Code-generating VDM for Embedded Devices

Victor Bandur, Peter W. V. Tran-Jørgensen,
Miran Hasanagić, and Kenneth Lausdahl

Department of Engineering, Aarhus University, Denmark
{victor.bandur,pvj,miran.hasanagic,lausdahl}@eng.au.dk

Abstract. Current code generators for VDM target high-level programming languages and hardware platforms, and are hence not suitable for embedded platforms with limited memory and computational power. This paper presents an overview of a new component of the Overture platform, a code generator from a subset of VDM-RT to ANSI C, suitable for such embedded platforms, as well as FMI-compliant co-simulation. The subset includes object-orientation and distribution features. This component is delivered as part of the Horizon 2020 project INTO-CPS and the generated code conforms to a novel semantics of VDM-RT developed under the same project.

Keywords: VDM, code generation, C, embedded platforms, Overture tool, FMI

1 Introduction

Current code generators for VDM [7] target high-level programming languages such as Java and C++. This severely limits the choice of target hardware platform, as only C compilers are available for many platforms with limited computational power and memory, such as microcontrollers. In this paper we address this by proposing a VDM-to-C translation, called VDM2C [27], that targets such platforms. VDM2C translates models written using VDM's real-time dialect, VDM-RT [19], into ANSI C, compliant with the C89 standard. Compliance with C89 ensures that a wide range of hardware platforms are supported, as many only have C89 compilers.

VDM2C is developed as part of the Horizon 2020 project INTO-CPS [18] to support (1) the implementation of VDM-RT models via code generation, and (2) Functional Mock-up Interface (FMI) compliant co-simulation [2] of VDM-RT models exported as Functional Mock-up Units (FMUs) embedding code-generated versions of the models. Both of these features are implemented in two separate plugins that can be installed in Overture [4]. In this paper we focus mostly on the C translation, and demonstrate by example how a VDM-RT model of a water tank controller is translated into C code that is executed in co-simulation. In the INTO-CPS project VDM2C has supported both system analysis (co-simulation) and implementation (code-generation) activities for several industrial case study models (see section 5).

VDM2C distinguishes itself from other VDM code generators currently available by targeting embedded devices. VDMTools [9] supports code generation of VDM-SL and VDM++ to both Java [12] and C++ [13, 14]. Overture only offers VDM-SL- and

VDM++-to-Java code generation, although there exist prototype versions of VDM-RT-to-Java [17, 29] and VDM++-to-C++ code generators that are not yet included in the tool [16]. Tran-Jørgensen [28] gives a comprehensive description of code generation for VDM.

Common to all these code generators is the fact that they target object-oriented languages, which means that VDM's modularisation features (modules and classes) either have a one-to-one correspondence in the target language, or can easily be represented in the generated code using (say) classes. C, on the other hand, lacks adequate native support for modularisation, and therefore VDM modules and classes cannot easily be translated into C. To address this, VDM2C implements a name mangling scheme.

VDM2C further distinguishes itself from other code generators in the way it manages memory in the generated code: memory is allocated via a *runtime* and deallocated using a user-guided garbage collector. This approach to memory management is different from that of other high-level languages as it requires that the garbage collection routine be run manually by the user. Although this approach necessitates minor user intervention, it has the advantage that desired timing requirements can more easily be met, as the garbage collection routine is never called unpredictably by the runtime. For comparison, memory is managed using garbage collection in Java, and in C++ memory can to some extent also be managed transparently using constructs such as smart pointers.

This paper is organised as follows: section 2 introduces the technologies that VDM2C targets; section 3 describes the architecture of VDM2C, including its supported feature set and limitations; section 4 demonstrates the translation by example; section 5 provides an assessment of the translation and describes applications of VDM2C in the INTO-CPS project. Finally, section 6 concludes this paper and discusses future plans.

2 Background

VDM-RT is a superset of VDM++ [8] that adds facilities for specification of distributed software systems. In VDM-RT, a system architecture is modelled in the **`system`** class using special class constructs for CPUs and buses. Objects deployed to (that is, specified to execute on) one CPU can invoke a function or operation on an object deployed to a different CPU, which causes data to be send across the connecting bus. In addition to the user-defined CPUs there exists a virtual CPU, which by default runs infinitely fast without affecting system time. Objects that are not explicitly deployed to one of the user-defined CPUs are deployed to the virtual CPU. The virtual CPU is typically used for objects that represent elements that are external to the system, such as the environment. Every CPU is connected to the virtual CPU via the virtual bus. VDM2C uses the connection topology information found in the **`system`** class to generate the code that enables communication between objects deployed to different CPUs.

The subset of VDM-RT relevant here consists only of deterministic language constructs, those for which the code generator does not need to exercise a choice in translation. For instance, the statement **`let`** `a = 3` **`in`** `a` specifies that `a` may only take on the value 3, whereas the statement **`let`** `a` **`in set`** `S` **`be st`** `a > 1` **`in`** `a` allows `a` to take on any value in `S`. When translating the latter statement, a code genera-

tor must choose one of these values to assign to the entity implementing `a`. Furthermore, when translating the timing features of VDM-RT (**`duration`** and **`cycles`**) it is necessary to fix the meaning of such constructs. These constructs can be interpreted in two different ways. The first interpretation is *prescriptive*, in the sense that a statement annotated with the statement **`duration`** `X` must be guaranteed to take `X` time units when implemented. The second interpretation is *descriptive*, in the sense that an annotation such as **`duration`** `X` records the fact that the corresponding implementation *is known* to take `X` time units to execute. VDM2C does not generate real-time implementations, so the interpretation adopted is the latter. Section 3.4 discusses language feature support further.

3 Architecture of VDM2C

3.1 Implementation

VDM2C is implemented using Overture's Code Generation Platform (CGP), a framework that facilitates the implementation of code generators for VDM [17]. The workflow of the CGP is as follows. First, the CGP creates an Intermediate Representation (IR) of the VDM model subject to code generation. This IR, which initially mirrors the structure of the VDM model's abstract syntax tree, is used as a representation of the generated code. Afterward, the IR is subjected to a series of behaviour-preserving transformations in order to bring it to a form that is easier to translate to code. In its final form, all nodes in the IR that are nontrivial to code-generate are replaced with ones that, ideally, can be transliterated into the target language. Generation of target language code (*e.g.* C) is enabled by the CGP's code-emission framework, which uses the Apache Velocity template engine at its core to aid the code emission process [21].

The translation of VDM to C is primarily achieved using CGP transformations. As an example, VDM2C uses a transformation to replace literals (*e.g.* **`true`**) with corresponding runtime calls that manage memory allocation and create the corresponding C value. As another example, quantified expressions and collection comprehensions, which C does not support natively, are transformed into collections of statements that perform the equivalent operation using iteration *etc.*

3.2 The runtime library

Implementations generated from VDM-RT models consist of two parts, the generated code and a native *runtime library*. The runtime library is fixed and does not change during the code generation process. The design of the runtime library is based on the following four sources [15, 5, 11, 10]. We illustrate its design here by means of simple VDM models.

The runtime 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 1.1. A pointer to `TypedValue` is **`#define`**d as `TVP`, and is used throughout the implementation.

```
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; int intVal; bool boolVal; double doubleVal;
  char charVal; unsigned int quoteVal;
} TypedValueType;
struct TypedValue {
  vdmtype type; struct TypedValue **ref_from;
  TypedValueType value;
};
struct Collection {
  struct TypedValue** value; int size; int buf_size;
};
```

Listing 1.1: Fundamental code generator data type.

Elements of this type are always dynamically allocated using the C function `malloc`. The decision to follow a dynamic allocation approach was justified as follows. First, the mandate of the INTO-CPS project does not include mission-critical software, making the risk of software crashes due to failed dynamic allocations acceptable. Second, dynamic runtime memory allocation allows the code generator to support a large subset of VDM-RT, such as collections of indeterminate size.

An element of `TVP` carries information about the type of the VDM value it represents and the value proper. Any element of this type only ever stores a relevant value in one of the fields of `value`. Therefore, in order to minimize unused space, the value storage mechanism is a C **`union`**, as it only takes up as much space as the largest field, in this case **`void`**`*`. Members of the basic 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 structure can be reused. Functions that construct such basic values are provided, for example `TVP newInt(`**`int`**`)`, `TVP newBool(`**`bool`**`)` and `TVP newQuote(`**`unsigned int`**`)`. All the operations defined by the VDM language manual on basic types are implemented one-to-one. Members of structured VDM types, such as sets and sequences, 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 that accommodate these different situations, for example `newSetVar(size_t, ...)`, `newSetWithValues(size_t, TVP*)` and `newSeqWithValues(`

`size_t, TVP*)`. These rely on two functions for constructing the inner collections of type **`struct`** `Collection` at field `ptr`: `TVP newCollection(size_t, vdmtype )` and `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 a structured type. All the operations defined in the VDM language manual on structured types are implemented one-to-one.

VDM's object-orientation features are fundamentally implemented in the runtime library using C **`struct`**s. In brief, a class is represented by a **`struct`** whose fields represent the fields of that class. The functions and operations of the class are implemented as functions associated with the corresponding **`struct`**. To demonstrate the translation, consider the example VDM class in listing 1.2.

```
class A
instance variables
  private i : int := 1;
operations
  public op : () ==> int
  op() == return i;
end A
```

Listing 1.2: Example VDM model.

The code generator produces the two files `A.h` and `A.c`, shown in listing 1.3 and listing 1.5, respectively.

```
...
#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);
```

Listing 1.3: Corresponding header file `A.h`.

The basic construct is a **`struct`** containing the fields and the virtual function table of the class, as defined in the runtime library, see listing 1.4:

```
#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
```

Listing 1.4: Macro for defining class virtual function tables.

The virtual function table contains information necessary to resolve a call to `op` in a multiple inheritance context, as well as a field that receives a pointer to the implementation of `op`.

The rest of the important parts of the implementation consist of the function implementing `op`, the definition of the virtual function table pointing to it and the complete constructor mechanism.

```
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);
  }
}
static  TVP _Z2opEV(ACLASS this) {
  TVP ret_1 = vdmClone(newBool(true));
  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;
}
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;
}
```

Listing 1.5: Corresponding implementation file `A.c`.

Construction of an instance of class `A` starts with a call to `_Z1AEV`. An instance of `A` is allocated and its virtual function table is populated with the pointer to the implementation of `op`, `_Z2opEV`. The latter name is the result of a name mangling scheme [30, 1]. As C does not provide an adequate modularization mechanism, this scheme is implemented in order to avoid name clashes in the presence of inheritance. A header file called `MangledNames.h` provides the mappings between VDM model identifiers and mangled names in the generated code. The scheme takes both the class name, member name and parameter types into account. Listing 1.6 shows the contents of the file for the example model.

```
#define A_op _Z2opEV
#define A_A _Z1AEV
```

Listing 1.6: File `MangledNames.h`.

The example populated `main.c` file, shown in listing 1.7, illustrates how to make use of the generated code.

```
int main() {
  TVP a_instance = _Z1AEV(NULL);
  TVP 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;
}
```

Listing 1.7: Example `main.c` file.

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, initially generated by VDM2C, and afterward adapted by the user, also contains helper functions that aggregate all these calls into corresponding global initialization and tear-down functions, called in a precise order. 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`, shown in listing 1.8, carries out the calculations necessary for calling the correct version of `_Z2opEV` in the presence of inheritance and overriding (which is not the case here). In this listing, the result of the function call 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`.

```
#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)
```

Listing 1.8: Macros supporting function calls.

3.3 Garbage collection

The Overture CGP is designed for code generation to programming languages with garbage collection support, such as Java, and as such makes no provision for explicit management of allocated memory. VDM2C takes advantage of the structure of INTO-CPS models of discrete controllers to implement a simple garbage collection scheme, such that the generator need not emit any specific memory management code. The model structure that makes a simple garbage collection scheme possible follows the usual cyclic nature of discrete controllers. It is known that after each execution of the control task, all values generated are of two types. In the first case, values are stored in objects' instance variables, which are known to persist for the lifetime of the objects. This can be treated explicitly. In the second case, all other values are deemed intermediate. At the end of one iteration of the control task, all these values can be reclaimed safely by the garbage collector.

3.4 Supported feature set and limitations

VDM2C supports a large subset of VDM with the exception of pattern matching, pre- and post-conditions, and invariants. With regard to concurrency, periodic threads are the only kind of concurrency construct that can be generated, a feature specifically implemented for INTO-CPS. Furthermore, only certain parts of the `IO`, `MATH` and `CSV` standard libraries are currently supported. A more complete description of the supported feature set is available in the Overture user manual [24].

With regards to the readability of the generated code, the authors are of the firm belief that in a model-driven setting, generated code must never be inspected or modified. However, generated code must inevitably be debugged. In support of this, the generated code includes annotations referring back to the source model so that potentially incorrect behaviour in the generated code can be traced back to it.

3.5 Models of Distributed Systems

VDM2C generates C code specific to each CPU according to the information in the **`system`** class. The objects, which are instantiated inside this special class, are subsequently referred to as distributed objects. A distributed object is considered to be either

local or *remote* with respect to a given CPU, depending on whether communication to that object requires information to be transmitted across a bus. A *local* method invocation is handled by the `CALL_FUNC` macro as described in section 3.2. A *remote* call, on the other hand, is routed via a bus to another CPU responsible for processing the invocation.

The main challenge in achieving Remote Method Invocation (RMI) in the generated code is addressed by establishing awareness of local and remote objects. One way to achieve this is by using an existing distribution technology based on RMI. In Overture's Java code generator this is achieved using Java RMI [26] as the enabling technology for remote calls [16]. For VDM2C, the CORBA [22] middleware would be a similar solution. However, when targeting embedded systems, CORBA would limit the generated code in two ways: it would not easily allow the use of propietary bus drivers and it would introduce a large performance overhead. For these reasons, and to gain full control of RMI, the VDM2C runtime library is extended with custom support for this feature.

During the code generation process, local objects are marked using the `VDM_CLASS` type to distinguish them from remote objects, which are marked with a different type. This distinction allows one to use a dispatch macro to wrap `CALL_FUNC` in order to decide whether to dispatch a call as local or remote, as shown in listing 1.9. Also note that, in that listing, the remote object is passed to a generic function called `send_bus`. Based on the system's architecture, this function is generated for each CPU in order to perform routing of remote calls via the correct bus. This routing mechanism relies on all distribution objects having unique IDs. These are assigned during the code generation process. Essentially, the `send_bus` function (for a given CPU) is a case analysis on the IDs of the objects reachable from that CPU, which calls the bus function corresponding to each distributed object. The individual bus functions are generated as skeletons and named according to the respective bus in the VDM-RT model. It is the responsibility of the user to implement these bus functions, as they depend on the specific communication protocol and hardware.

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

Listing 1.9: Dispatching a call as either local or remote.

The CPU receiving the remote call needs the ability to process the call and pass results back to the invoking CPU. Due to the nature of this synchronous communication flow, VDM2C generates a dispatching mechanism for each CPU that processes remote invocations based on object IDs. In addition, VDM2C provides functionality to encode VDM data types that are transmitted across networks. Hence the developer only needs to implement the platform-specific aspects of the communication, while VDM2C provides data serialisation and ensures correct function invocation at the application level. VDM2C uses the data description language Abstract Syntax Notation One (ASN.1) together with an ASN.1 compiler suitable for embedded systems [20] for this, together with the work of Fabbri *et al.* [6] for conversion to/from `TVP`.

For a distributed VDM-RT model to be supported by VDM2C, certain requirements must be met. First, all collections must have a maximum size defined. Although this is a direct consequence of the particular ANS.1 compiler being used [20], it is nevertheless also considered good practice when designing embedded systems that are significantly constrained in terms of resources. Second, all distributed objects that model system behaviour must be deployed to user-defined CPUs, and configured inside the **`system`** class. This is a consequence of VDM2C omitting generation of the virtual CPU, which is usually only used for the test environment, and therefore not considered a significant limitation.

4 Example

In this section we demonstrate how VDM2C can be used to generate and validate the implementation of a controller for the INTO-CPS Water Wank pilot study [23]. In this system a source delivers a constant flow of water to a *water tank* that is equipped with a sensor monitoring the water level in the tank. The water level is regulated using a valve that, when opened, causes water to leave the tank. The *controller* tries to keep the water level between a minimum and a maximum threshold by opening and closing the valve.

Since the tank is a physical system, it is best described using a continuous time formalism such as differential equations. The control logic, on the other hand, is best expressed using a discrete event formalism, so this part of the system is modelled in VDM. For validation purposes, FMI-compliant co-simulation is used. The water tank and controller are exported as two separate FMUs, standalone and tool-wrapper, respectively. The FMI standard [2] contains further information on FMUs and co-simulation.

4.1 The generated C code

When the VDM model has been validated, VDM2C can be used to translate it to C code. The generated code can be executed or validated by either (1) creating an entry function that uses the generated code directly, or (2) using Overture's FMU exporter.

An example of the first approach is shown in listing 1.10. First, the garbage collector is initialised using the `vdm_gc_init()` call, which needs to be executed before calling the generated code. Afterward, **`static`** objects must be initialised. This is achieved by calling the `System_static_init` function. Next, the `loop` function is invoked once, which runs the control loop. Finally, the call to `vdm_gc` performs garbage collection as described above. Note that listing 1.10 only shows a single call to the control function. VDM2C cannot emit code that complies with the specified period for a given task such as `loop`, as this is platform-specific. Calling the task with the appropriate period must be achieved manually in most cases, using knowledge of the target hardware.

Using the second approach, the C code is embedded in an FMU. In this case, the exporter analyses the specified thread period and constructs a thread execution mechanism that runs the control task periodically. This mechanism does not rely on any operating system threading support. In a co-simulation setting, the thread execution mechanism ensures that the thread calling frequency matches the thread period specified in the VDM model.

```
int main(void) {
 vdm_gc_init();
 System_static_init();
 CALL_FUNC(Controller, Controller, g_System_controller,
     CLASS_Controller__Z4loopEV);
 vdm_gc();
 return 0;
}
```

Listing 1.10: Example of how to use the generated code.

4.2 Validation

Using the FMU exporter, it is possible to generate and test both the tool-wrapper and standalone water tank controller FMUs against the water tank FMU. As shown in figure 1, two co-simulations are run for 30 seconds each using the same water tank FMU: one of the co-simulations uses the tool-wrapper FMU, and the other co-simulation uses the standalone FMU. Throughout the simulations the controllers are expected to keep the water level between 1 (the minimum threshold) and 2 (the maximum threshold). Figure 1 only shows the water level (indicated using the top-most plot) for one of the experiments since the water levels for both experiments are similar and this keeps the figure simpler. In addition, the figure shows the two valve output plots for both the tool-wrapper FMU and the standalone FMU (the bottom-most plots), which are indistinguishable as they overlap. The fact that the two valve output plots overlap means that both the tool-wrapper and standalone FMUs behave identically (they produce the same valve output), which is the desired behaviour. Figure 1 further shows that the water level rises until it reaches the maximum threshold, then the controller opens the valve (the output becomes 1) and the water level drops to the minimum threshold, at which time the valve closes (the output becomes 0). Based on this, we conclude that the generated code behaves correctly.

Concerning deployment to embedded platforms, the generated watertank controller occupies 10 KB of flash memory when compiled for an AVR ATmega 1284p microcontroller with the AVR Libc toolchain [25]. This includes the generated controller code and the runtime library.

4.3 Deployment to Hardware

Deployment refers to the final stage of putting the compiled C code onto an embedded device for execution. VDM2C does not provide any deployment capabilities by itself, and this must usually be done manually. However, having an FMI interface to the C code makes it possible to adapt existing deployment tools to support this process. In the INTO-CPS project, the tool tasked with this is the 20sim4C [3] tool. It exploits the FMI interface description for connecting to hardware ports and deploying the code. Specialized template files for each target platform provide, among other things, information about the port connections available on that platform. A graphical user interface can

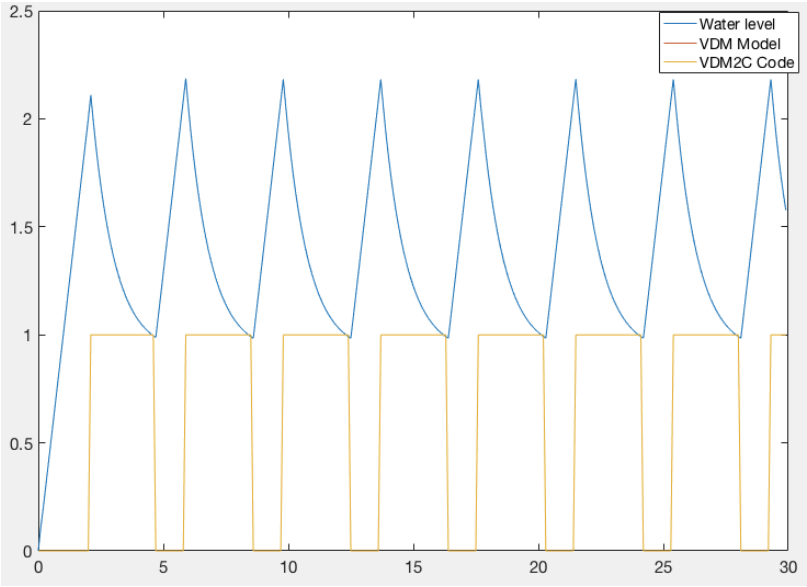

Fig. 1: Comparison of the value outputs of the tool-wrapper and standalone FMUs.

be used to then link the ports defined in the FMU's hardware port description to these hardware ports. The tool then compiles and downloads the code to the platform.

5 Assessment

We assess the code generator along multiple testing and performance trajectories. The functionality of the generator is ensured by a comprehensive test suite. Its performance in terms of binary size, speed and memory footprint is assessed separately. In this section we describe our approach to both. Correct behaviour of the code generator is first ensured through exhaustive testing. Language feature support development in the generator is driven by INTO-CPS industrial case studies and pilot studies. Once all language feature support for any given test model is added, the test model itself is added as a regression test to ensure that the code emitted by the generator for these models always compiles. Because the models are created in the context of co-simulation, the only way to further test that the generated code behaves correctly is by using a co-simulation test environment. It was decided to only carry out compilation tests for these models because functionality is further tested in co-simulation as part of other INTO-CPS development activities. The industrial case studies used are as follows:

– **UTRC**[1] **fan-coil unit** – Model of a controller whose purpose is to actuate several components of an HVAC unit to maintain constant ambient temperature. The model

[1] United Technologies Research Center, Cork, Ireland.

uses PID control, so its main contribution to the test suite is for correct translation of arithmetic expressions.
- **Agro Intelligence Lawnmower** – Model of a discrete controller for an autonomous lawnmower. This model additionally tests the `MATH` and `CSV` library implementations.

The INTO-CPS pilot studies are as follows:

- **Line following robot** – Model of a "bang-bang" controller for a line following robot that aims to keep a black path between two reflection sensors mounted at the front of the robot. The model tests basic decision logic and access to public class members.
- **Single water tank** – The example used in section 4.
- **Three cascading water tanks** – A simple variation on the single tank model.

Naturally as language feature support is added, unit tests for each feature are included in the test suite. More than compilation tests, these are the main functionality tests of each language feature. This is the second testing trajectory. The third trajectory is standalone testing of the runtime library. Similar to the testing suite for the generator, the runtime library test suite also contains functionality tests for each feature that is added in support of a language feature. The fourth and last trajectory is manual testing of the industrial and pilot case studies in co-simulation, as standalone FMUs. The generated code is assessed for performance in terms of execution speed, memory usage and binary size. The code generator is a product of the INTO-CPS project, where the goal is to emit code suitable for resource-constrained embedded platforms. Accordingly, the benchmark for execution speed is its performance relative to the requirements of the case and pilot studies requiring deployment to embedded platforms. These requirements have been satisfied for the line following robot described above as well as for a railway interlocking system. The line following robot is controlled by an AVR ATmega1284p[2] microcontroller, whereas the railway interlocking system is composed of multiple PIC32MX440F256H[3] microcontrollers.

6 Conclusion and future plans

In this paper we propose a translation from VDM-RT to C, called VDM2C, that uses a user-guided garbage collection scheme to manage memory in the generated code. Highlights and limitations of the translation were presented and demonstrated using a model of a water tank level controller. In addition to being exposed as a standalone code-generation plugin in Overture, VDM2C can also be used, as shown in this paper, to generate FMUs that embed code-generated VDM-RT models. In the INTO-CPS project, VDM2C has supported the development of several industrial case studies, specifically the co-simulation activities, as enabled by this FMU exporter.

One significant limitation of the code generator is that it is not currently possible to call the garbage collector at arbitrary locations in the generated code. As discussed earlier, in certain cases this limitation can lead to high, temporally-localized

[2] 8-bit microcontroller, 128 KB flash memory, 16 KB RAM.
[3] 32-bit microcontroller, 256 KB flash memory, 32 KB RAM.

memory usage, a behaviour that is unsuitable for some applications. This can be alleviated by making the reclamation mechanism more sophisticated at the expense of execution speed (mostly due to bookkeeping overhead.) Such an alternative memory management scheme can be implemented and made available in the runtime library, giving the user of the code generator the flexibility of choosing the scheme best suited to the application.

Looking forward, there are several immediate improvements that can be made in terms of extending the coverage of VDM2C to include a larger subset of VDM-RT. Specifically, we plan to add support for pattern matching, additional concurrency constructs and to improve coverage of Overture's standard libraries (see section 3.4). We envisage that adding full support for concurrency will be challenging as this feature usually is highly dependent on the specific hardware platform that is targeted.

As another item of future exploration, we plan to analyse the execution time and memory usage of the generated code by comparing our results to those obtained using other code generators for VDM. The immediate plan is to compare with the C++ code generator of VDMTools, as it has applicability to embedded platforms, but more general comparisons to Overture's Java generator and those of other model-based tools are envisaged. The goal of this analysis is to better understand the benefits and shortcomings of VDM2C, compared to other code generators, and to help us further improve the performance of the tool.

References

1. Avabodh: Name Mangling and Function Overloading. http://www.avabodh.com/cxxin/namemangling.html, accessed July 13 2017
2. Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A.: The Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: Proceedings of the 9th International Modelica Conference. Munich, Germany (Sep 2012)
3. Controllab: 20-sim 4C. http://www.nongnu.org/avr-libc, accessed August 30 2017
4. Couto, L.D., Larsen, P.G., Hasanagic, M., Kanakis, G., Lausdahl, K., Tran-Jørgensen, P.W.V.: Towards Enabling Overture as a Platform for Formal Notation IDEs. In: Proceedings of the 2nd Workshop on Formal-IDE (F-IDE) (2015)
5. EventHelix: Comparing C++ and C (Inheritance and Virtual Functions). http://www.eventhelix.com/RealtimeMantra/basics/ComparingCPPAndCPerformance2.htm, accessed July 13 2017
6. Fabbri, T., Verhoef, M., Bandur, V., Perrotin, M., Tsiodras, T., Larsen, P.G.: Towards integration of Overture into TASTE. In: Proceedings of the 14th Overture Workshop. Aarhus University, Department of Engineering, Cyprus, Greece (2016)
7. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008)
8. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
9. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices 43(2) (Feb 2008)
10. Go4Expert: How Virtual Table and _vptr works. http://www.go4expert.com/articles/virtual-table-vptr-t16544/, accessed July 13 2017

11. Go4Expert: Virtual Table and _vptr in Multiple Inheritance. http://www.go4expert.com/articles/virtual-table-vptr-multiple-inheritance-t16616/, accessed July 13 2017
12. Group, T.V.T.: The VDM++ to Java Code Generator. Tech. rep., CSK Systems (Jan 2008)
13. Group, T.V.T.: The VDM-SL to C++ code generator. Tech. rep., CSK Systems (Jan 2008)
14. Group, T.V.T.: The VDM++ to C++ code generator. Tech. rep., CSK Systems (Jan 2008)
15. Håkon Hallingstad: Classes in C. http://www.pvv.ntnu.no/~hakonhal/main.cgi/c/classes/, accessed July 13 2017
16. Hasanagić, M., Larsen, P.G., Tran-Jørgensen, P.W.V.: Generating Java RMI for the distributed aspects of VDM-RT models. In: Proceedings of the 13th Overture Workshop (Jun 2015)
17. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Proceedings of the 12th Overture Workshop. Newcastle University (Jan 2015)
18. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated Tool Chain for Model-based Design of Cyber-Physical Systems: The INTO-CPS Project. In: Proceedings of the CPS Data Workshop. Vienna, Austria (Apr 2016)
19. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Proceedings of the 13th international conference on Formal methods and software engineering. vol. 6991. Springer-Verlag, Berlin, Heidelberg (Oct 2011)
20. Mamais, G., Tsiodras, T., Lesens, D., Perrotin, M.: An ASN. 1 compiler for embedded/space systems. Embedded Real Time Software and SystemsERTS (2012)
21. The Apache Maven Project website. https://maven.apache.org (2017), accessed July 13 2017
22. OMG: The Common Object Request Broker: Core Specification. (Nov 2002)
23. Payne, R., Gamble, C., Pierce, K., Fitzgerald, J., Foster, S., Thule, C., Nilsson, R.: Examples Compendium 2. Tech. rep., INTO-CPS Deliverable, D3.5 (Dec 2016)
24. Peter Gorm Larsen, Kenneth Lausdahl, Peter Tran-Jørgensen, Joey Coleman, Sune Wolff, Luis Diogo Couto, Victor Bandur, N.B.: Overture VDM-10 Tool Support: User Guide. Tech. Rep. TR-2010-02, The Overture Initiative (May 2010)
25. Savannah: AVR Libc website. http://www.nongnu.org/avr-libc, accessed August 30 2017
26. Sun: Java Remote Method Invocation Specification (2000)
27. The Overture project: The VDM2C Github repository. https://github.com/overturetool/vdm2c, accessed July 13 2017
28. Tran-Jørgensen, P.W.V.: Enhancing System Realisation in Formal Model Development. Ph.D. thesis, Aarhus University (Sep 2016)
29. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML-annotated Java. International Journal on Software Tools for Technology Transfer (Feb 2017)
30. Wikipedia: Name mangling. https://en.wikipedia.org/wiki/Name_mangling, accessed July 13 2017