out to create converters between the `TVP` structure used in our VDM-RT C code generator and the corresponding data types in ASN.1 [FVB+16], referred to as `asn2tvp`[13]. As a consequence a `TVP` value can be converted to an ASN.1 representation, serialized and sent across the network. When received, this data can be de-serialized, and converted back to `TVP` values. In compari-

[12] An overview is available at `https://en.wikipedia.org/wiki/Abstract_Syntax_Notation_One`.

[13] The work for mapping between `TVP` and ASN.1 [FVB+16] is available at `https://github.com/tfabbri/DataModellingTools`, and a script for installation on Linux and Mac OSX can be found at `https://raw.githubusercontent.com/tfabbri/DataModellingTools/vdm-b-mapper/dmt/vdm_tests/INSTALL.sh`.

son, generic message representations such as XML or JSON can be used, but these require generic parsers at run-time, causing memory and CPU overhead. Note that we allow developers to interface with their own serialisation functions, similar to what is allowed for communication drivers, in order to provide additional flexibility when targeting resource-constrained embedded devices. For this reason, serialization can be added depending on need, as an underlying layer, supporting the low-level implementation.

The approach to using our code generator together with the ANS.1 notation and corresponding tools is illustrated in Figure 5, together with activities involved (names indicated on top of arrows) that are further discussed below. Again, note the overall approach: the code generator emits C code based on functionality and model architecture, while the developer is allowed to add proprietary network drives associated with each bus as captured inside the `system` definition. Such an approach not only allows the developer to both use custom drivers, but furthermore to consider non-idealised communication. So while the model is idealised, with this approach the developer can handle the actual low-level aspects of communication. Finally, with the tool `vdm2asn` (contained inside our `VDM2C`), an ASN.1 data file can be obtained. This file can then be used to generate specialised encoders/decoders in C code as well as mapping functions between ASN.1 types in C code and `TVP`s using the tools `asn1.exe` (developed by ESA) and `asn2tvp` (work presented in [FVB+16]). Additionally, note that we provide a distribution run-time (similar to the run-time library of the functional code) based on ASN.1 using the ESA `asn1.exe` tool, which provides serialisation and de-serialisation for basic types, such as integers, reals, booleans and quotes. However, for more specialised types, such as records, a specific ASN.1 file needs to be used to generate the encoding, and then `asn2tvp` can be used to generate the mapping functions. The approach using ASN.1 has been applied for serialisation in both the minimal example as well as the ClearSy case study. With respect to Figure 5, we discuss in more in detail below *initialisation* support of distributed systems, as well as generated *architecture awareness* support.

Initialisation During the initialisation process the resources for each node are created. With respect to architecture, this especially refers to allocation of distributed objects from the `system` class. For this reason the code generator is required to support set up before functional execution for each embedded device. Due to the imposed VDM-RT modelling guidelines, as described below, it can exploit knowledge of deployment in order to establish

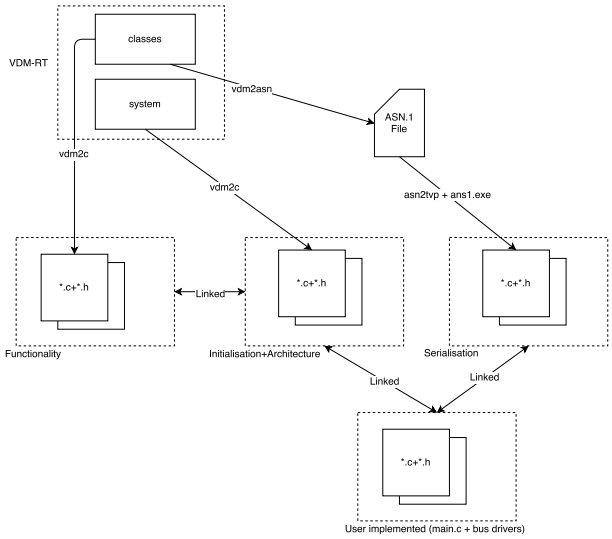

Figure 5: Activities involved for distributed code generation.

awareness of local and remote objects inside the `system` definition. During the code generation process, each distributed object is assigned a unique number ID, ensuring DES-wide identification for all. Then, with respect to each node, local objects are instantiated as normal instances, *e.g.* `TVP` type `VDM_CLASS`, while remote objects become of type `VDM_INT` instantiated with their unique ID. This is generated in the form of an initialisation method that sets up both local and remote distributed objects with respect to a given node. For example, if the node is called `cpu1`, the code generator emits an initialisation method for it as `cpu1_init()`. Furthermore, this method gets the set up invocations of local objects inside the `system` constructor (see Listing 25). Hence, the common `TVP` allows having a dispatch macro to wrap the `CALL_FUNC` macro in order to decide whether to dispatch a call as local or remote, as shown in Listing 21. For this listing, note that a remote object is passed to the generic method called `send_bus`. This method is generated based on the architecture, and is further discussed below.

Listing 21: `DIST_CALL` macro which dispatches between local and remote invocations.

```
#define DIST_CALL(sTy, bTy, obj, supID ,nrArgs ,funID, args...)
((obj->type==VDM_CLASS) ? CALL_FUNC(sTy, bTy, obj, funID, ## args) :
  send_bus(obj->value.intVal, funID, supID, nrArgs, ## args))
```

Overall the presented initialisation approach allows to set up a global reference to an object independently of its location, and additionally without the need to obtain an actual memory reference on the given node during

initialisation. Consequently this exploits both the location and distribution transparency features provided in the VDM-RT model more directly. However, another way to achieve distinction between local and remote objects would be to extend a location transparent RMI technology with an initialisation algorithm which sets up and obtains the correct references. Such an approach has been applied in [HLTJ15] for the VDM-RT to Java code generator using the Java RMI technology [Sun00]. For VDM2C, the CORBA middleware [OMG02] would be a similar solution. Using such an RMI technology requires a more sophisticated initialisation functionality, as it requires exchange of references during system set up. For example in [HLTJ15] where Java RMI is used, an initialisation algorithm is generated under the assumption of a global database for exchange of references. Having an arbitrary network structure, and no such database, makes it even more involved. When targeting embedded systems, in addition, a specific middleware technology would limit the generated code in two ways: first, it might not easily allow use of proprietary bus drivers as this technology relies on its own infrastructure. Second, it would introduce large performance as well as memory overhead, since it is a general-purpose technology that is not particularly suited for embedded devices. For these reasons, and to further gain full control of RMI, support for this feature is implemented as an extension of the VDM2C runtime library.

Architectural Awareness Above we discussed the routing of invocation with respect to object deployment. While local calls are handled as a regular calls, remote calls are routed to the generic `send_bus` method, with the signature shown in Listing 22.

Listing 22: Signature of `send_bus` function.

```
TVP send_bus(int objID, int funID, int supID, int nrArgs, ... )
```

This method needs to forward the remote call to the correct bus in accordance with the VDM-RT system architecture, *e.g.* calling the user-defined driver. In order to support such functionality, the code generator needs to extract relevant information from the `system` definition together with an algorithm to emit architecture support code. Particularly, routing is achieved using the architecture defined by the `CPU` and `BUS` constructs, together with the deployment of objects. Listing 23 illustrates the algorithm to dispatch with respect to the analysis; internally the code generator analyses various relations: set of objects deployed to a node (`cpuToObjs`), set of BUSses connected to each node (`busToCpus`) and the set of nodes connected to each

BUS (`cpuToBusses`). With respect to a computation node, the `generate` method basically emits a switch functionality for each bus that is based on a set of objects that can be reached by the given bus. In this way `send_bus` forwards the remote call to a function with the exact name as in the model together with a standardised parameter list, which enables the user-defined low-level communication driver to be linked.

Listing 23: Pseudocode for algorithm which generates `send_bus` functionality for each CPU from a VDM-RT model.

```
cpuToBusses : Map[cpu -> {set of bus}]
busToCpus   : Map[bus -> {set of cpus}]
cpuToObjs   : Map[cpu -> {set of object}]

for each cpu do
  for each bus in cpuToBusses(cpu)
    for each cpu_con in (busToCpus(bus)\cpu)
      generate(bus, cpuToObjs(cpu_con))
```

Another facet of architectural awareness is the ability to handle an incoming remote invocation, *e.g.* when `send_bus` initiates a call from another CPU. The code generator supports this by emitting the method `getRes`, having a standardised interface as well (Listing 24), which ensures correct dispatching to the local object resource.

Listing 24: Signature of `getRes` function.

```
TVP getRes(int objID, int funID, int supID, int nrArgs, TVP args [])
```

This functionality is provided by using the knowledge of locally deployed objects, *e.g.* `cpuToObjs`, to generate switch functionality based on the unique IDs of distributed objects. Since both `send_bus` and `getRes` are related to the architecture, they are generated as part of the emission of the `system` definition.

Simulation in VDM-RT: Once a VDM-RT model has been constructed, it can for example be validated by means of simulation, where the VDM interpreter [LLB11] makes use of the object deployment inside the `system` definition in order to determine when network communication is required. This supports a holistic approach towards architecture validation and exploration independently from the overall functionality. For example trying to access an object without having a communication channel yields an error by the interpreter, providing early design feedback when modelling a distributed architecture. In order to facilitate a simulation of a model of a System Under Development (SUD), the context in which it appears, *i.e.* its environment,

also needs to be modelled. However, since we are only interested in the precise performance of the SUD, VDM-RT includes a concept of a virtual CPU and a virtual BUS that are both infinitely fast. Consequently, for simulation purposes all objects not deployed to a computing node automatically get placed on the virtual CPU connected to all other computing nodes via the virtual BUS. Finally, since we are targeting an implementation of a DES, it is also worth noting that communication in a VDM-RT context is considered at an abstract level, *i.e.* it is idealised, so no messages are lost.

5.4.2 VDM-RT Modelling Guidelines

A systematic process for gradually introducing complexity towards a VDM-RT model exists [LFW09]. However, this process is not sufficient for automatic generation of code in a distributed context. For this reason we put in place modelling guidelines to constrain the way VDM-RT is used, such that code generation can be achieved. Within this context existing guidelines are consulted, such as MISRA C [MIR04] for embedded systems together with similar literature for general distributed systems principles [CDK05]. In addition, these guidelines extend upon the principles described in previous work [HLTJ15], which involves a code generator supporting distributed Java-based systems using Java RMI as the underlying technology. Consequently, from a development perspective, a more abstract VDM-RT model might gradually be transformed such that it complies with the guidelines, and enables efficient generation of code.

The following modelling guidelines can be divided into two parts, namely architecture *structure* considerations and a notation *subset* definition. The *structure* is solely for the `system` definition, as it attempts to standardise the architecture model. On the other hand, the *subset* is imposed on the architecture definition together with the normal classes of a VDM-RT model, and inspired by combining established practices for both object oriented design in embedded as well as distributed systems.

From the perspective of a typical embedded control system consisting of controllers that use sensors and actuators, Listing 25 presents the *structure* to follow when defining an architecture in VDM-RT (this structure is inspired by the more generic model called System used in the INTO-CPS Year 2 VDM-RT semantics deliverable, D2.2b [FCC+16]). The first part consists of instantiating actuator and sensor objects to which controller objects can obtain a reference as part of their creation. These objects are referred to as distributed objects, which defines objects that might initiate network com-

munication. Next, the system architecture is modelled using the `CPU` and `BUS` constructs. Finally, the constructor of the `system` class contains the deployment of objects to nodes. Inside this constructor we allow invocation of methods on distributed objects, for set-up purposes, as long as these do not invoke other distributed objects, in order to avoid possible network communication during system start-up. This restriction makes the DES initialisation automatable. Moreover, network communication inside an instance might be initiated in two ways: (1) all distributed objects are accessible directly in the entire model (*e.g.* `Sys‘controller.method()`), since they are declared as `public static`; (2) using the object references that are passed during instantiation as shown in Listing 25 for controllers.

Listing 25: Structure for the ideal `system` definition in VDM-RT.

```
system Sys

instance variables
  -- Actuators
  public static act1_1 : Actuator_1 = new Actuator_1(...),
  ...
  public static act1_k1 : Actuator_n = new Actuator_n(...),

  -- Sensors
  public static sen1_l1 : Sensor_1 = new Sensor_1(...),
  ...
  public static sen1_ln : Sensor_n = new Sensor_n(...)

  -- Controllers
  public static ctrl_1 : Ctrl_1 :=
      new Ctrl_1(act1_1,...,act1_k1,sen1_1,...,sen1_l1);
  ...
  public static ctrl_n : Ctrl_n :=
      new Ctrl_n(actn_1,...,actn_kn,senn_1,...,senn_ln);

  -- CPUs
  public static cpu_1 : CPU := new CPU(sp1, s1);
  ...
  public static cpu_n : CPU := new CPU(spn, sn);

  -- BUSses
  public static bus_1 : BUS :=
      new BUS (bp1, b1, cpu_subset_1);
  ...
  public static bus_n : BUS :=
      new BUS (bpn, bn, cpu_subset_n);

operations
  public System : () ==> System
  (
    act1_1.setup(...)
    ...
    ctrl_n.setup(...)

    cpu1.deploy(act1_1)
    ...
    cpun.deploy(ctrl_n);
  );
```

```
end Sys
```

Together with the structure shown in Listing 25, the following modelling guidelines are presented in order to be exploited by the code generator, *e.g.* enable it to automate the generation of code (below we consider our SUD as the part of the VDM-RT model which should be generated and deployed to hardware platforms):

(1) All objects related to the SUD have to be instantiated inside the `system` definition:

This enables the code generator to know the exact location of all distributed objects, which it can exploit during analysis in order to generate architectural support for each node.

(2) All objects related to the SUD must be deployed to user-defined `CPU`s:

All instances created in the `system` must be deployed to explicit nodes as created by the designer. Consequently, no objects containing system functionality may be placed on the virtual CPU, which must only contain instances that are necessary to simulate the SUD as part of model validation using Overture. This supports the use of validation tests during development, which are not generated by the code generator, as they are not part of intended functionality.

(3) Two nodes should only be connected directly by at most one bus:

It is possible to design an architecture where a specific remote object can be reached by multiple network channels from the same device. However, this introduces additional non-determinism in the implementation as the chosen network channel might change for the same remote invocations. In fact, the VDM interpreter ensures determinism in such a scenario by picking the same bus every time. For this reason, this guideline aligns the model interpretation with the implementation. Note that it is conceivable that an object can decide at run-time which of several communication channels (potentially further implemented by different communication technologies) to use for communication, but this is not supported here.

VDM-RT enables introducing certain design characteristics that cannot be realised in an actual implementation of a DES. For this reason some additional modelling guidelines are imposed focusing on both embedded as well as distributed development guidelines for normal (non-`system`) classes:

(1) Globally accessible instance variables (*e.g.* object state) are disallowed:

Such a variable would require a global reference being periodically updated, while also creating potential race conditions.

(2) All data of an object is accessed through access methods:

In principle, the code generator could be updated to support direct field access. However, as we strive to provide general principles that can be used for other technologies as well, this enforces the rule adapted by other distributed technologies, such as CORBA and Java RMI.

(3) All collections must define a size limitation following a prescribed pattern:

For collections in a VDM-RT model the size cannot be decided statically before runtime. However, working with systems that have limited memory constraints as well as network communication, it usually is not desirable to have unknown sizes of collections in the implementation. As a consequence, this rule enforces a range definition for all collections using a specific modelling pattern, using the VDM-RT invariant construct (`inv`). Hence collection types are required to be restricted to a range of values by following an invariant pattern: "`inv card collection <= X`", where `X` is a numerical value, `X > 0`, and `card collection` is the size of the collection. This enables validation of the device having sufficient memory. In addition, this enables efficient data serialisation of data that needs to be communicated between computing nodes.

5.4.3 Example - VDM-RT Distributed Watertank

In this subsection we present a minimal example of distribution in a VDM-RT context. This example is inspired by the INTO-CPS pilot case study involving the water tank controller. However, in this example we present a distributed version in order to illustrate the distribution support, involving a supervisor and two water tank controllers that are under the control of the supervisor.

VDM-RT model Listing 26 shows the architecture that is created inside the `system` definition. Note that the two `WatertankController` objects are passed as arguments in order to be invoked in accordance with the architecture of the VDM-RT model. As described above, modelling of distribution

is supported by two implicit classes inside VDM-RT called `CPU` and `BUS`: `CPU` models an independent processor, and allows object instances to be deployed to it. Deployment in this context means that execution of the implementation of a particular instance is carried out on a given processor. `BUS` captures a communication channel and allows `CPU` instances to exchange information by connecting them. Hence, based on the deployment definition, a call to an object is either local (on the same CPU) or remote (on another CPU).

Listing 26: Example of distribution in VDM-RT.

```
system System
instance variables

  public static wC1 : WatertankController := new WatertankController();
  public static wC2 : WatertankController := new WatertankController();
  public static supervisor : Supervisor := new Supervisor(wC1, wC2);

  cpu1 : CPU := new CPU(<FP>, 22E6);
  cpu2 : CPU := new CPU(<FP>, 22E6);
  bus : BUS := new BUS(<CSMACD>, 72E3,{cpu1,cpu2});

operations

public System : () ==> System
System () ==
(
  cpu1.deploy(wC1);
  cpu1.deploy(supervisor);
  cpu2.deploy(wC2);
);
end System
```

Listings 27 and 28 show the two classes used in Listing 26, `Watertank-Controller` and `Supervisor`, respectively. Listing 27 represents a simple controller for a water tank, reading its level and setting its valve. Listing 28 shows how the objects can be invoked inside the `Supervisor`, which controls the controllers without knowing their location with respect to the architecture inside VDM-RT. In this example, with respect to the object `supervisor`, `wc1` is the local object, while `wc2` is the remote object (based on the deployment arrangement of the objects in Listing 26). Both local and a remote calls are invoked as object method invocations, *e.g.* as Listing 28 illustrates, both the local and remote object methods are invoked in the same way. However, according to the VDM-RT semantics, the remote call will be sent over the specified bus.

Listing 27: Example of controller, which reads the water level and sets the valve.

```
class WatertankController

instance variables
```

```
env : Environment := new Environment();

operations

public step : () ==> ()
step() ==
(
  env.step();
  IO`print(env.readWaterLevel());
  IO`print("\n");
);

public readLevel : () ==> int
readLevel() ==
  return env.readWaterLevel();

public setValve : bool ==> ()
setValve(val) ==
  env.setValve(val);

end WatertankController
```

Listing 28: Example of the supervisor that uses two watertank controllers.

```
class Supervisor

instance variables
wC1 : WatertankController;
wC2 : WatertankController;

operations

public Supervisor : WatertankController * WatertankController ==> Supervisor
Supervisor (w1, w2) == (

wC1 := w1;
wC2 := w2;
);

public step : () ==> ()
step() ==
(
  let level1 = wC1.readLevel(), level2 = wC2.readLevel()
    in
    (
      if(level1<4) then
        wC1.setValve(true);
      if(level1>12) then
        wC1.setValve(false);

      if(level2<2) then
        wC2.setValve(true);
      if(level2>6) then
        wC2.setValve(false);
    );

)

end Supervisor
```

Finally, the overall VDM-RT model can be tested and validated by a simple example as shown in Listing 29, *e.g.* `Run()` inside the `World` class.

Listing 29: Example of the supervisor that uses two watertank controllers

```
class World

operations

-- The entry function, running on CPU1 in example
public Run : () ==> bool
Run () ==
(

for - in [i | i in set {1,...,20}] do
  (
    IO`print("Value from controller 1: \n");
    System`wC1.step();
    IO`print("Value from controller 2: \n");
    System`wC2.step();
    System`supervisor.step();
  );

  return true;

)

end World
```

C code In this part we only introduce the parts which need to be added manually by the user in order to make the code run for each CPU. This example is meant to illustrate the main functionality between invocation involving network communication from a VDM-RT model. Hence one node (CPU1) exemplifies a remote invocation, while the other node (CPU2) illustrates handling of an incoming call. The code generator supports dispatching toward a bus defined in the VDM-RT model, while the HW bus implementation can be changed in the execution platform. The communication flow between the send and receive functions indicates that the receiver CPU is able to handle remote invocations, while also running a local execution. In the VDM-RT semantics a CPU has a scheduler, but schedulers are not supported. For this reason, we propose a design pattern. This design pattern is based on the case study needs, and it is a scheme to implement basic remote handling for a CPU. The design pattern assumes there is exactly one periodic thread running on each CPU. This is illustrated in Figure 6. However, note that, similar to the design pattern described above, an existing scheduler on a given platform could also be used in an implementation, by manually scheduling needed methods (that are generated).

Generally, Figure 7 illustrates the flow between the send and receive functions

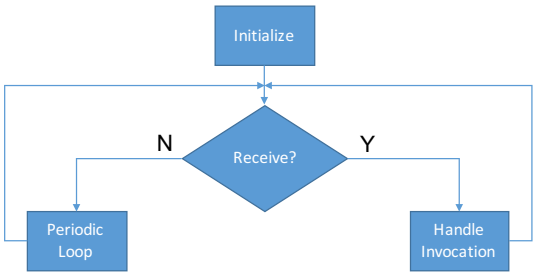

Figure 6: Design pattern enabling a CPU to handle remote invocations while running a periodic thread.

between two CPUs:

1. The send function sends data on a communication channel, which invokes the receive function.
2. The invocation is received, and a possible result is sent back, or an acknowledgment that the function invocation is finished.

This figure also shows that the send function is required to wait for the receive function to handle the request, since calls in VDM-RT are synchronous by default. Asynchronous calls are not supported by the code generator, so the communication is required to be synchronous when implementing the specific call.

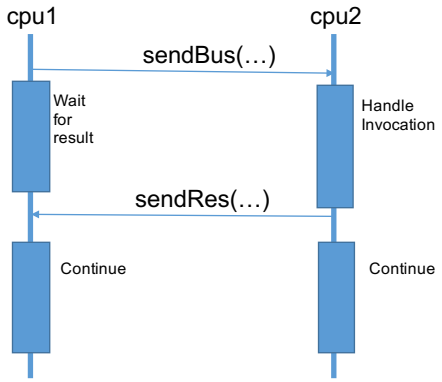

Figure 7: Flow diagram for sending data across a network.

In the following two paragraphs we introduce the manually produced files which enable both CPUs to execute. Each of the two nodes needs to create a hardware-specific implementation of the files `bus.h` and `bus.c` (names following the BUS names from a VDM-RT model (see Listing 26)), as well as create

a corresponding `main` function. In this example we use a simple TCP/IP set up for the bus, in order to illustrate main functionality. In particular in this example we assume that CPU1 will run the validation functionality as defined inside the operation `Run()` shown in Listing 29.

CPU 1: First, the specific bus driver is updated, which means updating the generated files `bus.h` and `bus.c` (name generated based on the `BUS` name from the VDM-RT model shown Listing 26). Listing 30 shows how a TCP/IP based bus is initialised and how values are sent across the network using it. In particular, note that the automatically generated function `send_bus` dispatches the call to the function `bus_send` (the prefix "bus" for the latter function is generated based on the `BUS` name from the VDM-RT model shown Listing 26), which considers the low-level aspects as discussed above. Also note for the implementation of `bus_send` that this allows the developer to construct his own data protocol for a given communication channel. Finally, an example `main.c` file for cpu1 is shown in Listing 31, which is a manually implemented version of the `Run` function, but only invoking the two local objects, *i.e.* `wC1` and `supervisor`.

Listing 30: `Highlighted parts for bus.c for cpu1`

```
// Initialisation of a BUS driver and send functionality

#include "bus.h"

byte buffer[BUF_SIZE];
struct sockaddr_in cli_addr, serv_addr;
socklen_t clilen;
struct hostent *server;
int newsockfd, n, sockfd, portno;

void bus_init(){
  portno = atoi("51717");

  server = gethostbyname("localhost");

  if (server == NULL) {
    fprintf(stderr,"ERROR, no such host\n");
    exit(0);
  }

  bzero((char *) &serv_addr, sizeof(serv_addr));

  serv_addr.sin_family = AF_INET;

  bcopy((char *)server->h_addr,
       (char *)&serv_addr.sin_addr.s_addr,
       server->h_length);

  serv_addr.sin_port = htons(portno);
}
```

```
...

TVP bus_send(int objID, int funID, int supID, int nrArgs, va_list args){

  // 1. Package data
  byte sendArr[BUF_SIZE]; // Array to be send
  byte buf_size = (byte) 4 + 1; // Simple serialization of the known types
  sendArr[0] = buf_size;
  sendArr[1] = (byte) objID;
  sendArr[2] = (byte) funID;
  sendArr[3] = (byte) supID;
  sendArr[4] = (byte) nrArgs;
  serialise(sendArr, nrArgs, args); // Serialise arguments

  // 2. Send data
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");
  if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
    error("ERROR connecting");
  n = write(sockfd,sendArr,BUF_SIZE);
  if (n < 0)
    error("ERROR writing to socket");

  // 3. Wait for result
  byte bufferRes[BUF_SIZE];
  bzero(bufferRes,BUF_SIZE);
  printf("Invoked function with result\n");
  n = read(sockfd,bufferRes,BUF_SIZE);
  if (n < 0)
    error("ERROR reading from socket");
  close(sockfd);

  // 4. Deserialise result and return it
  TVP res = deserialiseRes(bufferRes);
  return res;
}
```

Listing 31: `main.c for cpu1`

```
#include <stdio.h>
#include <stdlib.h>
#include "Vdm.h"
#include "System.h"
#include "World.h"

int main(void) {

  // Overall initialisation
  vdm_gc_init();
  System_static_init();
  bus_init();
  cpu1_init();

  int i;

  for(i=0; i<20; i++){

    DIST_CALL(WatertankController, WatertankController,
      vdmCloneGC(g_System_wC1, NULL), CLASS_ID_WatertankController_ID, 0,
      CLASS_WatertankController__Z4stepEV);
```

```
    DIST_CALL(Supervisor, Supervisor, vdmCloneGC(g_System_supervisor, NULL),
      CLASS_ID_Supervisor_ID, 0, CLASS_Supervisor__Z4stepEV);
  }

  vdm_gc();
  puts("!!!Done!!!");

  return EXIT_SUCCESS;
}
```

CPU 2: In this scenario cpu2 needs to handle a remote invocation, as well as run the `step` of the deployed water tank controller. Listing 32 illustrates especially the receive parts of the hardware driver for the network communication channel. In particular note how the automatically generated function `getRes` can be used in order to obtain a result based on the incoming data. Finally, Listing 33 presents the `main.c` file and illustrates how a remote invocation is handled.

Listing 32: `Highligted parts of bus.c for cpu2`

```
// Low-level hardware specific to handle an incoming call
void handle_bus(){
  // 1. Read incoming data
  bus_recieve(buffer, BUF_SIZE); // BUS specific

  // 2. Deconstruct data
  TVP res = bus_deconstructData(buffer, BUF_SIZE); // General

  // 3. Serialise the result
  byte resBuff[BUF_SIZE];
  serialiseRes(resBuff, res);

  // 4. Send result back
  int n;
  n = write(newsockfd, resBuff, BUF_SIZE);
  close(newsockfd);
  if (n < 0) error("ERROR writing to socket");
}

// Implemented by user, lower-level part
TVP bus_deconstructData(byte* data, int len){
  // 1. User specific protocol data
  int buf_size = (int) data[0];
  int objID = (int) data[1];
  int funID = (int) data[2];
  int supID = (int) data[3];
  int nr_args = (int) data[4];

  // 2. Define arguments
  // Define number of max arguments
  int max_args = 10;
  TVP args[max_args];

  // 3. Deserialise arguments
  deserialise(data, nr_args, args);
```

```
  // 4. Obtain the result from function call
  TVP res = getRes(objID, funID, supID, nr_args, args);
  return res;

}

// Low-level hardware specific
void bus_recieve(byte *buffer, int len){
  // 1. Recieve data, block until accept
  printf("Recieving data \n");
  newsockfd = accept(sockfd,
      (struct sockaddr *) &cli_addr,
      &clilen);
  if (newsockfd < 0)
    error("ERROR on accept");
  bzero(buffer,len);

  // 2. Read incoming data
  int n;
  n = read(newsockfd,buffer,len-1);
  if (n < 0) error("ERROR reading from socket");
}
```

Listing 33: `main.c for cpu2`

```
#include <stdio.h>
#include <stdlib.h>
#include "Vdm.h"
#include "System.h"
#include "World.h"
#include <unistd.h>
#include <pthread.h>
#include "bus.h"

int main(void) {

  // 1. Garbage Collector and Static System class init
  vdm_gc_init();
  System_static_init();

  // 2. BUS init
  bus_init();

  // 3. CPU2 init
  cpu2_init();

  // 4. Run-time
  while(1){
    TVP ret = CALL_FUNC(WatertankController, WatertankController,
      g_System_wC2, CLASS_WatertankController__Z4stepEV);
    handle_bus();
    vdm_gc();
  }

  puts("!!! Done .... CPU 2 !!!!!!");

  return EXIT_SUCCESS;
}
```

Finally, note that each individual CPU gets its own folder with a corresponding CMake file, which can be used to rebuild a new executable when the hardware bus driver has been updated in accordance with the example presented above.

5.4.4 ClearSy - Distributed Railway Interlocking

The code generator has been used to generate the implementation of the INTO-CPS ClearSy case study model. This case study involves distributed interlocking of a train segment, as illustrated in Figure 8. Compared to the minimal example, note that for this case study, every node needs to both run a periodic thread as well as handle a remote invocation. Hence they follow the design pattern as shown in Figure 6. Moreover, note in Figure 8, that the entire track layout has been partitioned into five separate modules: ZV1, ZP, ZV2, ZQ2, ZQ3. Each of these modules corresponds to an individual embedded device, which is responsible for its own equipment, but communicates with the other modules. This case study has been modelled by ClearSy as a VDM-RT model, where the distribution is described in the `system` definition. Finally, hardware has been developed by ClearSy, shown in Figure 9, in order to deploy the code, as well as to enable simulation as illustrated in Figure 10. Moreover, Figure 9 shows six independent modules, one for Ethernet communication (largest board in the middle) when executing test scenarios, and five for the parts of the interlocking connected as a UART ring network. Each board is able to lock three relays for route reservation, manipulate two switches, three track circuits and one pair of signal lights. Hence the set up shown in Figure 9 supports Hardware-in-the-Loop (HiL simulation) in accordance to the principles shown in Figure 10. Hence this HiL set up makes it possible to validate that both the functionality as well as distribution support work as expected when deployed to the hardware.

For the low-level communication, ClearSy's existing proprietary communication drivers are used. Here is an example, as described above for the distributed code generator, of interfacing the generated code with existing hardware-specific drivers. For ClearSy it was necessary to deploy the generated C code to a PIC32MX440F256H micro-controller, as well as use their safety-specific drivers for the UARTs on the micro controller. Hence each of the five zones shown in Figure 9 is running on a PIC32MX440F256H platform, and using PIC32 physical UARTs.

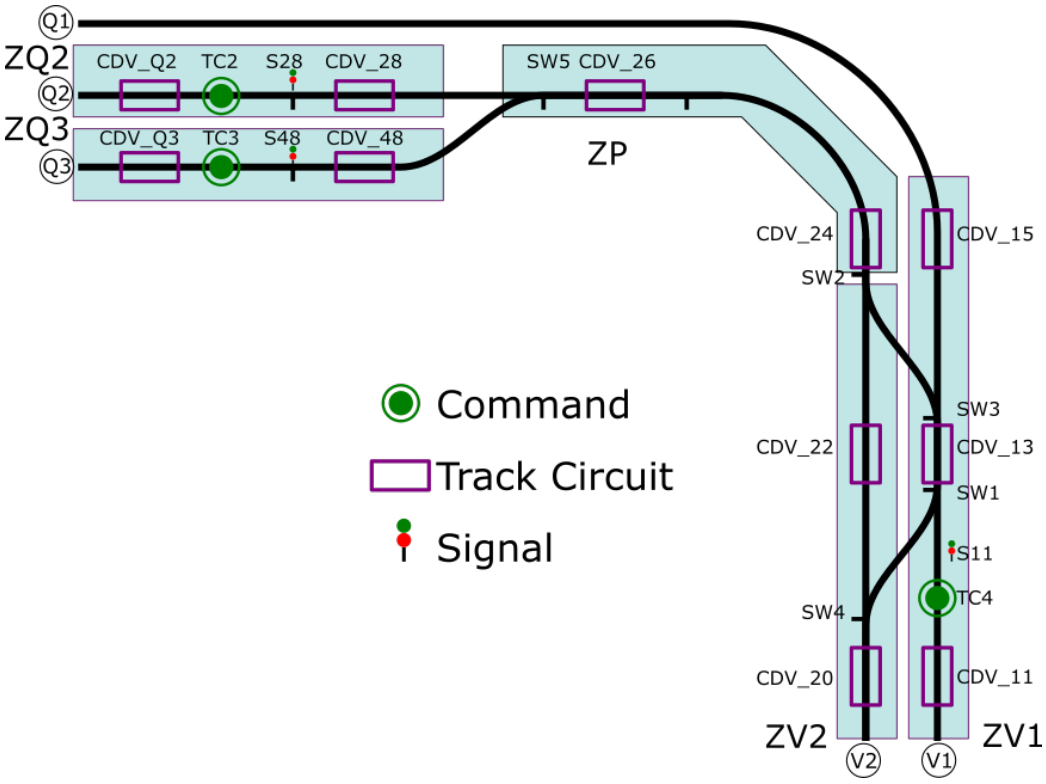

Figure 8: ClearSy distributed interlocking, partitioned into five modules, each responsible for its own equipment, but communicating with the other modules.

Validation by scenarios The ClearSy model can be validated by means of running various scenarios, *i.e.* simulating a train arriving and passing through the entire system. Here we focus on the Q2V2 scenario, which can be described as follows by looking at Figure 8: A train arrives at Q2 (CDV_Q2), the TC2 (remote command) is set to high to signal to establish a route from Q2 to V2. Hence the establishment of this route requires reservation of zones Q2, ZP and ZV2. Once the route has been established, the authorization light on Q2 (S28) turns green. Next, the train moves to the next segment (CDV_28), and the authorization light on Q2 turns red in order to prevent other trains from entering the segment. This scenario is shown below as is illustrates both the functionality as well as distribution support, as the nodes need to communicate in accordance to the VDM-RT model.

As a means of validating the generated code, we show plots of three different instances of the interlocking system, namely Model-in-the-Loop (MiL), Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL). MiL corresponds

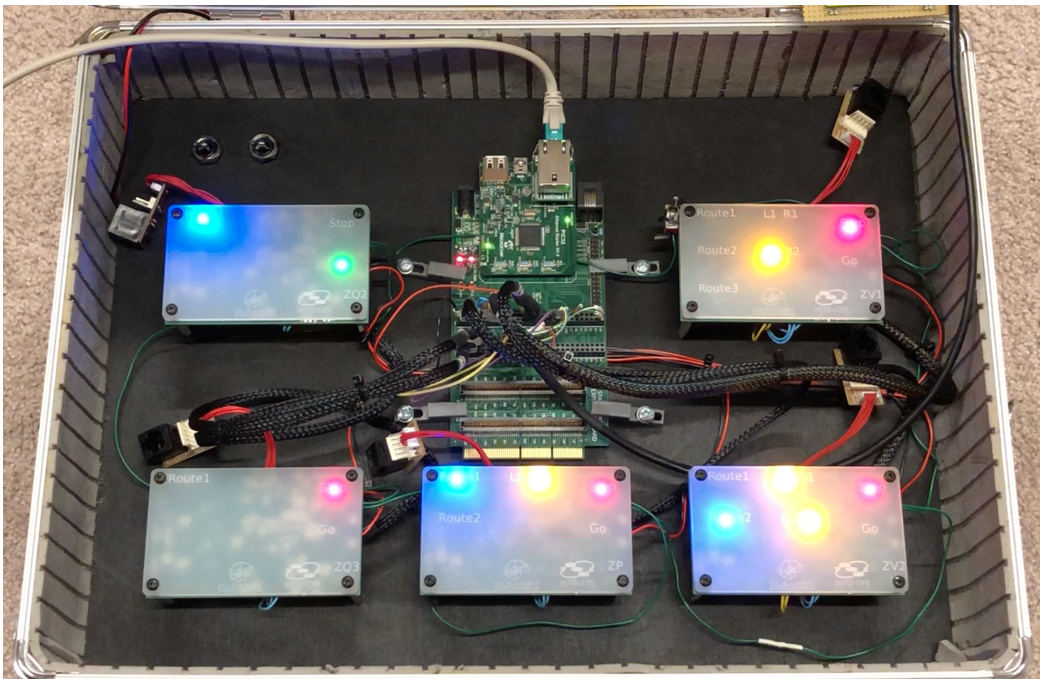

Figure 9: ClearSy HiL platform showing the state of the system when the Q2V2 scenario has been established.

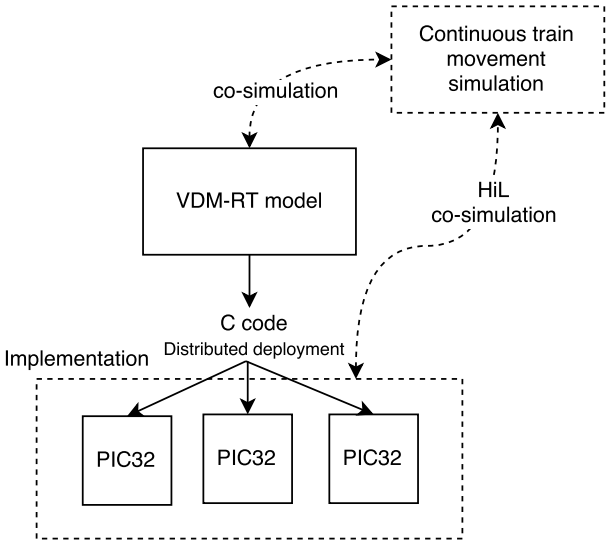

Figure 10: Principles of set up from model simulation and hardware simulation.

to running the VDM-RT model, exported as an FMU, itself as part of the

simulation. SiL corresponds to running the generated C as an FMU in accordance to the FMI simulation semantics. Hence the SiL is just the entire VDM-RT model generated as C code, where each `CPU` has its own thread of execution. Hence showing that the SiL behaves correctly ensures that the generated C code is behaving correctly, before considering deployment and using hardware bus drivers. Finally, the HiL corresponds to running the distribution support of the code generator, and obtaining an executable for each Zone (CPU in the VDM-RT model). Each of these executables can then be deployed to the PIC32MX440F256H controller.

Note that for all the various plots MiL, SiL and HiL we focus on the first 15 seconds of simulation, as it is here that the route and the main functionality of the distributed interlocking occurs. After the 15 seconds the train has passed the authorisation light, which becomes red, and the train moves between the different track segments, without any further changes to the equipment.

First, Figure 11 shows the MiL plots for the relevant variables for this particular scenario Q2V2, and is described as follows. The first sub-plot shows the state of the three telecommands (TC) as well as the current track segment location of the train. Note that the CDV values, as seen in Figure 8, are mapped to a unique value (*e.g.* track segment value 10 means CDV_Q2 and track segment value 8 means CDV_28) is occupied. The second sub-plot displays the relevant switch commands, in this case the two switches on ZV2 and one switch of ZP. As can be seen, only one of the switches of ZV2 sends a command, which means this particular switch has to switch its current position. The third sub-plot shows the current switch position of the train track, and as it can be seen, the second switch of ZV2 changes its position, as would be expected due to the command in the second sub-plot. Finally, the fourth sub-plot displays the status of the three authorization lights. In this case it is important to notice that the correct authorization light turns green (AUT_Q2), and it only goes to green after switching has completed. This functionality needs both to interact with the physical switching relays, as well as to communicate accordingly with the other modules. This functionality represent some of the safety aspects in this system, so even though each module is separate, they are required to communicate in order to safely establish a route.

Second, Figure 12 shows the SiL, where the C code is run as a single FMU application, and each CPU from the VDM-RT model is running in threads in accordance to the FMI co-simulation semantics. As can be seen in Figure 12 the flow between the variables is behaving as the MiL co-simulation in Figure 11. Hence this raises confidence that the generated C code is behaving

correctly, *i.e.* same as the VDM-RT model, before considering deployment to the hardware shown in Figure 9.

Third, Figure 13 shows the execution of the generated code on the hardware shown in Figure 9. Note that in the HiL the physical relays are build, so the code needs to interact and lock these and send the correct messages. The first sub-plot shows similar behaviour as MiL and SiL. For the second sub-plot we cannot obtain the physical command to switch in order to plot it. For this reason the second sub-plot of the command is only illustrative, and based on the third sub-plot where the physical switch position in between states (*i.e.* purple and yellow lines are both 1 in the third sub-plot). Moreover, as can be observed, switch position values are correct compared to the MiL and SiL results. Finally, the fourth sub-plot shows the correct authorization light turning green. However, note that in the HiL co-simulation the actual switching of the ZV2 switch takes longer, and needs afterwards to communicate with ZQ2 in this case. Again, note that the authorization light first turns green once the switch is in the correct position, and turns red when the train passes to the next segment. Note that Figure 9 shows the various lights of the hardware when the Q2V2 scenario has been established by sending telecommand TC2 to module ZQ2: the blue lights on ZQ2, ZP and ZV2 indicate that the route is locked, the orange light indicates the switch position and only the authorisation light on Q2 is green. In order to establish this route, communication messages are sent twice around the UART ring network and are initiated by ZQ2 when the TC2 command is received, first to check that the route can be locked on each relevant module (*i.e.* not occupied), and afterwards, once they are successfully locked, module ZQ2 sends messages to the other two modules to switch the relevant switches. Hence the HiL co-simulation runs in an embedded environment and uses as well the garbage collector developed for embedded devices, so validating its behaviour as well. The HiL co-simulation executed without the garbage collector crashes only after a few executions of the control loop.

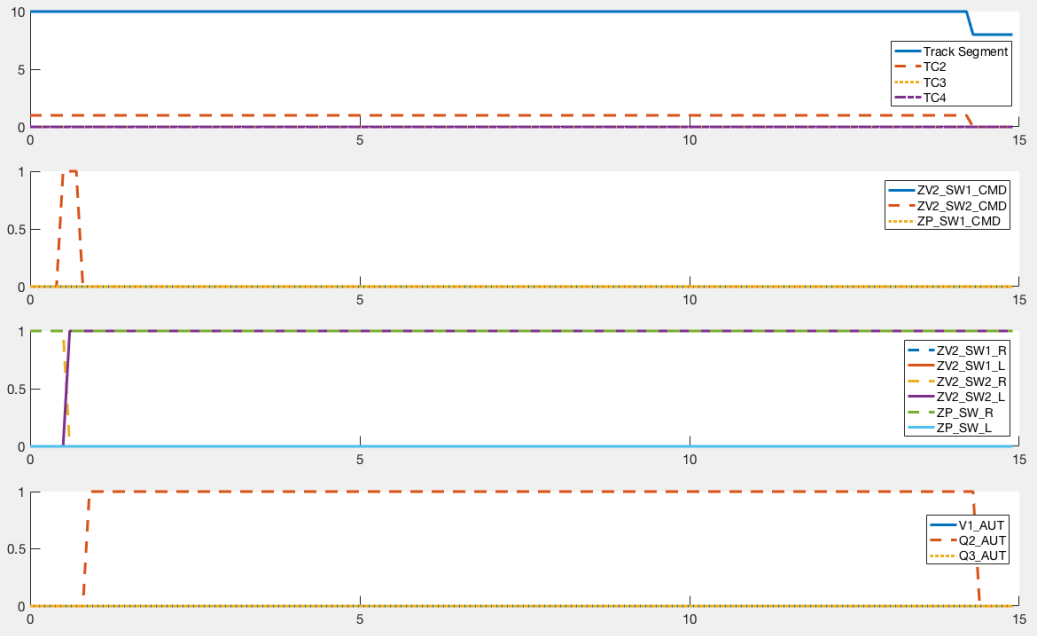

Figure 11: MiL for scenario Q2V2, showing the initial establishment of the Q2V2 route, and train passing the authorization light.

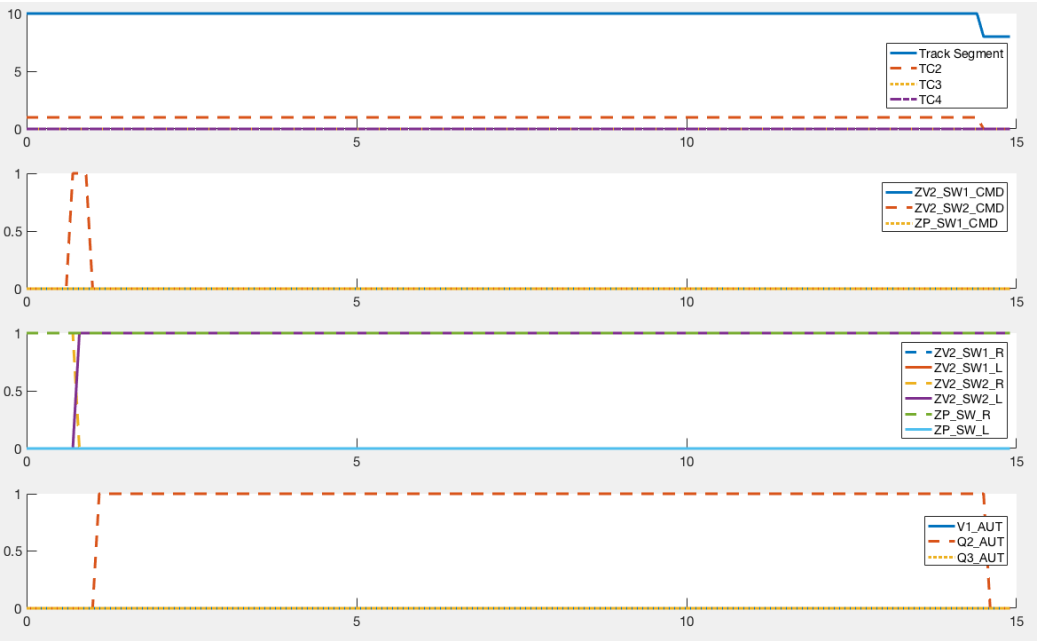

Figure 12: SiL for scenario Q2V2, showing the initial establishment of the Q2V2 route, and train passing the authorization light.

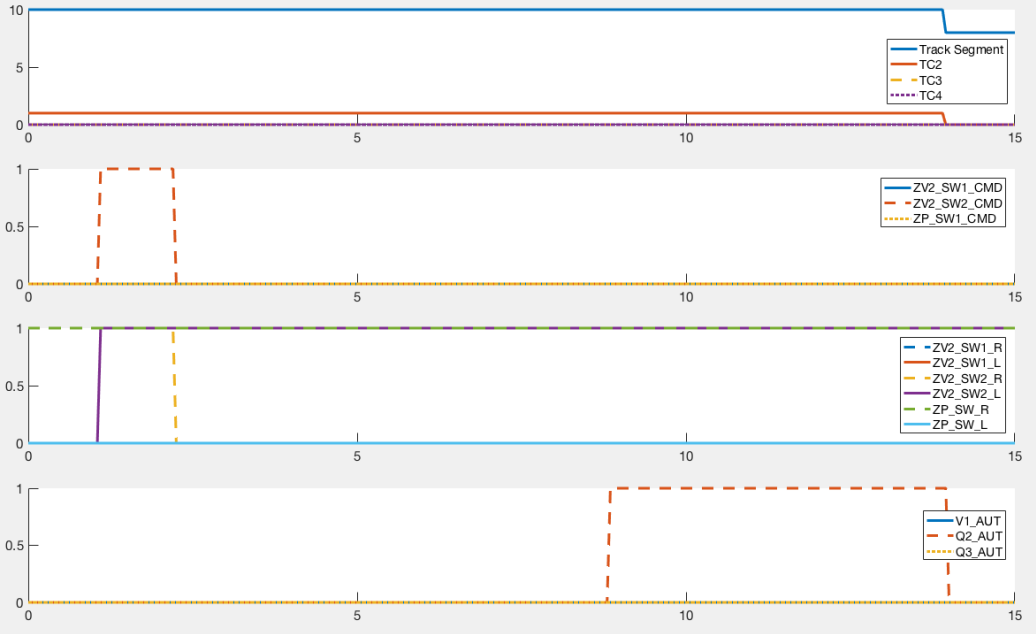

Figure 13: HiL for scenario Q2V2, showing the initial establishment of the Q2V2 route, and train passing the authorization light.

5.5 Ambiguities Not Addressed by the Semantics

Deliverable D2.1b lists a number of ambiguities that semantic foundations for VDM-RT must address. The list is here reproduced. We discuss briefly how each issue is addressed.

1. *Initialization of static instance variables.* Because static variables can be used without creating an instance of the declaring class, it is not sufficient for the generated class constructor to initialize these members. Instead, the code generator emits initialization functions named `ClassName_static_init()`, for static fields, and `ClassName_const_init()` for `value` definitions. These initializers must be called manually in the correct order. The code generator emits helper functions which aggregate these calls in the correct order.
2. *Initialization order of instance variables*: Initialization starts at the leaves of the inheritance hierarchy and proceeds upward.
3. *Calling multiple explicit superclass constructors*: These constructors can be called like any other operation from within subclass constructors.
4. *Multiple inheritance superclass initialization*: Initialization proceeds in the order in which the superclasses are defined in the model.
5. *Implicit calls to default constructors*: In the generated code, default constructors are the only way to create an instance of a class.
6. *Overridden vs. local operations in superclass constructors*: Subject to the same rules as calling overridden operations.
7. *Invariant checking during construction*: Invariants are not supported.
8. *Are constructors inheritable?* Constructors are not inherited.
9. *Overriding/overloading polymorphic/curried functions*: Overriding, overloading and polymorphic functions are supported using name mangling. Curried functions are not supported.
10. *Pre-post conditions in OO state context*: Pre-condition checks are supported. Post-conditions are not supported.
11. *Diamond inheritance*: Overture prevents model ambiguities of this kind.

5.6 Implementation as Overture Plugin

The C code generator is implemented as a standard Eclipse plugin to Overture. It can be obtained through the Eclipse software install system from the following URL:

```
http://overture.au.dk/vdm2c/master/repository
```

Overture facilitates development of code generators targeting standard imperative programming languages through a *code generation platform* [JLC15]. The framework introduces an intermediate representation of VDM-RT abstract syntax which is amenable to relatively straightforward translation to any imperative language. The C code generator generates code for this intermediate representation.

The C code generator is invoked from the context menu in the Project Explorer as shown in Figure 14.

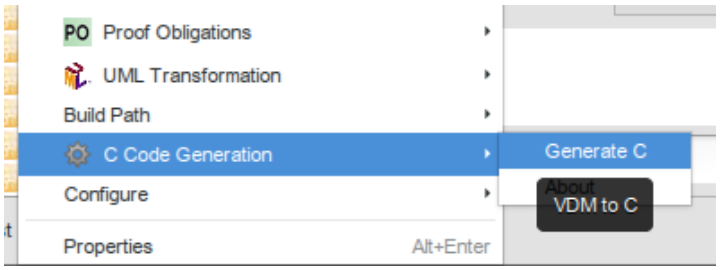

Figure 14: Invoking the code generator.

6 FMU Compilation Service

VDM-RT models can be exported as source code FMUs using Overture's FMU export feature. This feature first invokes the C code generator, then bundles a source code FMU. Aarhus University hosts an FMU cross-compilation server which takes as input a source code FMU and returns a standalone FMU, cross-compiled for 32- and 64-bit Windows and Linux platforms, as well as Mac OSX. The service is available at

```
http://sweng.au.dk/fmubuilder
```

The service is most commonly accessed through the INTO-CPS Application. Both the Overture FMU export feature and the INTO-CPS appli-

cation are documented in the INTO-CPS user manual, Deliverable D4.3a [BLL+17].

7 Conclusions

This deliverable captures the FMI-compliant C and C++ code generation capability of the INTO-CPS tool chain at the end of Year 3 of the INTO-CPS project. The three tools 20-sim, OpenModelica and Overture can now fully integrate their code generation capabilities with the FMI requirements of the INTO-CPS project through export of source code and stand-alone FMUs.

The code generators of OpenModelica and 20-sim are considered mature. The C code generator of Overture, by comparison, is relatively new, so there are a number of avenues to explore in the immediate future. The rest of this section is dedicated to a description of some ideas for future development.

Further development of code generation support for the distribution aspects of VDM-RT can be addressed. Specifically, support for specific bus technologies and related protocols can be added by providing library implementations for these.

A promising avenue for future research is timing behaviour. Recall the distinction made between *descriptive* and *prescriptive* interpretations of RT constructs of VDM-RT. Compared to the descriptive interpretation, assuming the prescriptive view for these constructs is clearly a more ambitious approach from a code generation perspective. We believe that it is possible to implement this interpretation in a code generator. But full support can only be achieved in combination with a real-time operating system that can make guarantees about the timing behaviour of the implementation. It is conceivable that RT constructs can be code generated for specific RTOSs if it is known what types of guarantees the RTOS can make for a given target platform. For instance, if the best and worst case execution times of a code-generated function can be profiled on the target controller, and the RTOS can guarantee an upper bound on execution time given this information, then it can be claimed that the code generator can implement a corresponding `duration` statement placed on this function. The semantic basis for such an approach is already provided in Deliverable D2.2b [FCC+16].

Another avenue for future development is support for run-time post-condition and invariant violation checks. The typical workflow of developing a VDM-

RT model involves using the Overture tool to exercise the specification to a point where the developer is confident that a faithful implementation will not exhibit unanticipated run-time behaviour. The most important VDM-RT features in this endeavour are pre- and post-conditions and invariants. For an extra layer of security, it is possible to allow for hooks in the generated code such that handlers can be called in critical cases of pre- and post-condition and invariant violations. Currently, run-time pre-condition failures cause the program to abort. The actions of the implementation in such circumstances can be specific to the application, and so they are best left to the developer to implement manually. For instance, it may be necessary to reset hardware, close opened files *etc.* An alternative is to provide code-generation support for the exception handling mechanism of VDM-RT and use it to provide the necessary infrastructure where such bespoke error handlers are necessary.

References

[BB97] Breunese and J. F. Broenink. Modeling mechatronic systems using the sidops+ language. In *The Society for Computer Simulation International*, pages 301–306, 1997.

[BHPG16] Victor Bandur, Miran Hasanagic, Adrian Pop, and Marcel Groothuis. FMI-Compliant Code Generation in the INTO-CPS Tool Chain. Technical report, INTO-CPS Deliverable, D5.2c, December 2016.

[Bjø79] D. Bjørner. *The Vienna Development Method: Software Abstraction and Program Synthesis*, volume 75: Math. Studies of Information Processing of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[BLL+17] Victor Bandur, Peter Gorm Larsen, Kenneth Lausdahl, Casper Thule, Anders Franz Terkelsen, Carl Gamble, Adrian Pop, Etienne Brosse, Jörg Brauer, Florian Lapschies, Marcel Groothuis, Christian Kleijn, and Luis Diogo Couto. INTO-CPS Tool Chain User Manual. Technical report, INTO-CPS Deliverable, D4.3a, December 2017.

[Bro90] Jan F. Broenink. *Computer-aided physical-systems modeling and simulation: a bond-graph approach*. PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands, 1990.

[BTJHL17] Victor Bandur, Peter W. V. Tran-Jørgensen, Miran Hasanagic, and Kenneth Lausdahl. Code-generating VDM for Embedded Devices. In John Fitzgerald, Peter W. V. Tran-Jørgensen, and Tomohiro Oda, editors, *Proceedings of the 15th Overture Workshop*, pages 1–15. Newcastle University, Computing Science. Technical Report Series. CS-TR- 1513, September 2017.

[CDK05] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[Con13] Controllab Products B.V. http://www.20sim.com/, January 2013. 20-sim official website.

[CSK07] CSK. VDMTools homepage. *http://www.vdmtools.jp/en/*, 2007.

[FCC+16] Simon Foster, Ana Cavalcanti, Samuel Canham, Ken Pierce, and Jim Woodcock. Final Semantics of VDM-RT. Technical report, INTO-CPS Deliverable, D2.2b, December 2016.

[FCL+15] Simon Foster, Ana Cavalcanti, Kenneth Lausdahl, Ken Pierce, and Jim Woodcock. Initial Semantics of VDM-RT. Technical report, INTO-CPS Deliverable, D2.1b, December 2015.

[FE98] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90. Springer-Verlag, 1998.

[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.

[FPSP09] Peter Fritzson, Pavol Privitzer, Martin Sjlund, and Adrian Pop. Towards a text generation template language for Modelica. In Francesco Casella, editor, *Proceedings of the 7th International Modelica Conference*, pages 193–207. Linkping University Electronic Press, September 2009.

[Fri04] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, January 2004.

[FVB+16] Tommaso Fabbri, Marcel Verhoef, Victor Bandur, Maxime Perrotin, Thanassis Tsiodras, and Peter Gorm Larsen. Towards integration of Overture into TASTE. In Peter Gorm Larsen, Nico Plat, and Nick Battle, editors, *The 14th Overture Workshop: Towards Analytical Tool Chains*, pages 94–107, Cyprus, Greece, November 2016. Aarhus University, Department of Engineering. ECE-TR-28.

[HJ98] Tony Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, April 1998.

[HLG+15] Miran Hasanagić, Peter Gorm Larsen, Marcel Groothuis, Despina Davoudani, Adrian Pop, Kenneth Lausdahl, and Victor Bandur. Design Principles for Code Generators. Technical report, INTO-CPS Deliverable, D5.1d, December 2015.

[HLTJ15] Miran Hasanagic, Peter Gorm Larsen, and Peter W.V. Tran-Jørgensen. Generating Java RMI for the distributed aspects of

VDM-RT models. In *Proceedings of the 13th Overture Workshop*, pages 75–89, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan, June 2015. Center for Global Research in Advanced Software Science and Engineering. GRACE-TR-2015-06.

[JLC15] Peter W. V. Jørgensen, Morten Larsen, and Luís D. Couto. A Code Generation Platform for VDM. In Nick Battle and John Fitzgerald, editors, *Proceedings of the 12th Overture Workshop*. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446, January 2015.

[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.

[LFW09] Peter Gorm Larsen, John Fitzgerald, and Sune Wolff. Methods for the Development of Distributed Real-Time Embedded Systems using VDM. *Intl. Journal of Software and Informatics*, 3(2-3), October 2009.

[Lin15] Linköping University. http://www.openmodelica.org/, August 2015. OpenModelica official website.

[LLB11] Kenneth Lausdahl, Peter Gorm Larsen, and Nick Battle. A Deterministic Interpreter Simulating A Distributed real time system using VDM. In Shengchao Qin and Zongyan Qiu, editors, *Proceedings of the 13th international conference on Formal methods and software engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 179–194, Berlin, Heidelberg, October 2011. Springer-Verlag. ISBN 978-3-642-24558-9.

[MIR04] MIRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems, October 2004.

[OL17] Julien Ouy and Thierry Lecomte. Railways Case Study 3, (Confidential). Technical report, INTO-CPS Confidential Deliverable, D1.3b, December 2017.

[OMG02] OMG. *The Common Object Request Broker: Core Specification.* November 2002.

[Sjö15] Martin Sjölund. *Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models.* Doctoral

thesis No 1664, Linköping University, Department of Computer and Information Science, 2015.

[Sun00] Sun. *Java Remote Method Invocation Specification*. 2000.

[TBW+17] Bernhard Thiele, Thomas Beutlich, Volker Waurich, Martin Sjölund, and Tobias Bellmann. Towards a standard-conform, platform-generic and feature-rich modelica device drivers library. In J. Kofránek and F. Casella, editors, *Proceedings of the 12th International Modelica Conference*, pages 713–723. Modelica Association and Linköping University Electronic Press, May 15-17 2017.