Grant Agreement: 644047

INtegrated TOol chain for model-based design of CPSs

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

Technical Note Number: D5.2c

Version: 1.0

Date: December, 2016

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
Thierry Lecomte, CLE
Etienne Brosse, ST
Martin Peter Christiansen, 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
0.1	30-12-2015	Peter Gorm Larsen	Initial document version.
0.2	28-10-2016	Adrian Pop	OpenModelica updates.
0.3	31-10-2016	Marcel Groothuis	20-sim updates.
0.4	1-11-2016	Victor Bandur	Final draft for internal review.
0.5	14-12-2016	Adrian Pop	OpenModelica updates.
0.6	14-12-2016	Marcel Groothuis	20-sim updates.
1.0	15-12-2016	Victor Bandur	Final version for external review.

Abstract

This deliverable details the state of the INTO-CPS tool chain code generation capability at the end of project year 2. 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. The focus of this deliverable is on code generation in the context of the latter.

Contents

1 Introduction **6**

2 Background and Related Work **6**
2.1 Modelica 6
2.2 SIDOPS+ and Bond Graphs 7
2.3 VDM 9

3 FMI Code Generation with OpenModelica **9**

4 FMI Code Generation with 20-sim **12**
4.1 Code Generation Principles 12
4.2 Code Generation Capabilities 14
4.3 FMU Capabilities 15
4.4 3D Animation Viewer FMU 16

5 FMI Code Generation with Overture **17**
5.1 VDM and Code Generation 17
5.2 Semantics of VDM-RT 19
5.3 Achieving Translation 21
5.4 Translating Features of VDM-RT 42
5.5 Ambiguities Not Addressed by the Semantics 54
5.6 Implementation as Overture Plugin 55

6 FMU Compilation Service **56**

7 Conclusions **56**

1 Introduction

This deliverable describes the maturing code generation capability of the INTO-CPS tool chain at the end of project year 2. 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.

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.

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

[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/`.

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.

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

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 and so on.

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 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

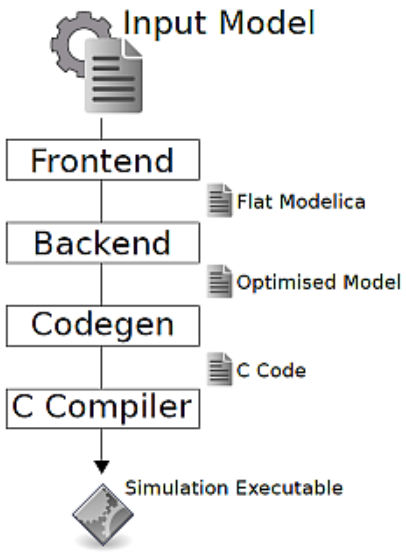

Figure 1: OpenModelica compilation phases.

[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++ 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.

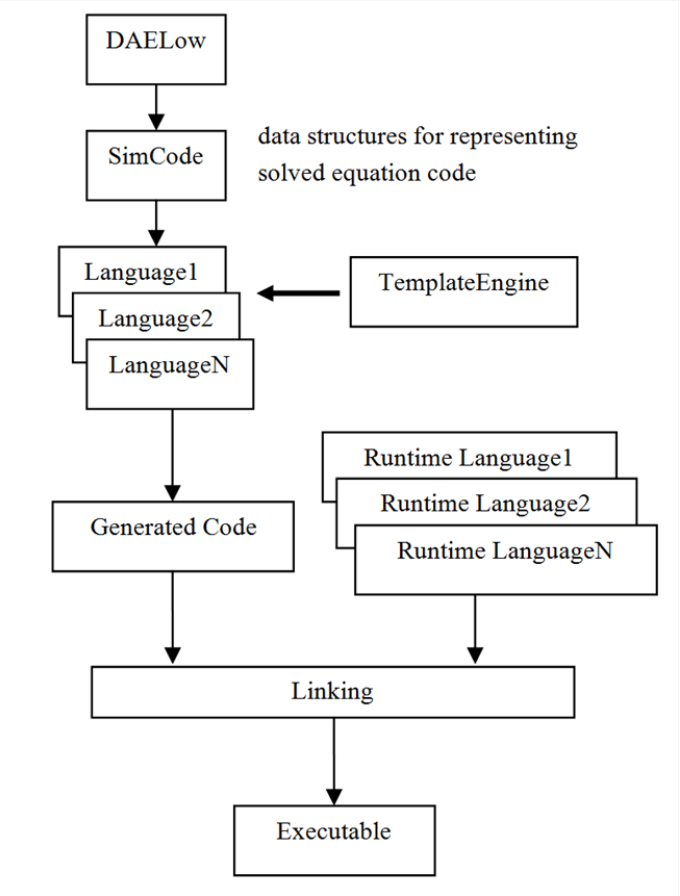

Figure 2: OpenModelica code generation using templates.

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. Currently 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. This capability is currently under development.

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 the model and translates it into a hybrid Differential Algebraic Equation (DAE) system. This DAE system is transformed into a causal form

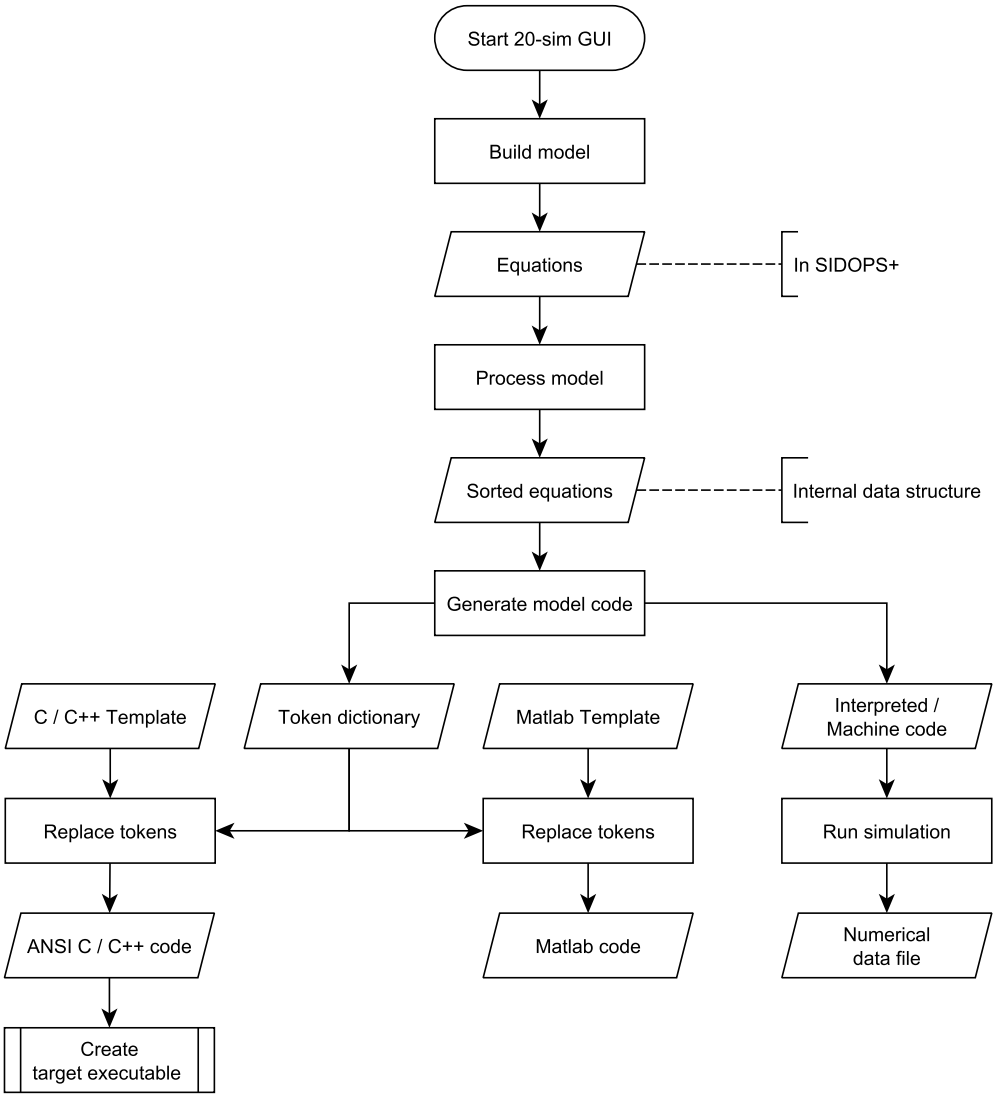

Figure 3: Flowchart of code generation from 20-sim.

(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 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 separate.
- **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.
- **Event functions:** `timeevent()`, `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 currently 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 "`[ ]`").

The following FMI features are not yet supported, but planned:

- Getting and setting the complete FMU state;

- Getting partial derivatives;
- Definition of used units;

4.4 3D Animation Viewer FMU

During the first year of the INTO-CPS project, it became clear that signal plots of a co-simulation experiment are not always the best way to check and represent the results. For multiple case studies, there was a clear wish for a 3D visualization of the co-simulation results similar to the existing 3D animation feature of 20-sim. A standalone 3D animation FMU has been developed to allow visualization during the co-simulation. Figure 4 shows an example of this FMU for the INTO-CPS line-follower case study.

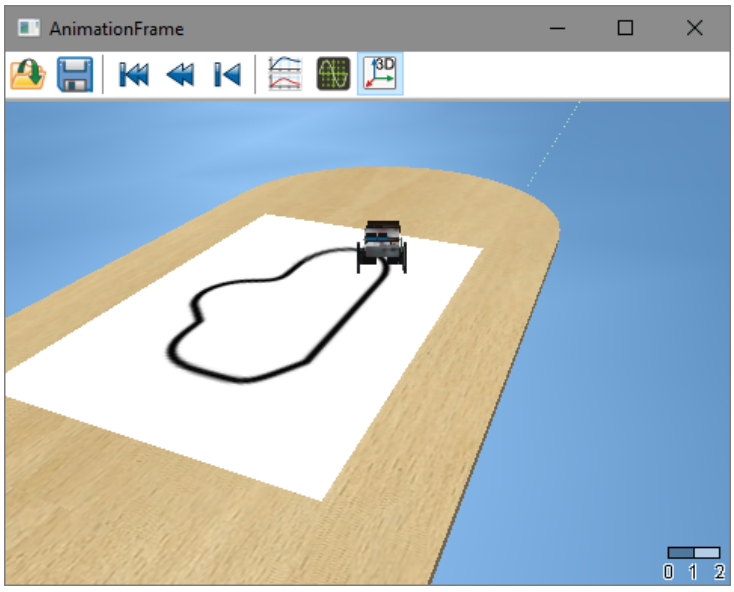

Figure 4: 3D animation FMU

20-sim can export an existing 3D animation plot as scenery XML (details can be found in deliverable D4.2a [BLL+16].) Based on this XML code and pre-built animation DLLs (Windows 32- and 64-bit only) one can manually generate a standalone 3D animation FMU. The limitation of the current 3D animation FMU is that the animation follows the co-simulation speed. When the co-simulation is slower or faster than real-time, the animation will follow the same speed. Ongoing work is to address this by converting the FMU to a tool-wrapper FMU that stays open after the actual co-simulation

experiment. A replay option is planned to replay the animation at real-time speed using recorded data from the co-simulation.

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 had to be developed from scratch. 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 important. 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.

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_RECORD, VDM_CLASS
} vdmtype;

typedef union TypedValueType {
  void* ptr;              // VDM_SET, VDM_SEQ, VDM_CLASS,
                          // VDM_MAP, VDM_PRODUCT
  int intVal;             // VDM_INT and INT1
  bool boolVal;           // VDM_BOOL
  double doubleVal;       // VDM_REAL
  char charVal;           // VDM_CHAR
  unsigned int uintVal; // VDM_QUOTE
} TypedValueType;

struct TypedValue {
  vdmtype type;
  TypedValueType value;
};

struct Collection {
  struct TypedValue** value;
  int 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`.

[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.

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. 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
#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).

[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.

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 VDMCG

#include "Vdm.h"

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


extern TVP numFields_1;

#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"

extern TVP numFields_2;

#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 that of single inheritance.

5.3.4 Memory Management

Currently the code generator attempts to manage memory usage by emitting calls to `vdmFree()` based on the role of new `TVP` variables. The level of abstraction of VDM-RT from which the code generation process starts means that the strategies for freeing all allocated `TVP` values are difficult to implement. For example, the VDM expression

```
1 + 2
```

currently 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. Work is ongoing to implement memory freeing strategies such that memory allocated as in this example can be freed at the appropriate place with corresponding calls to `vdmFree()`.

A parallel effort aims to develop a garbage collection 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 been modified to accept the address of the memory location from which the allocated memory is referenced (a pointer to `TVP`). A table is kept recording the relationship between these two locations. When allocating intermediate memory as in the example above, a null pointer is passed indicating that this memory is safe to reclaim once the containing statement has finished executing.

When the memory reclamation mechanism is executed, the value at each referencing location is checked against the corresponding address held in the allocation table. If these are not the same, then the memory in question can no longer be accessed from the corresponding location, and a call to `vdmFree()` is executed on it. If these values are the same then it is assumed that the reference is still in some scope, and therefore the memory referenced by it is still in use. The problem of variable scoping is handled based on the assumption that the evolution of the call stack will *eventually* overwrite local variables holding references to allocated memory. For example, assume that a function allocates a `newInt()` to variable `ni`. The variable `ni` is allocated on the call stack. The garbage collector is passed the address of `ni` through the call to `newInt()`. When the function exits, the stack pointer is modified

such that the next function invocation will make use of the same stack space. When this happens, there is a high probability that the value at the address of `ni` will be overwritten. This probability increases with subsequent function calls.

This 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. In this pattern of execution it is natural to invoke the garbage collector each time the periodic task has finished executing. This means that the user is granted the freedom to invoke the garbage collector anywhere, but in keeping with the philosophy that the generated code is meant as a black box, the collector is likely invoked in the code that is (necessarily) hand-written by the user to call into the model. As a result, the generator proper makes no decisions on where to invoke the garbage collector, simplifying the strategy further. This also allows the user to experiment with the granularity with which the garbage collector is invoked.

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.

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

The assessment compares the memory performance of the code generator on this model in its current state, its performance using the prototype garbage collection scheme, and its performance in the ideal case where all allocated memory is freed explicitly. The results are summarized in Figure 5 for initial value of 77,031, which is known to take 350 steps to converge.

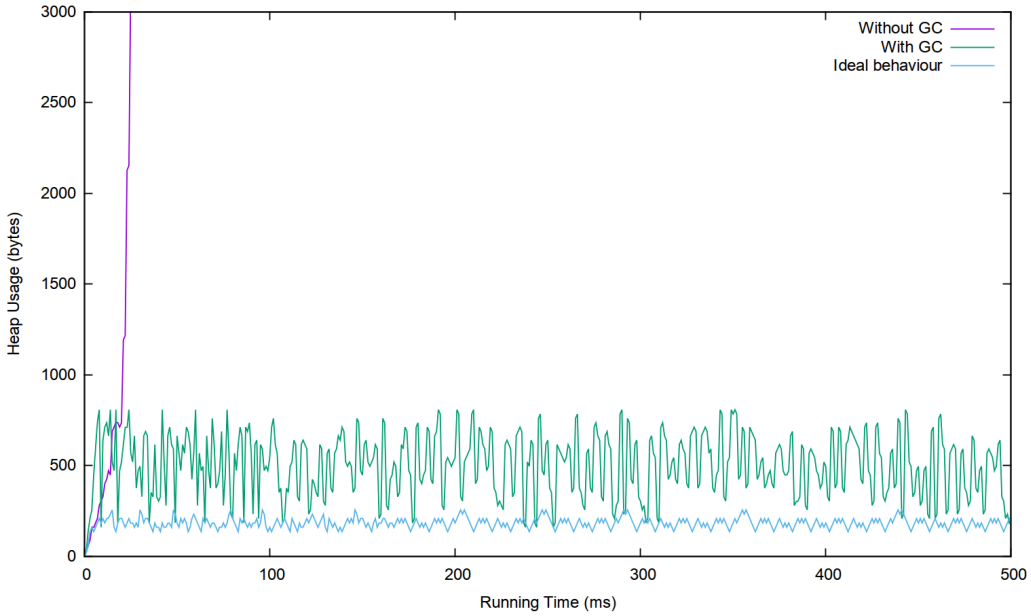

Figure 5: Memory performance of three implementations.

Because explicit memory management is not yet currently working fully, memory usage for the standard generated version of the code 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. Both results are promising in the context of embedded platforms with no real-time constraints.

5.4 Translating Features of VDM-RT

VDM-RT adds two novel modelling elements to VDM++: information for distributed architectures and timing information. This section describes as-

pects of how both can be addressed, and what approaches are currently taken.

5.4.1 Timing

In the context of code generation, timing information in a system model bears careful consideration, since its role can be interpreted in two very different ways: *prescriptive* or *descriptive*. By *prescriptive* we take a timing modelling construct, such as the VDM-RT duration statement to prescribe the duration of the corresponding statement in the code-generated implementation. Implementing this interpretation in a code generator is impossible without guarantees about the target architecture's timing behaviour. In very simple implementations for basic hardware, this can come directly from the hardware manufacturer. In more complex setups this can come from guarantees made by a real-time operating system.

The second interpretation of timing information is *descriptive*. In this case a faithful implementation of a VDM-RT model is observed in execution on its target platform and timing measurements are made using the appropriate instruments. This timing information is then fed back into the model, where annotations are made in the corresponding places. For instance, a model may specify the toggling of a pin on a digital I/O port. The implementation can be measured and the time required for the pin to flip its state – a function both of the execution speed of the controller in question, as well as of the material properties of the hardware – can be ascertained. The statements in the model responsible for this toggling action can then be annotated correspondingly. Such annotations are compositional. If a model is completely annotated with concrete timing information, then analysis can lead to predictions on the timing behaviour of the implementation in scenarios other than those in which it was first observed.

The C code generator is built around the latter interpretation for timing model elements. Consequently, all timing annotations are ignored and no effort is made on the part of the code generator to *obey* any timing annotations. To do so would require pairing the generated code with a real-time operating system. This is discussed briefly in Section 7.

With respect to the INTO-CPS semantics deliverable D2.2b [FCC+16], timed and untimed semantics for VDM-RT statements are given, depending on whether the statement in question falls under a `duration` annotation. Our decision here allows us to focus on respecting only the untimed seman-

tics.

5.4.2 Distributed Architectures in VDM-RT

This section describes the main principles of code generation for the distributed aspects of VDM-RT. Code generation of the distribution features will be explained by means of an example illustrating the main principles. The example is inspired by the more generic model called "System" used in the VDM-RT semantics deliverable D2.2b [FCC+16].

As previously described, the VDM-RT dialect extends the VDM++ dialect with support for modelling distributed systems. The distribution aspects are captured inside the `system` definition, as shown in the example in Listing 21. Distribution modelling 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. Both classes allow the specification of the speed of each, execution speed for `CPU` and communication speed for `BUS`. However, in this section we focus only on the distribution features.

Listing 21 shows an example of modelling distribution inside the `system` definition. First, two instances `s1` and `s2` of class `Sensor` are created and afterwards they are used to initialize an instance `c` of the class `Controller`. Second, the system architecture is modelled by creating two computational units `cpu1` and `cpu2`, of the implicit class `CPU`, and connecting them with a communication channel using the implicit class `BUS`. Third, the instances are deployed to the computational units, in this example `s1` and `c` are deployed to `cpu1`, while `s2` is deployed to `cpu2`. Hence the controller object `c` will have access to both a *local* object `s1` (*e.g.* deployed to the same CPU), and a *remote* object `s2` (*e.g.* deployed to a different CPU). Objects used inside the `system` definition are referred to as *distributed objects* and together these definitions form the connection topology of the distributed system.

It is important to note that, with respect to FMI, the approach taken is *one model, one FMU*: the type of distribution described here must be understood as *intra*-FMU, and not *inter*-FMU. The intent is that the distributed objects fall under the umbrella of a single FMU, not several as would be dictated by the distribution topology.

Listing 21: Example of distribution in VDM-RT.

```
system D
instance variables
  public static s1 : Sensor := new Sensor();
  public static s2 : Sensor := new Sensor();
  public static c : Controller := new Controller(s1,s2);

  cpu1 : CPU := new CPU(<FP>, 22E6);
  cpu2 : CPU := new CPU(<FP>, 22E6);

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

operations
public D : () ==> D
D () ==
(
  cpu1.deploy(s1);
  cpu1.deploy(c);
  cpu2.deploy(s2);
);
end D
```

Listings 22 and 23 show the two classes used in Listing 21, `Sensor` and `Controller`, respectively. Listing 22 represents a simple sensor class reading a constant temperature. Listing 23 shows how the objects can be invoked. In this example `sensor_local` is the local object, while `sensor_remote` is the remote object (based in the initialization of the object `c` in Listing 21). Both local and a remote calls are invoked as object method invocations, *e.g.* as Listing 23 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. Hence in this case the call `sensor_remote.readTemp()` will be sent over the `BUS` instance `bus`, due to the topology specified in the `system` definition.

Listing 22: Example of sensor class, returning a temperature value.

```
class Sensor

operations
public readTemp : () ==> int
readTemp () == return 2;

end Sensor
```

Listing 23: Example of controller, setting up references to sensor objects and calling their operations.

```
class Controller

instance variables
sensor_local : Sensor;
sensor_remote : Sensor;
```

```
operations
public Controller : Sensor * Sensor ==> B
Controller (s1,s2) == (
sensor_local := s1;
sensor_remote := s2;
);

public callLoc : () ==> int
callLoc () == return sensor_local.readTemp();

public callRem : () ==> int
callRem() == return sensor_remote.readTemp();

end Controller
```

5.4.3 Aspects of Code Generating Distribution Support

Following the discussion above, the entire system architecture, with respect to distribution especially, is captured inside the `system` definition in a VDM-RT model. Based on this, we describe here the general principles for generating support for distribution in a VDM-RT model. Three different implementation aspects for these VDM-RT constructs are highlighted: the *`CPU` class*, the *`BUS`* class and *distribution of calls*.

The *`CPU` class* is used to model independent processor units or computation nodes and deploy objects to them. For this reason each `CPU` instance corresponds to an individual processor or computation node (*e.g.* in a networked cluster) in the implementation of a VDM-RT model. As a consequence, all objects local to a given `CPU` must be instantiated. On the same CPU, remote objects are represented, abstractly, as references. Furthermore, a CPU in a VDM-RT model can be created with both speed and scheduler information. However, in the current approach these aspects are not supported, and each CPU is assumed to allow only sequential (though arbitrarily interleaved) execution without scheduling.

The *`BUS` class* is used to model a communication channel between CPUs. But in VDM-RT a `BUS` is an abstract representation of a communication bus, supporting abstract protocols. It does not specify a hardware bus type. As a consequence, the target distributed execution platform is assumed to provide a corresponding hardware communication bus, for example a UART (Universal Asynchronous Receiver/Transmitter) or CAN (Controller Area Network) bus together with a possible protocol.

Distribution of method calls must be handled explicitly in the implementation of a distributed system. As discussed above, a VDM-RT model hides whether

an invocation is local or remote with respect to a CPU. However, supporting this in the implementation requires *dispatching* between local and remote calls for a given CPU with respect to object deployment and the system architecture as modelled in the `system` class.

In the following section we use these three main concepts to describe the basic principles for code generation support for distribution.

5.4.4 Code Generation Principles

This section describes, based the discussion above, the main principles governing VDM-RT model analysis and transformation for supporting distribution in an implementation. The main steps for the distributed code generator can be described as follows:

- Analyze and extract information about the system architecture and object deployment inside the `system` definition.
- Based on the extracted `system` class information, enable dispatching of object method invocations, *e.g.* either as local or remote calls based on the system architecture.
- For a local call, use the normal local call macro as shown in Listing 9. For a remote call, send the data necessary to invoke the method on the object deployed to a given CPU across the bus specified in the VDM-RT model.

These three points, architecture analysis, invocation dispatching and remote calls, are further elaborated in the following three subsections, respectively.

5.4.5 System Architecture Analysis

When generating support for the distributed aspects of a VDM-RT model, the relevant architecture information must be extracted from the `system` class definition. The following information must be extracted:

- Which `CPU` instances are created. This indicates the individual microprocessors of the system, *i.e.* each CPU will get its own code implementation. Specifically, each CPU receives its own local representation of the `system` definition, containing the objects deployed to it.

- Which `BUS` instances are created. This indicates the communication channels created in the system.
- How the objects in the `system` definition are deployed to each `CPU` instance. This indicates which object has to be instantiated with respect to the implementation of a specific `CPU` instance.
- How the `CPU` instances are connected based on the `BUS` instances. This indicates the communications architecture between the different processors.

These four parts will in combination support the realization of a VDM-RT model. This architecture analysis is automatically performed by the code generator. The four points above can be illustrated using the simple example from Listing 21:

- Two implementations will be code generated for the two `CPU` instances `cpu1` and `cpu2`, respectively. Furthermore, each implementation consists of a local version of the `system` definition, generated as a class, containing information about local and remote objects.
- The `BUS` instance will be realized by a specific hardware bus, *e.g.* UART or CAN bus, and a communication protocol.
- In this example the objects `c` and `s1` are deployed to the code implementation of `cpu1`, while `s2` is deployed to `cpu2`.
- The `CPU` instances `cpu1` and `cpu2` are connected by the `BUS` instance `bus`. Hence the communication between these two will go through the `bus` instance.

This system architecture information supports the invocation dispatching between local and remote method invocations, as discussed in the following section.

5.4.6 Invocation Dispatching

The following subsection describes how the dispatching between a local and a remote call, with respect to a VDM-RT model as discussed above, is handled when code is generated. The basic idea of call dispatching is illustrated in Figure 6. The `CALL_FUNC(...)` macro is wrapped in another macro which handles the dispatching: if it is a local call then the normal function macro is invoked, otherwise the remote dispatcher is invoked, as described further below.

In general, dispatching depends on the object deployment topology inside the `system` definition, as discussed above. Hence this deployment information must be used when supporting the distributed aspects of VDM-RT. Each CPU implementation receives its own local version of the `system` definition, generated as a class. This local definition contains both objects deployed to that CPU and a map of which distributed objects are local and which are remote relative to that CPU.

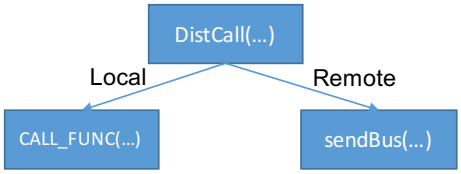

Figure 6: Dispatching method call locally or remotely.

The dispatching illustrated in Figure 6 is implemented in the code generator by giving each distributed object a unique number with respect to the `system` definition in a VDM-RT model during initialization. This ensures that all objects can be identified based on the unique ID in the whole distributed model. The motivation for giving each distributed object a unique ID number, starting from 1, will be clarified below. These IDs are assigned based on their order inside the `system` definition. Furthermore, it should be noted that with respect to any given CPU, the deployed objects are instantiated, while the remote objects only get an ID.

For each CPU, based on the distribution inside the `system` definition, a local map is created indicating whether each object is remote or local. This map is generated as an array of boolean variables called `DM`, where `true` indicates a local object, and `false` a remote object. For example Listing 24 shows the `DM` array with respect to `cpu1`: The first position, *i.e.* location 0 in C arrays, is used for locally created objects during run-time, hence this is always true. Positions 1 to the number of distributed objects are used for indicating the role of a distributed object based on their ID. Hence Listing 24 shows that with respect to `cpu1` the object with ID 1 is local (object `s1`), that with ID 2 is remote (object `s2`) and that with ID 3 is local (object `c`).

Listing 24: Distribution map with respect to `cpu1` based on Listing 21.

```
...
bool DM[4] = {true, true, false, true};
...
```

The distribution mapping inside the array `DM` is generated based on the system architecture described in the VDM-RT model. This information can now

be used for each CPU in order to dispatch a call either as local or remote. This enables the `CALL_FUNC(...)` macro to be wrapped inside a dispatcher macro, as shown in Figure 6. As a consequence, `DM` represents the local set of deployed objects with respect to a CPU, as described by the VDM-RT semantics, and documented in deliverable D2.2b [FCC+16]. This dispatcher is able, based on the distribution mapping, to either perform a *local* or *remote* call. For a local call the `CALL_FUNC(...)` macro is invoked. For a remote call a function to send data across a given bus is invoked. This remote call functionality is described in the following section.

5.4.7 Remote Method Invocation Handling

The following section describes the main principle with respect to a remote method invocation in a VDM-RT model, corresponding to `sendBus()` illustrated in Figure 6. The code generation for the remote method invocation is realized to provide two separate functions

1. Dispatching an object call to the correct bus based on a specific VDM-RT model.
2. Handling an incoming call from another CPU.

Hence this functionality essentially corresponds to a send and receive function between two CPUs, respectively. Additionally, communication in a VDM-RT model is point-to-point and proper access is ensured by the underlying protocol. These two functions in combination enable network communications, and are described next.

Figure 7 illustrates the flow between the send and receive functions 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. Currently asynchronous calls are not supported by the code generator, so the communication is required to be synchronous when implementing the specific call. This VDM-RT code generator generates support

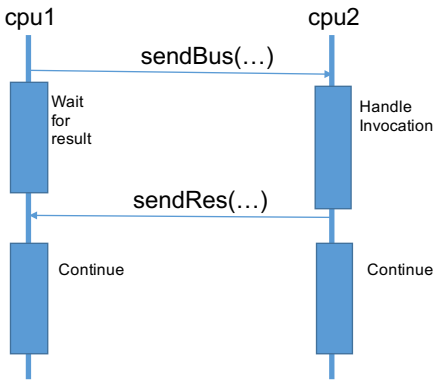

Figure 7: Flow diagram for `sendBus()` when sending data across a network.

for the high-level parts of the communications. The high-level parts for both sending and receiving functions are discussed next.

For the *send function*, the generator implements dispatching for a concrete bus depending on which object is invoked. Figure 8 shows the basic blocks of the send function: bus dispatch, data serialization, a protocol and the specific bus hardware (HW) API. As such the code generator supports bus dispatching to a concrete bus API, using the unique ID, while the low-level parts of this API can be adapted by users to their needs, as required by the case studies. Additionally, it shall be noted that these three lower level blocks are implemented for each concrete bus. This implementation for specific buses must capture both the specific protocol used and the actual HW bus type (*e.g.* UART). So 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. Hence these low-level blocks may be implemented toward user needs, providing flexibility. For example, the protocol and hardware bus can be implemented in accordance with the user needs. This follows from the project case studies, and additionally makes the distribution support more flexible by allowing proprietary implementations. However, initial support for serialization and de-serialization is provided and discussed below.

The *receive function* is responsible for handling the invocation when a remote call must be handled by the CPU on which the object is deployed. This function, just like the send function, is also generated with respect to objects deployed on specific CPUs. As such this function consists of the following blocks: object dispatch, data de-serialization, bus HW API. The code generator generates object dispatch for each individual CPU that is based on

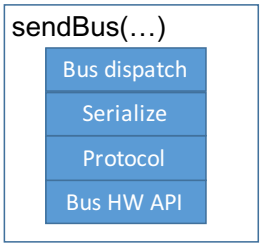

Figure 8: Structure of `sendBus()` function.

the IDs of the distributed objects. However, the data must be de-serialized before invoking the object dispatcher. After the data is de-serialized, the object dispatcher is able to reconstruct the call based on the object ID number and the function number, and the function arguments received. Finally, when the invocation is handled, the result is sent back to the invoking CPU by the `sendRes()` function, as shown in Figure 7.

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 this currently is 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 9.

First each CPU must be initialized, *i.e.* initialize all local values and objects, as described previously. Next, a check is performed to determine whether data is waiting to be received. If yes, then it handles the invocation, else it runs the periodic thread. Hence this design pattern is based on a polling mechanism, *i.e.* periodically checking whether data can be received. However, the handle receive function could also be implemented as an interrupt service routine. However, this may update variables during the execution of the periodic thread, which is not acceptable for the INTO-CPS case studies. For this reason the design pattern shown in Figure 9 has been adopted for the case studies for ClearSy and UTRC, which have distributed architectures. Were an operating system available, the remote method handling could be handled by its scheduler, similar to the scheduler semantics of a VDM-RT CPU.

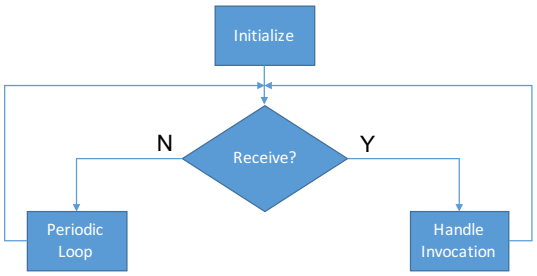

Figure 9: Design pattern enabling a CPU to handle remote invocations while running a periodic thread.

Serialization The functionality described above indicates that it is necessary to send data across a network. For this reason, the data must be serialized and de-serialized. For the basic types of VDM-RT this serialization may be straightforward. However, other types call for more complex serialization and de-serialization schemes. This serialization can be achieved either by introducing bespoke encoding together with encoders and decoders. However, it can be hard to ensure that data is transferred correctly on all possible CPUs and platforms. On the other hand, generic message representation can be used, such as XML or JSON. However, this requires generic parsers at run-time, causing memory and CPU overhead. Additionally, messages are encoded in verbose representations, adding communication overhead. Finally, it is possible to use the Abstract Syntax Notation One (ASN.1)[12]. This notation provides a data description language, and the ASN.1 compiler can produce corresponding encoder and decoder functions.

With respect to the above discussion, the VDM-RT code generator will use the ASN.1 notation as part of supporting serialization. In ASN.1, the exchanged data types can be described in a data description language. Initial 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]. 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. Research is being carried out on using the ASN.1 compiler developed by the European Space Agency (ESA), as found at `https://github.com/ttsiodras/asn1scc`.

With respect to the INCO-CPS case studies using distribution in code gen-

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

eration, only basic types support is currently required. As such using this ASN.1 notation is optional for these basic types. For this reason, serialization can be added depending on needs, as an underlying layer, supporting the low-level implementation. Currently, work is being carried out to convert specific VDM types introduced in a VDM model to their ASN.1 representation in order to generate encoders and decoders.

5.4.8 VDM-RT Model Implementation and Limitations

Since the VDM-RT notation allows great flexibility in modelling a distributed system, it may not be possible to generate support for its implementation unless some guidelines are followed. Some limitations and guidelines are discussed below.

As described in the VDM-RT semantics, a VDM-RT model also has a virtual CPU, where all resources that are not deployed to a `CPU` instance inside the `system` definition are deployed. Furthermore, this virtual CPU is connected to all other CPUs by a virtual bus, which is infinitely fast. The virtual CPU and bus are useful for testing and simulation purposes, but should be limited to not containing any of the implementation of a distributed system. Hence everything that should be part of the implementation should be deployed to real CPU instances in a VDM-RT model. As such the virtual CPU and bus can be used for testing and simulation. This also includes avoiding using `public static` declarations outside the `system` definition, since semantically they are placed on the virtual CPU. Additionally, if two CPUs are connected by two buses, according to the VDM-RT semantics an arbitrary bus is chosen for any given communication. In order to avoid building non-determinism resolution into the code generator, such structures should be avoided in VDM-RT models.

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 yet supported.
8. *Are constructors inheritable?* Constructors are not inherited.
9. *Overriding/overloading polymorphic/curried functions*: Not yet treated.
10. *Pre-post conditions in OO state context*: Pre- and post-conditions are not yet 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 10.

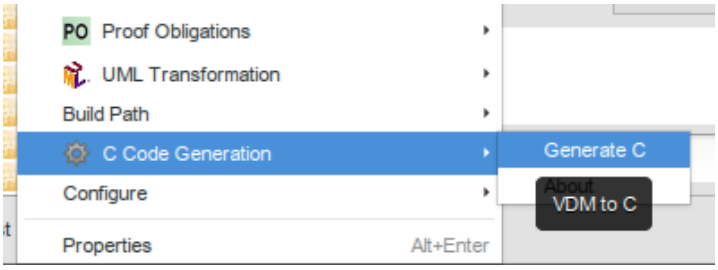

Figure 10: Invoking the code generator.

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 can also be accessed using the INTO-CPS application. Both the Overture FMU export feature and the INTO-CPS application are documented in the INTO-CPS user manual [BLL+16].

7 Conclusions

This deliverable captures the FMI-compliant C and C++ code generation capability of the INTO-CPS tool chain at the end of project year 2. The three tools, 20-sim, OpenModelica and Overture, can now fully integrate their code generation capability 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 mature, requiring mostly development of standalone FMU export. Next steps for OpenModelica include generation of code that can be deployed to embedded platforms. On the part of 20-sim, the facilities of 20-sim 4C are being expanded to facilitate deployment to a growing number of embedded platforms, driven by the project case studies. 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.

Beside expanding the generator's language coverage capabilities, currently driven by pilot and industrial case studies, the primary target is to generate C code with good enough memory behaviour and footprint that it can be deployed on resource-constrained embedded platforms. This effort is currently targeting PIC and ATmega microcontrollers.

Further development of code generation support for the distribution aspects of VDM-RT will also be addressed. Specifically, we will focus on supporting the specific bus technologies and related protocols in the case studies, by providing library implementations of some of the lower blocks in Figure 8. This work will be carried out in close collaboration with the owners of the case studies using distribution, namely UTRC and ClearSy. Furthermore, integration of the concrete protocols together with the generated C code for distribution will be addressed. Finally, the serialization using ASN.1 for more complex data structures is currently being researched, and will be integrated as well.

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 pre-/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. 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.

[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+16] Victor Bandur, Peter Gorm Larsen, Kenneth Lausdahl, Casper Thule, Anders Franz Terkelsen, Carl Gamble, Adrian Pop, Etienne Brosse, Jrg Brauer, Florian Lapschies, Marcel Groothuis, Christian Kleijn, and Luis Diogo Couto. INTO-CPS Tool Chain User Manual. Technical report, INTO-CPS Deliverable, D4.2a, December 2016.

[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.

[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.

[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.

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

[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.