AARHUS UNIVERSITY

Coversheet

This is a post-peer-review, pre-copyedit version of an article published in *Design Automation for Embedded Systems*. The final authenticated version is available online at: <https://doi.org/10.1007/s10617-019-09227-0>. Contentwise, the accepted manuscript version is identical to the final published version, but there may be differences in typography and layout.

How to cite this publication

Please cite the final published version:

Hasanagić, M., Fabbri, T., Larsen, P.G. *et al.* Code generation for distributed embedded systems with VDM-RT. *Des Autom Embed Syst* **23**, 153–177 (2019). <https://doi.org/10.1007/s10617-019-09227-0>

Publication metadata

Title:	Code generation for distributed embedded systems with VDM-RT
Author(s):	Hasanagić, M., Fabbri, T., Larsen, P.G. <i>et al.</i>
Journal:	<i>Design Automation for Embedded Systems</i>
DOI/Link:	https://doi.org/10.1007/s10617-019-09227-0
Document version:	Accepted manuscript (post-print)

General Rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

If the document is published under a Creative Commons license, this applies instead of the general rights.

Code Generation for Distributed Embedded Systems with VDM-RT

Miran Hasanagić · Tommaso Fabbri · Peter Gorm
Larsen · Victor Bandur · Peter Tran-Jørgensen ·
Julien Ouy

Received: date / Accepted: date

Abstract Developing embedded systems that are distributed is a challenging endeavour, since they need to ensure system-wide properties as well as existence of a large number of possible candidate system architectures. Various model based techniques advocate raising the abstraction level in order to support a holistic view of such a distributed embedded system. Furthermore, automatically generating implementation specific code from models can support realisation efforts including avoiding inconsistencies between model and code. In this paper we present how such efforts can be aided for a distributed embedded system modelled in the real time dialect of the Vienna Development Method, VDM-RT, by means of automatic code generation. The contributions in this paper are 1) code generation capabilities for distributed embedded system modelled in VDM-RT; 2) demonstration of its applicability for an industrial case study involving a distributed interlocking system from the railways domain. Additionally, we discuss the balance between code generation for a model, which assumes idealised communication (no messages lost), and manually produced code, e.g. in the industrial case study legacy low-level code ensures fault-tolerant communication in the final implementation.

Miran Hasanagić
Aarhus University, Denmark
E-mail: miran.hasanagic@eng.au.dk

Tommaso Fabbri
University of Pisa, Italy
E-mail: tommaso.fabbri@for.unipi.it

Peter Gorm Larsen
Aarhus University, Denmark
E-mail: pgl@eng.au.dk

Victor Bandur
Aarhus University, Denmark
E-mail: victor.bandur@eng.au.dk

Peter Tran-Jørgensen
Aarhus University, Denmark
E-mail: pvj@eng.au.dk

Julien Ouy
ClearSy, France
E-mail: julien.ouy@clearsy.com

Keywords Distributed Embedded Systems · Code Generation · Co-simulation · Validation · VDM-RT

1 Introduction

Today many products involving development of embedded systems are inherently distributed, where individual nodes need to achieve a system-wide objective by communicating on a network. Moreover, there is an increasing need for such Distributed Embedded Systems (DESSs) as a consequence of increased intelligence in “smart” devices [17]. Conceptually, DESSs have a high level of complexity, especially when these systems must be resistant to different kinds of faults [14, 41]. Therefore, a significant amount of time is often spent on validation activities. Furthermore, if it is decided at some point in the development process to restructure the overall architecture of the system, then reusing a previous designs often becomes challenging. From a business perspective this increases the overall development costs as well as time to market.

Approaches that have been advocated to address complexity in dependable systems are model-based techniques, which support abstract modelling [9, 50]. Some of these notations enable design of system architectures [6, 48], and hence support development of DESSs. In particular, they support reasoning about core properties, and enable various facets of a System Under Development (SUD), such as architecture, gradually to be introduced. However, when models developed using such methods have been validated, the final software implementation is often manually produced and deployed to the different computing nodes that constitute the hardware platform of the system.

In this paper we present an approach that uses code generation to translate models written in the real time dialect of the Vienna Development Method (VDM-RT), which enables abstract modelling of DESSs, into C code. In our previous work [4] we focused on translating VDM-RT models in a non-distributed setting (i.e. a single node). Hence compared to [4], our work presented in this paper focuses on a distributed setting, and specifically contributes the following:

- first, it presents code generation capabilities for DESSs modelled in VDM-RT. In particular, we present new VDM-RT modelling guidelines together with a code generation tool, which in combination enable automatic realisation of such VDM-RT models, and
- second, it illustrates the capabilities using an industrial case study involving the development of a distributed train interlocking system.

Consequently, this paper, when compared to our work in [4], describes the considerations as well as presents new code generation extensions when generating C code for realising a DES from a VDM-RT model. This is achieved by presenting modelling guidelines together with new code generations extensions. So this code generator makes it possible to automatically produce an implementation that can be deployed to a collection of computing nodes. Hence we propose a method for automatically implementing models of DESSs which potentially could be adopted by other code generators supporting different formalisms. In addition to reducing development efforts through tool automation, this approach also reduces the probability of introducing issues in the final implementation through manual translation of model into code.

Usually embedded systems interact with the physical environment using various sensors and actuators [17]. Within this context, VDM-RT models may be validated by means of co-simulation in a Functional-Mock-up Interface (FMI) setting [5] enabling simulation

of models written in different formalisms [18]. Such co-simulation can, for example, be achieved using technology developed within the INtegrated TOol chain for model-based design of Cyber Physical Systems (INTO-CPS) project [26,27,45]. Hence the code generator, presented in this paper, further supports the VDM-RT methodology within such a context by enabling automated realisation in C code, which can be validated by co-simulation as well.

This paper is organised as follows. First, Section 2 describes the modelling principles of VDM-RT, specifically those that support development of DESs. Afterwards, in Section 3, we introduce modelling guidelines that enable code generation of VDM-RT models. This is followed by Section 4, which describes the code generation process and how architecture support is addressed in the generated code. We continue, in Section 5, by describing how INTO-CPS technology, including the code generator, has supported the successful development of an industrial case study from the railway domain. Afterwards, Section 6 describes related work, and finally, Section 7 concludes and presents future plans.

2 Modelling Distributed Embedded Systems in VDM-RT

The VDM-RT notation extends the modelling language VDM++ [15] that facilitates object oriented modelling together with concurrency, with functionality for designing real-time computations [33] and statically distributed systems [48,47]¹. In addition, VDM++, itself, extends VDM-SL [28], which is an ISO standardised formal method enabling basic modular structuring of sequential systems. From an overall design perspective VDM-RT supports a holistic development approach of DESs as well as using a range of various analytical techniques, ranging from animation to mathematical proof, as a means to ensure the consistency of a model and its correctness with respect to requirements.

In a distributed system the embedded devices communicate via a network infrastructure, and as VDM-RT is object oriented all interaction occurs between instances of classes, constituting the minimal system components. The invocation flow between objects in a DES architecture is illustrated in Figure 1: the objects *a* and *b* are deployed to the same node (*cpu1*), while *c* is deployed to another (*cpu2*). Furthermore, if *a* invokes *b* it is a *local call*, while interacting with *c* is a *remote call*. Hence such call distinction is introduced as an architectural feature, and becomes important since the latter is more involved due to network communication. Moreover, note that in VDM-RT functionality of an instance is captured either by **operations** or **functions**, where only the former can access and change state. For the remainder of this paper the term “method” is used to describe encapsulated functionality in both a VDM model and a C code context.

Fig. 1 DES architecture: Example of local and remote communication between the objects *a*, *b* and *c*

¹ Experiments with describing dynamically reconfigured distributed systems have also been carried out [34].

Fig. 2 Alternative system architectures for deployment of three objects (squares represents nodes while lines refer to bus connections)

In regards to architecture design VDM-RT is both *location* as well as *distribution* transparent. *Location transparency* for objects is described by the communication paradigm Remote Method Invocation (RMI), which proscribes that local and remote invocations are transparent at the application level. Several technologies support the RMI paradigm, and provide the proper architecture routing depending on the object location, such as Common Object Request Broker Architecture (CORBA) [35], Java RMI [44] and Distributed Component Object Model (DCOM) [21]. *Distribution transparency* follows when the additional complexities of distribution are hidden, which allows designers to consider a DES holistically. Consequently the system-wide functionality is independent of the system architecture, e.g. deployment of objects. Hence from a design perspective an RMI technology is not distribution transparent, since an object needs to obtain a reference of a remote instance, before being able to invoke it. As a consequence only having location transparency requires creation of a new architectural set up for the distributed objects, even with equivalent system-wide functionality. For example, consider the deployment shown in Figure 1, object *a* might invoke the objects *b* and *c*. In this case, it is possible to construct four distinct architectures as shown in Figure 2, where Figure 1 depicts the second option. Within this context, in [49] the authors present an analysis and exploration of alternative architectures based similar deployment possibilities.

In VDM-RT both facets of transparency are supported by seamlessly separating the functionality to normal classes as in VDM++, while having a special class, defined as a **system** definition, containing the architecture description. Figure 3 depicts relationships of VDM-RT model components: an object is a class instance, which can be deployed to a CPU (a computing node), that might be linked to other CPUs using a BUS construct (a network channel). Additionally, the node and network constructs may only be used inside the **system** definition as it captures the distribution. Hence all objects from the SUD are deployed to specific nodes explicitly, while other parts of the VDM-RT model do not need to know about the overall architecture. For example Listing 1 illustrates how the DES shown in Figure 3 is modelled inside the **system** definition.

```

system Sys

instance variables
  -- Objects
  public static b = new B();
  public static c = new C();
  public static a = new A(b, c);

  -- CPUs

```

Fig. 3 Overview of relationships of VDM-RT model components

```

public static cpu1 : CPU := new CPU(...);
public static cpu2 : CPU := new CPU(...);

-- BUSses
public static bus : BUS := new BUS (... , {cpu1, cpu2});

operations
public System : () ==> System
(
    cpu1.deploy(a);
    cpu1.deploy(b);
    cpu2.deploy(c);
);

end Sys

```

Listing 1 Modelling a DES as shown in Figure 3 in VDM-RT, where an ellipsis indicates omitted parts

Once a VDM-RT model of a DES has been constructed, various validation techniques can be applied. As an example it can be validate by means of simulation, where the VDM interpreter [30] makes use of the object deployment inside the **system** definition in order to determine when communication between nodes is required. This supports a holistic approach towards architecture validation and exploration independently from the overall functionality. For example trying to access an object without having a communication channel yields an error by the interpreter, providing early design feedback when modelling a distributed architecture. Nevertheless, in order to simulate a SUD model, the context in which it appears, i.e. its environment, also needs to be captured. Hence since we are only interested in the precise performance of the SUD, VDM-RT includes a concept of a virtual CPU and a virtual BUS that both are infinitely fast. Consequently for simulation purposes all objects not deployed to a computing node, automatically get placed on the virtual CPU connected to all other computing nodes via the virtual BUS. Finally, since we are targeting an implementation of a DES, it is also worth noting that communication in a VDM-RT context is considered at an abstract level, i.e. it is idealised so no messages are lost. This aspect requires consideration during model realisation, and is further discussed in Section 4.

3 VDM-RT Modelling Guidelines for Automatic Code Generation

A systematic process for gradually introducing complexity towards a distributed system using VDM-RT modelling is described by Larsen et al. [25]. However, this process is not

sufficiently constrained to enable automatic code generation for a DES. For this reason, guidelines are proposed in order to constrain the modelling within VDM-RT such that realisation can be automated. These guidelines extend upon the principles described in our previous work [19], which involved a code generator supporting distributed Java based system using Java RMI as the underlying technology.

The following modelling guidelines can be divided into two distinct parts, namely an architecture *structure* as well as a notation *subset* definition. The *structure* is solely for the **system** definition, as it attempts to standardise the VDM-RT architecture model. On the other hand, the *subset* is imposed on the architecture definition together with the normal classes of a VDM-RT model, and is inspired by combining established practices for both object oriented design in embedded and distributed systems. Note that we did not change the semantics of VDM-RT. The presented guidelines are based on our experiences when analysing various VDM-RT models, and researching how they can be automatically generated. Hence these guidelines present parts that usually have not been considered when designing a VDM-RT model, but can make the implementation afterwards more challenging.

From the perspective of a typical embedded control system, consisting of controllers that use sensors and actuators, Listing 2 presents the *structure* to follow when defining an architecture in VDM-RT: The first part consists of instantiating actuator and sensor objects which controller objects can obtain a reference to as part of their creation. Subsequently instances in the **system** definition are referred to as distributed objects, since they capture the object deployment architecture of the SUD. Next, the system architecture is modelled using the CPU and BUS constructs. Finally, the constructor of the **system** definition contains the deployment of objects to nodes. Inside this constructor we allow invocation of methods on distributed objects, for set up purposes, as long as these do not invoke other distributed objects, in order to avoid network communication during initialisation of the DES. Furthermore, it enables to automate the DES initialisation by the code generator, as described in Section 4 below. Note that network communication inside an object might be initiated in two ways: (1) all distributed objects are accessible directly in the entire model (e.g. `Sys.controller.method()`), since they are declared as **public static**; (2) using the object references that are passed during instantiation as shown in Listing 2 for controllers.

```
system Sys

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

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

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

```

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

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

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

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

end Sys

```

Listing 2 Structure for the target **system** definition in VDM-RT for code generation

Together with the structure shown in Listing 2, the following architectural modelling guidelines are presented in order to be exploited by the code generator:

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

This enables the code generator to know the exact location of all distributed objects, which it can exploit during analysis in order to generate architectural support for each node, as further described in Section 4.

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

This ensures that all instances inside the **system** must be deployed to explicit nodes as created by the designer. Consequently no objects containing SUD functionality are allowed to be placed on the virtual CPU, which must only contain instances that are necessary to simulate the SUD as part of model validation. This way it supports having validation tests during development that are not generated by the code generator, as they are not part of intended functionality.

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

It is possible to design an architecture where a specific remote object can be reached by multiple network channels from the same device. However, this introduces additional non-determinism in the implementation as the chosen network channel might change for the same remote invocations. In fact, the VDM interpreter ensures determinism in such a scenario by picking the same bus every time. For this reason, this guideline aligns the model interpretation with the implementation.

As a means to align the implementation of a VDM-RT model with certain design characteristics of a distributed system, some additional modelling guidelines are imposed for the normal classes below:

(1) Static methods that are accessible DES-wide must not update state components:

Globally accessible methods can be invoked without the need of an object instance, e.g. accessed directly by their name. If such methods contain side-effects, e.g. changed data of distributed objects, it is not clear at which computing node the change shall take place.

(2) Instance variables (i.e. object state) that are accessible DES-wide are disallowed:

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

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

In principle, the code generator could be updated for supporting direct field access. However, we strive to provide general principles that can be used for other technologies as well, and this enforces the rule adapted by other distributed technologies, such as CORBA and Java RMI, and as stated in [10]: *“Some languages, for example Java and C++, allow programmers to define objects whose instance variables can be accessed directly. But for use in a distributed object system, an object’s data should be accessible only via its methods.”*

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

For collections in a VDM-RT model the size cannot be decided statically before runtime. However, working with systems that have limited memory available as well as limited network communication bandwidth, it usually is not desirable to have unknown sizes of collections in the implementation. As a consequence, this rule enforces a range definition for all collections using a specific modelling pattern, as discussed further in Section 4.5. This supports possible validation of the device having sufficient memory. In addition, this enables efficient serialisation of data that needs to be shared between computing nodes on a network.

4 Code Generation

4.1 Overview

Overall the process of the code generator is taking a VDM-RT model as input, in order to output an individual executable for each embedded device in a DES model, which corresponds to a CPU that is defined inside the **system** definition. Even though this paper focuses on architecture support in accordance with the VDM-RT model, a node can be divided into three facets, each needing consideration as part of code generation: *functionality*, *initialisation*, and *architecture*. The *functionality* facet refers to the application level functionality running on each node. The *initialisation* facet refers to the allocation of resources and establishment of connections between all distributed objects before functional execution. The *architectural* facet refers to distribution awareness of a node in accordance with deployment of the objects.

Figure 4 shows the entire code generation process as well as illustrating its relation to a VDM-RT model. Furthermore, it indicates tools involved (names on top of arrows) and is further discussed below, while “Uses” refers to how the different parts of C code are linked. Note the overall approach: the code generator emits C code based on functionality and model architecture, while the developer is allowed to add proprietary network drivers associated with each bus as captured inside the **system** definition. Such an approach not only enables developers to use their own drivers, but furthermore to consider non-idealised communication. So while the model level network communication is idealised, with this approach a developer can handle these low-level considerations as part of a network driver in

a DES. Additionally, the serialisation of data is provided as an external part, even though the code generator could integrate this within the generated model code. However, this enables the serialisation block to be implemented with respect to user needs, as well as providing flexibility in a resource constraint setting. Consequently it provides a trade-off between code generation support for distribution described in a model, and using specific technology for an implementation. Subsequently, Section 4.2 introduces the main contributions from [4] needed to understand the contributions of this paper, which are presented in the remaining subsections. Hence these contributions present how the code generator is extended in order to support generating code for DESs, when following the guidelines presented above. In general, the division between functionality and distribution in VDM-RT enables to have systematic code generation extensions [20].

Fig. 4 Distributed Code Generator Architecture and tools involved

4.2 Functionality

Generating the functionality corresponds to emitting code from the normal classes of a VDM-RT model, as indicated in Figure 4. This subsection presents the contributions from our previous work [4], which is needed in order to understand the contributions presented below. A detailed description of this functional code generation process that targets embedded systems is provided in [4]. However, the design of the functional code generation affects the support for distribution, and hence relevant parts are discussed below.

The emitted implementation consist of the generated C code from the VDM-RT model together with a runtime library, which does not change. In particular, this runtime library provides a fundamental data structure supporting all the VDM-RT data types, and is called `TypedValue`. The definition of `TypedValue` is provided in Listing 3, and a pointer to `TypedValue` is defined as `TVP`, which 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;           // 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;
};

```

Listing 3 Fundamental code generator data types

All elements of TVP only store a relevant values in one of the fields, and to minimize memory usage, the value storage mechanism is a C union, since it only takes up as much space as the largest field. Additionally, the basic types **int**, **char**, etc. are stored as values in fields. Methods that construct such basic values are provided, for example “TVP newInt(**int**)” and “TVP newBool(**bool**)”. Structured VDM types (like sets and sequences) are stored as references (**void***ptr), because of their variable size. So all the methods defined in the VDM language manual [29] on both basic and structured types are implemented one-to-one.

The object orientation features of VDM-RT are implemented in the runtime library using C **structs**. In this context, a class is represented by a **struct**, where its fields represent the fields of that class. Furthermore, access to methods is implemented as a virtual table, i.e. implemented as functionality associated with the corresponding struct. Hence all objects are represented as instances of TVP as well. For this reason a method call is achieved as shown in Listing 4, in this case on object a of class A (see VDM-RT model in Listing 1), using the CALL_FUNC macro which invokes the step method (name mangled to CLASS_A__Z4stepEV) by using the virtual table. Furthermore, the relevance of the CALL_FUNC macro for call distinction with respect to the architecture is discussed below.

```
TVP ret = CALL_FUNC(A, A, g_System_a, CLASS_A__Z4stepEV);
```

Listing 4 Invocation of step operation on object g_System_a

Generally, VDM-RT is very suitable for modelling DES, but some low-level needs of embedded development, can be achieved better in C code, such as direct memory manipulation or creating optimised code for a specific platform. For this reason an important aspect is supporting linkage against native code. Such functionality is supported by this code generator by specifying a method as **is not yet specified** in VDM-RT. Then after the translation to C code, the code generator provides an interface which the developer can use for linking a native C code function with the generated code during compilation. Furthermore, such an approach might also be used when needing to link against assembly code. In

general, this also enables linkage against C code for various drivers when targeting a specific platform.

4.3 Initialisation

In a VDM-RT model during the initialisation process the resources for each node are allocated, and with respect to architecture it especially refers to allocation of distributed objects. Hence the code generator has to support initialisation of a DES, before functional execution for each embedded device. Due to the imposed VDM-RT modelling guidelines it can exploit knowledge of the static deployment for establishing awareness of local and remote objects inside the **system** definition. During the code generation process each distributed object is assigned a unique number ID, ensuring DES-wide identification for all. Then with respect to each node local objects are instantiated as normal instances, e.g. TVP type `VDM_CLASS`, while remote objects become of a remote type instantiated with their unique number. In this way the common TVP type enables having a dispatch macro to wrap the `CALL_FUNC` macro in order to decide whether to dispatch a call as local or remote, as shown in Listing 5. In addition, note that a remote object is passed to the generic method called `send_bus`. This method is generated based on the architecture, and is further discussed in Section 4.4. The code generator emits an initialisation method for each node, in order to set up both its local and remote objects. Consider, as an example, the DES model shown in Listing 1: the object `a` needs to obtain a reference to the remote instance `c` *before* its functional execution. Furthermore, if the node is called `cpu1`, the code generator emits an initialisation method for it as `cpu1_init()`. This method, moreover, gets the method invocations of its local objects as defined inside the **system** constructor (see Listing 2).

```
#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 5 `DIST_CALL` macro which dispatches between local and remote invocations

In general the presented initialisation approach allows to set up a global reference to an object independently of its location, and additionally without the need to obtain an actual memory reference on the given node during initialisation. Consequently this exploits both the location and distribution transparency features provided in the VDM-RT model more directly. However, another way to achieve distinction between local and remote objects, would be to extend a location transparent RMI technology, with an initialisation algorithm which sets up and obtains the correct references. Such an approach has been applied in [19] for the VDM-RT to Java code generator using the Java RMI technology, and for this C code generator the CORBA middleware could provide a similar solution. Nevertheless, using an existing RMI technology requires a more involved initialisation functionality, as it requires exchange of references during system set up. For example in [19] where Java RMI is used, an initialisation algorithm is generated under the assumption of an existing global database for exchange of references. Having an arbitrary network structure as in a DES usually, and no such database, makes it even more involved. When targeting embedded systems, in addition, a specific middleware technology would limit the generated code in two ways: first, it might not easily allow use of proprietary bus drivers as this technology relies on its own infrastructure. Second, it would introduce large performance as well as memory overhead, since it might be a general purpose technology not particularly suited for embedded devices.

Hence this provides additional motivation to gain full control of RMI, and for implementing it as an extension of the runtime library.

4.4 Architectural Awareness in Remote Call Dispatching

In this part we discuss how the generic remote invocation method `send_bus`, described in Section 4.3, is forwarded in accordance to the DES model deployment. Hence while the local call is handled as a regular call, the remote is routed to the generic `send_bus` method, which is unique for each node. This method needs to forward the remote call to the correct bus in accordance with the VDM-RT system architecture, e.g. calling the user defined driver. In order to support such an aspect, the code generator needs both to extract relevant information from the **system** definition together with an algorithm to emit architecture based code. The routing is achieved using the architecture defined by the CPU and BUS constructs, together with the deployment of objects. Listing 6 illustrates the algorithm of dispatching with respect to the analysis; internally the code generator analyses various relations: set of deployed objects to a node (`cpuToObjs`), set of connected BUSses to each node (`busToCpus`) and set of nodes connected to each BUS (`cpuToBusses`). With respect to a computation node, the `generate` method emits a switch functionality for each bus based on a set of objects that can be reached by the given bus. In fact, this ensures that the `send_bus` forwards the remote call to a method with the exact name as in the model together with a standardised parameter list, which enables the user-defined low-level communication driver to be linked.

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

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

Listing 6 Pseudo code for algorithm which generates `send_bus` functionality

Another facet of architectural awareness is the ability to handle an incoming remote invocation, i.e. when the `send_bus` initiates a call from another CPU. The code generator supports this by emitting the method `getRes`, having a standardised interface as well, which ensures correct dispatching to the local object resource. The `getRes` method is generated by knowledge about locally deployed objects (`cpuToObjs`) to generate dispatching based on the unique IDs of distributed objects. Note that since both `send_bus` and `getRes` are related to the architecture, they are generated as part of the emission of the **system** definition. The signatures of these methods are shown in Listing 7.

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

Listing 7 Signatures of `send_bus` and `getRes` methods

4.5 Serialisation

VDM-RT supports arbitrary structured types, such as records and sets which moreover might be nested. In general it is challenging to support data transfer across all possible embedded

platforms. One way to achieve this is by use of a generic message representation (like XML or JSON), but these formats require generic parsers that introduce additional memory and CPU overhead. Another approach is using serialisation functionality tailored for embedded systems, which enables generation of type specific encoding functionality. One such tool is the Abstract Syntax Notation One (ASN.1) translation tool, referred to as `asn2c` in Figure 4, developed by the European Space Agency (ESA) targeting serialisation for embedded systems [32]. ASN.1 is a data description language, and `asn2c` generates encoding/decoding capabilities for each type in C code.

In order to use ASN.1 as part of data marshalling, as shown in Figure 4, it requires two distinct capabilities: *generate an ASN.1 file from types defined in a VDM-RT model* (`vdm2asn`) and *conversion between TVP and types generated from an ASN.1 file* (`asn2tvp`). The latter functionality is based on our previous work described in [12]. How these two tools together with `asn2c` support serialisation is illustrated in Figure 4: All structured types from a VDM-RT model are extracted and an ASN.1 file generated by `vdm2asn`. Next, based on this file `asn2c` generates bespoke encoders/decoders, while `asn2tvp` generates conversion method between TVP and structured types in C code generated by `asn2c`. Hence the generated files can be used for serialising purposes of structured VDM types. Note that this generates serialisation capabilities based on the VDM types created by the user.

The serialisation generation can be illustrated by an example: consider the VDM record as defined in Listing 8. First, `vdm2asn` generates an ASN.1 file as shown in Listing 9. Next, `asn2c` emits basic encoders, while `asn2tvp` generates converters as shown in Listing 10. Overall collection types are required to be restricted to a range of values by following an invariant pattern. As a consequence, this rule enforces a range definition for all collections using a specific modelling pattern, using the VDM-RT invariant construct (`inv`). Hence collection types are required to be restricted to a range of values by following an invariant pattern: “`inv x >= 0 and y <= 20`”, where `x`, `y` are numerical values, and `y > x`. This patterns can be used on both collections (e.g. sets) and to limit numerical values, where the latter is exemplified in Listing 8 on the integer type `MInt`. These restrictions are imposed by the constraints introduced in Section 3, which enables the size to be known and TVPs to be serialised and sent across the busses.

```
...
public MInt = int
inv x ==
    x >= 0 and x <= 20;

public static Wtdata ::
    level   : MInt
    valve   : bool;
...
```

Listing 8 Modelling a range limited integer (`MInt`) and a record (`Wtdata`) in VDM-RT

```
...
MInt ::= INTEGER (0 .. 20)

Wtdata ::= SEQUENCE {
    level MInt,
    valve BOOLEAN
}
```

...

Listing 9 Generated ASN1 file from Listing 8

```

flag asn1SccWTdata_Encode(const asn1SccWTdata* pVal,
    BitStream* pBitStrm, int* pErrCode, flag bCheckConstraints);

flag asn1SccWTdata_Decode(asn1SccWTdata* pVal, BitStream* pBitStrm,
    int* pErrCode);

void Convert_WTdata_from_VDM_to_ASN1SCC(asn1SccWTdata *ptrASN1SCC,
    TVP *VDM);

void Convert_WTdata_from_ASN1SCC_to_VDM(TVP *ptrVDM,
    const asn1SccWTdata *ptrASN1SCC);

```

Listing 10 Interface of bespoke encoders and converter methods based on the ASN1 file in Listing 9

5 Industrial Case Study: Distributed Railway Interlocking

This section presents an industrial case study within the railways domain from the company ClearSy, and illustrates how the code generation tool has supported the development efforts. The main focus is on the development approach using VDM-RT modelling and code generation. First, Section 5.1 introduces the case together with development challenges, which involves the evolution an existing product to a distributed setting. Section 5.2, afterwards, shows how this is achieved using VDM-RT. Additionally, the implementation considerations when using the code generator as well as related validation using co-simulation are discussed in Section 5.3 and Section 5.4, respectively. Finally, Section 5.5 discusses challenges and practical aspects through the whole development process.

5.1 Overview

This case study considers an existing tramway station, and Figure 5 presents the railway layout: trains enter at the points V1, Q2 and Q3, and afterwards request a route to be allocated by issuing a telecommand (TC) (green dots). Furthermore, the access to the interlocking system is controlled by the signals S11, S28 and S48, where red and green colours are used for wait and go signals, respectively. When a train issues a TC, the interlocking system establishes a route using switches (SWs), and gives the train green light to move to the next segment indicated by a unique CDV number. In order to guarantee that collision freedom, no other train is allowed to proceed, while the current train passing that segment. Overall such an interlocking system is in charge of a complete line, where it computes the status of actuators (switches) based on signalling safety rules that are encoded as “boolean equations” as shown in Figure 6, usually managing around 180,000 equations recalculated several times per second. Moreover, these equations compute the commands to be issued to track-side devices: they encode the safety behaviour that enable trains to move from one position to another through routes that are allocated and then released.

The *existing* embedded system is based on central interlocking principles, captured as Ladder logic, where all decision are made globally, which uses a powerful PLC (Pro-

Fig. 5 Partial scheme plan of a train line as seen from the sky, including track circuits, switches and traffic lights

Fig. 6 Boolean equations that lead the signalling system

grammable Logic Controller) for safety equations and kilometres of wires to connect individually each sensors and actuators to the PLC. Hence the PLC centrally provides safety guarantees, such as collision freedom for trains. Furthermore, it makes use of five mechanical safety relays that externalize the state of a route and allow redundancy between software logic and electronic circuits.

The *new* embedded system is based on a *distributed interlocking system*, i.e. a DES. Such an approach might help to reduce the global length of the connections and therefore the cost and the risk of failure. Within this context, Figure 5 depicts a division into five zones (highlighted in blue), namely ZQ2, ZQ3, ZP, ZV1 and ZV2, where each zone is controlled by an interlocking module (i.e. an embedded device). Furthermore, each module would require less computation power than a central interlocking as fewer local devices have to be accounted and local decisions, that only involve local device could be taken faster and would result in potentially faster train transfers. However, such distribution has to be carefully designed so that each module can ensure the required local safety properties, and communica-

tions have to be introduced so that the global safety properties are guaranteed. As a matter of fact, nine mechanical relays are now required for the distributed version since most routes are distributed over several modules and all of them need to protect the different routes with different relays.

5.2 VDM-RT modelling

The distributed interlocking system is modelled in VDM-RT. First, the centralised interlocking system given as Ladder logic is translated to a VDM-RT model, and without architecture considerations. This centralised model is validated by means of co-simulation in an FMI setting (see Section 1) with a physical model of the environment from another tool, such as train movement and physical relays. Hence this ensure that the initial model (without distribution) exhibits valid behaviour, i.e. equal to the Ladder logic.

Afterwards, the architecture is captured following the guidelines described in Section 3. In this regard, the holistic view of a DES supported by VDM-RT, enables an easier transfer to a distributed setting, as less implementation details need consideration. For the distributed model, especially, each interlocking module indicated in Figure 5, is captured in form of a class. Furthermore, common functionality between nodes is shared by inheritance, since some of the equations are kept or duplicated between the different modules, while others have been created specifically for a given embedded device. Compared, the centralised interlocking model contains 180 equations, while the distributed one has 247 equations shared between the five modules (85 for ZV1, 49 for ZV2, 37 for ZQ2 and ZQ3 and 39 for ZP). The final architecture is shown in Listing 11, where the **system** definition supported high level exploration between alternatives. Finally, this DES model in VDM-RT is co-simulated and validated, as the centralised VDM-RT model was.

```

system Interlock
instance variables

public static zp : ZP := new ZP();
public static zq2 : ZQ2 := new ZQ2();
public static zq3 : ZQ3 := new ZQ3();
public static zv1 : ZV1 := new ZV1();
public static zv2 : ZV2 := new ZV2();

CPU1: CPU := new CPU(<FP>, 180E6);
CPU2: CPU := new CPU(<FP>, 180E6);
CPU3: CPU := new CPU(<FP>, 180E6);
CPU4: CPU := new CPU(<FP>, 180E6);
CPU5: CPU := new CPU(<FP>, 180E6);

UART1 : BUS := new BUS (<FCFS>, 1e6, {CPU1, CPU3});
UART2 : BUS := new BUS (<FCFS>, 1e6, {CPU2, CPU3});
UART3 : BUS := new BUS (<FCFS>, 1e6, {CPU2, CPU4});
UART4 : BUS := new BUS (<FCFS>, 1e6, {CPU4, CPU5});
UART5 : BUS := new BUS (<FCFS>, 1e6, {CPU5, CPU1});

operations

public Interlock: () ==> Interlock
Interlock() == (
-- Initialize neighbour
  Interlock`zp.setZone(<ZP>, Interlock`zq3, Interlock`zv2);

```

```

Interlock`zq3.setZone(<ZQ3>, Interlock`zq2, Interlock`zp);
Interlock`zq2.setZone(<ZQ2>, Interlock`zv1, Interlock`zq3);
Interlock`zv1.setZone(<ZV1>, Interlock`zv2, Interlock`zq2);
Interlock`zv2.setZone(<ZV2>, Interlock`zp, Interlock`zv1);
-- Deployment
CPU1.deploy(zp);
CPU2.deploy(zq2);
CPU3.deploy(zq3);
CPU4.deploy(zv1);
CPU5.deploy(zv2);
end Interlock

```

Listing 11 Architecture of the distributed interlocking system in VDM-RT

5.3 Implementation

Subsequently, we focus on the final prototype implementation of our interlocking system together with its deployment to a hardware set up. In this regard, Figure 7 shows a prototype realisation of the architecture shown in Listing 11: six independent modules, one for Ethernet communication (the largest board in the middle) for co-simulation, and five for the devices (PIC32MX micro-controller) of the interlocking connected by Universal Asynchronous Receiver-Transmitters (UARTs). Additionally each device is connected to physical relays, where each is able to lock three relays for route reservation, manipulate two switches, three track circuits and one pair of signal lights.

Fig. 7 Overview of the Hardware-in-the-Loop set up, and showing the state of the system when the Q2V2 scenario has been established by issuing telecommand TC2

Fig. 8 The state-machine of each embedded device in our case study

In order to obtain the final implementation, a developer has to provide *scheduling for each node* as well as *network drivers*, as discussed above and indicated in Figure 4. The *scheduling for each node* has to be light-weight due to resource constraints on the nodes. Hence a simple scheduling mechanism responsible for both running a periodic task and handle incoming invocations has been implemented, as shown in Figure 8: First, the node is initialised in state `INIT` (see Section 4.3). After initialisation it enters the `IDLE` state, if there is an incoming remote indication (e.g. data on the bus) the `HandleRemoteInvocation` state is entered and the invocation is handled using generated the `getRes` method described in Section 4.4, or it enters the `PeriodicTask` state, i.e. running its periodic method from the VDM-RT model.

For adding *proprietary network drivers*, we use existing safety critical drivers developed by ClearSy. In this regard, the generated empty bus functions are replaced by this driver implementation. So it provides easy interfacing with the model code. Moreover, it is now possible to handle non-idealised communication within the driver implementation, while the generated code assumes idealised communication. As an example, Cyclic Redundancy Check (CRC) support can be added inside the low-level implementation, as a means to ensure valid communication between nodes.

5.4 Validation by co-simulation

The Hardware-in-the-Loop (HiL) set up, as shown in Figure 7, can be run in a co-simulation setting as depicted in Figure 9. In particular for illustration purposes, we focus on a Q2V2 scenario, which can be described as follows by looking at Figure 5: A train arrives at Q2 (CDV.Q2), the TC2 (remote command) is set to high to signal to establish a route from Q2 to V2. Furthermore, the establishment of this route requires reservation of zones ZQ2, ZP and ZV2. Then, once the route has been established, the authorization light on Q2 (S28) turns green. Finally, the train moves to the next segment (CDV.28), and the authorization light on Q2 turns red in order to prevent other trains from entering the segment.

Figure 10a shows running the generated C code in a Software-in-the-Loop (SiL) co-simulation setting, which is supported by the FMI exporting capabilities of our C code generator [4]. Basically, the SiL simulation corresponds to executing the entire VDM-RT model generated as a single C code executable, where each CPU has its own thread of execution. Hence showing that the SiL behaves correctly ensures that the generated C code is behaving

Fig. 9 Overview of HiL co-simulation approach

correctly, before considering deployment and using hardware bus drivers. Finally, the HiL simulation corresponds to running the distribution support of the code generator, and obtaining an executable for each interlocking module (a CPU in the VDM-RT model). Each of these executables can then be deployed to the PIC32MX440F256H controller. Furthermore, the corresponding HiL simulation is shown in Figure 10b. For our case study the focus is on correct order between signals, and as Figure 10a and Figure 10b show, they exhibit equal safety critical behaviour in this regard: The correct switch changes its position (sub-plot 3) due to a change signal (sub-plot 2), and *afterwards* the light on Q2 changes to green (sub-plot). Furthermore, once the train moves into the next track segment (sub-plot 1) the light goes to red again (sub-plot 4). The hardware shown in Figure 7, shows the state of the DES once the Q2V2 scenario has been established by issuing telecommand TC2, where only the light on the ZQ2 interlocking module is green.

5.5 Discussion

In general our modelling notation together with code generation provides useful support for evolving the SUD to a distributed setting. The holistic modelling approach in a VDM-RT model enables easier consideration of different architectures, as well as faster prototyping. Moreover, the architecture can be updated in VDM-RT and the code can be regenerated, while the user implemented parts do not need to change. So an important facet is the possibility to only integrate platform specific code once, e.g. node run-time together with propriety network drivers. Validation is enabled by means of co-simulation for the different representations of the SUD. Hence from a development perspective we can consider the DES as a holistic system, while generating new models, without updating the platform specific code again. However, there are challenges related to implementing the communication drivers such that they preserve the semantics of a VDM-RT model. While such a model allows defining abstract protocols, such as First-Come-First-Serve or Fixed-Priority, the developer has to ensure this in the implementation.

A key aspect, when discussing a distributed system, is inference of communication channels. In VDM-RT it is specified which objects communicate with each other using the **system** definition class. Hence this provides verifiers with a holistic view of the distributed system during specification. In this regard, the abstraction is on purpose chosen to be with

(a)

(b)

Fig. 10 Software-in-the-Loop (a) and Hardware-in-the-Loop (b) co-simulation for train scenario Q2V2: showing the initial establishment of the Q2V2 route, and train passing the authorization light

CPUs (and each CPU can have multiple processes modelled as threads) and BUSses that connect such CPUs. Furthermore, inside the processes it is not referred to a specific channel that is used for communication to an object that is placed on a different CPU. This supports having only one central place to change the overall topology of the system (the **system** class). Since channels are not named explicitly, the functionality inside instances of classes is agnostic about whether the functionality is located at the same CPU (as a monolithic solution) or inside a distributed setting. On the other hand, if it was referred to the channels explicitly inside the processes, and the topology needs to change, it would require updating all instances referring to such a channel. Nevertheless, the drawback of this implicit approach in VDM-RT is that there is no clean calculus, which can be applied to prove certain properties by applying algebraic laws.

Another aspect, unrelated to distribution, is the execution time of the generated C code, since it is not optimised for logical equations calculations. Hence for systems having time requirements, a developer needs to implement these critical parts manually. On the other hand, for systems comparable to ours where DES-wide dependability is the objective as well as re-usability for future products, the presented code generation support can aid overall development efforts.

Generally, this case study provided an example of a DES where both the number of nodes as well as links between them are not changed during run-time. However, other systems (such as sensor networks) have characteristics where both aspects are dynamic during run-time. VDM-RT cannot model such systems, but research has been carried out for supporting such dynamic characteristics of a DES in VDM-RT [34]. For this reason such dynamic DES could be considered for future work of code generation support, which might include translating towards specific technologies that support these needs. In this context, the contributions of this paper can be used as the basis for such future research.

Another important aspect when designing a DES, finally, is energy consumption. In the interlocking case study energy is not a concern, but for other systems having a low energy consumption is one of the most important design parameters. In [23] and [22] the authors present how VDM-RT can be applied when considering energy during the design of an embedded system. Hence future work could be to combine the code generation of a VDM-RT model with such energy research. Furthermore, in [43] they present the Atom LEAP platform which focuses on energy efficient embedded systems. Hence, as an example, code generation towards specifically the Atom LEAP platform could enable to include energy consideration during the generation process. This would require further research for both VDM-RT as well as extending the code generation capabilities with such concerns in mind.

6 Related Work

Various approaches discuss aiding development efforts by means of code generation support for distribution, showing it can be a worthwhile endeavour. Below we divide code generation towards distributed embedded systems into four areas: (1) general model based design approaches, (2) targeting specific network drivers or technology, (3) general support for communication synthesis and (4) integrated tools. In particular, the final category can show a potential of generating VDM-RT models targeting their infrastructure. Some of the work covers several areas, but is grouped into the most suited one. However, note for the following related work the majority focus on code generation based on components directly connected via their ports, while our work targets object oriented DES in general.

Within area (1), in [2] they consider low-level state-machines based code generation. Furthermore, several approaches [3, 7, 8] present general model-based approaches, such as Unified Modelling Language (UML), combining deployment of functional code with targeting an architecture by having platform specific models. In comparison with area (1), our contributions provide a different trade-off between generating platform agnostic C code from a model, while still allowing to interface with both native code as well as various network drivers. For area (2), in [24] and [1] the authors consider targeting specific protocols, respectively FlexRay and Controller Area Network (CAN). While VDM-RT models define abstract protocols mentioned above, these approaches might support additional analysis since the targeted hardware is known. On the other hand, in [16] they present a more high-level approach targeting CORBA middleware, while [42] targets an AUTomotive Open System ARchitecture (AUTOSAR) infrastructure. So the contributions in area (2) can be used for supporting specific technologies in our code generator, hence provide possible future work. As part of area (3), the authors in [36, 13, 31] introduce approaches for targeting arbitrary architectures, allowing to use network specific drives. Their approach is similar to our approach, considering architectures, while their models are captured in a component-based setting. However, in our work the focus is on an object oriented design which might be more suitable for certain types of DESs.

Area (4) concerns integrated ways of supporting modelling together with code generation. The tools Distributed Object Model Environment (DOME) [38, 39] and Programming Temporally Integrated Distributed Embedded Systems (PTides) [11] support such an approach. DOME is based on an approach where objects deployed to different nodes can be connected with each other. Especially, DOME enables a modelling approach similar to VDM-RT, and supports the implementation with a runtime environment, which can include real-time constraints. In comparison to DOME, our contributions attempt to support a different way of combining automatically generated code together with native code and network drivers in an object oriented context. PTides, furthermore, supports modelling in particular focusing on time aspects in a control system, using component-based design, while ensuring timing aspect in the implementation by having their own PTides runtime. Hence compared to our code generator, PTides uses components and can address timing aspect, where the latter might provide possible future work for our code generator. The tool chain “The ASSERT Set of Tools for Engineering” (TASTE) [37] adopts a more general approach for development of DES. Their approach is based on creating architectures of components in the Architecture Analysis and Design Language (AADL), which are implemented in different notations such as Matlab/Simulink and Spark Ada. In order to connect these heterogeneous models as well as across distributed architectures, they use the ASN.1 notation as glue code, as mentioned Section 4.5. However, TASTE currently does not fully provide integrated FMI co-simulation capabilities as VDM-RT.

Finally, in this paper we focus on code generation of resource constraint distributed embedded nodes using an object oriented modelling language. Hence an important comparison can be made to the work presented in [46], where they present an object oriented notation for embedded systems with limited resources. The work in [46] targets much more resource limited nodes, in comparison to our approach: they neither have a run-time library nor use dynamic memory allocation. However, the usage of the run-time enables our code generator to support a larger subset of VDM-RT [4]. Generally, a key difference is the support for distributed systems. Nevertheless, the principles of modelling and generating code for DESs, as discussed in this paper, might be worth considering for supporting distribution in the work presented in [46], as it is based on objects as well.

7 Concluding Remarks

In this paper we presented a code generator for the VDM-RT notation together with modelling guidelines, which aid the development efforts of a DES, and contains two contributions. First, it presents how a VDM-RT model describing a DES can be supported by a code generation tool, which enables a developer to interface with hardware specific code. Second, it demonstrates how the tool has been applied on an industrial case study involving a distributed interlocking system from the railways domain. In particular for the case study it is paramount that the developer can use proprietary network drives, as well as considering non-idealised communication with respect to a specific network driver. The industrial case, additionally, illustrates both the holistic design and co-simulation capabilities, while the code generator supports model realisation in form of C code. However, a VDM-RT model defines abstract network protocols, and the user needs to ensure the correct semantics in the implementation of the bus drivers. Hence future work includes to support more concrete protocols, which afterwards could be realised by a code generator. For example a VDM-RT model could be integrated with a network simulator such as NS-3 [40] in order to simulate communication. In addition, ASN.1 support, discussed above, enables to know the data packet size in the implementation as it enforces range definitions, which could be passed back to the network simulator, creating an integrated approach. Furthermore, not related to distribution, the transfer of non-functional properties could be addressed, such as time, by targeting specific embedded platforms. Moreover, the current ASN.1 generation might be used to integrate the generated C code with code from external tools, similar to the TASTE tool chain. Finally, future work for code generation of VDM-RT models can include research of dynamic systems and energy consideration.

Acknowledgements We thank the anonymous reviewers for their constructive comments, which helped us to improve the manuscript. We would also like to thank Nick Battle for valuable feedback on the content of this paper. The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047. Tommaso Fabbri would also like to thank the European Space Agency for providing financial support for the ESA Summer of Code in Space enabled the ASN.1 work presented here.

References

1. Almohammad, A.: Rigorous code generation for distributed real-time embedded systems. Ph.D. thesis, Northumbria University (2013)
2. Anand, M., Kim, J., Lee, I.: Code generation from hybrid systems models for distributed embedded systems. In: Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on, pp. 166–173. IEEE (2005)
3. Bambagini, M., Natale, M.D.: A code generation framework for distributed real-time embedded systems. In: Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on, pp. 1–10. IEEE (2012)
4. Bandur, V., Tran-Jørgensen, P., Hasanagic, M., Lausdahl, K.: Code-generating VDM for embedded devices. In: The 15th Overture Workshop, Newcastle, England (2017)
5. Blochwitz, T.: Functional Mock-up Interface for Model Exchange and Co-Simulation. <https://www.fmi-standard.org/downloads> (2014)
6. Bowman, H., Derrick, J.: Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches. Cambridge University Press, New York, NY, USA (2011)
7. Chiozzi, G., Andolfato, L., Karban, R., Tejeda, A.: A UML profile for code generation of component based distributed systems. In: Proceedings of the 13th International Conference on Accelerator and Large Experimental Physics Control Systems (2011)

8. Ciccozzi, F., Cicchetti, A., Sjödin, M.: Full code generation from UML models for complex embedded systems. In: Second International Software Technology Exchange Workshop (STEW) November 28th, Kista, 2012 (2012)
9. Clarke, E.M., Wing, J.M.: Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys* **28**(4), 626–643 (1996)
10. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: *Distributed Systems: Concept and Design*. Addison-Wesley (2007)
11. Eidson, J.C., Lee, E.A., Matic, S., Seshia, S.A., Zou, J.: Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE* **100**(1), 45–59 (2012)
12. Fabbri, T., Verhoef, M., Bandur, V., Perrotin, M., Tsiodras, T., Larsen, P.G.: Towards integration of Overture into TASTE. In: P.G. Larsen, N. Plat, N. Battle (eds.) *The 14th Overture Workshop: Towards Analytical Tool Chains*, pp. 94–107. Aarhus University, Department of Engineering, Cyprus, Greece (2016). ECE-TR-28
13. Fan, Y.H., Wu, J.O., Wang, S.F.: Software synthesis of middleware for heterogeneous embedded systems. In: *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*, pp. 2084–2087. IEEE (2012)
14. Fich, F., Ruppert, E.: Hundreds of Impossibility Results for Distributed Computing. *Distributed Computing* pp. 121–163 (2004)
15. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: *Validated Designs for Object-oriented Systems*. Springer, New York (2005). DOI 10.1007/b138800. URL <http://overturetool.org/publications/books/vdoos/>
16. Fredj, M., Radermacher, A., Gerard, S., Terrier, F.: eC3M: Optimized model-based code generation for embedded distributed software systems. In: *New Technologies of Distributed Systems (NOTERE), 2010 10th Annual International Conference on*, pp. 279–284. IEEE (2010)
17. Gide, L., Koljonen, T., Lohstroh, J., ten Berg, A., Foster, A.: *Artemis strategic research agenda*. Artemis (2016)
18. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: a Survey. *ACM Comput. Surv.* **51**(3), 49:1–49:33 (2018)
19. Hasanagic, 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*, pp. 75–89. Center for Global Research in Advanced Software Science and Engineering, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan (2015). URL <http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf>. GRACE-TR-2015-06
20. Hasanagić, M., Tran-Jørgensen, P.W., Nilsson, R.S., Larsen, P.G.: Realization of distributed system models using code generation extensions. *Software: Practice and Experience* **49**(3), 478–497 (2019)
21. Horstmann, M., Kirtland, M.: DCOM architecture. Dcom technical article, Microsoft Corporation, Redmond, WA (1997). URL [http://msdn2.microsoft.com/en-us/library/ms809311\(d=printer\).aspx](http://msdn2.microsoft.com/en-us/library/ms809311(d=printer).aspx)
22. Isasa, J.A.E., Jørgensen, P.W., Ballegaard, C.: Modelling Energy Consumption in Embedded Systems with VDM-RT. In: *Proceedings of the 4th International ABZ conference*. (2014)
23. Isasa, J.A.E., Larsen, P.G.: Modelling Different CPU Power States in VDM-RT. In: *Proceedings of the 11th Overture Workshop 2013*, Aarhus University (2013)
24. Kang, D.I., Gerber, R., Golubchik, L., Hollingsworth, J.K., Saksena, M.: A software synthesis tool for distributed embedded system design. In: *ACM SIGPLAN Notices*, vol. 34, pp. 87–95. ACM (1999)
25. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM. *Intl. Journal of Software and Informatics* **3**(2-3) (2009)
26. 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: *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*. IEEE, Vienna, Austria (2016). [Http://ieeexplore.ieee.org/document/7496424/](http://ieeexplore.ieee.org/document/7496424/)
27. Larsen, P.G., Fitzgerald, J., Woodcock, J., Gamble, C., Payne, R., Pierce, K.: Features of integrated model-based co-modelling and co-simulation technology. In: Bernardeschi, Masci, Larsen (eds.) *1st Workshop on Formal Co-Simulation of Cyber-Physical Systems*. LNCS, Springer-Verlag, Trento, Italy (2017)
28. Larsen, P.G., Hansen, B.S., Brunn, H., Plat, N., Toetenel, H., Andrews, D.J., Dawes, J., Parkin, G., et al.: *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language* (1996)
29. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: *VDM-10 Language Manual*. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (2013)

30. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: S. Qin, Z. Qiu (eds.) Proceedings of the 13th international conference on Formal methods and software engineering, *Lecture Notes in Computer Science*, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-24559-6_14. URL <http://dl.acm.org/citation.cfm?id=2075089.2075107>. ISBN 978-3-642-24558-9
31. Lednicki, L., Carlson, J.: A framework for generation of inter-node communication in component-based distributed embedded systems. In: Emerging Technology and Factory Automation (ETFA), 2014 IEEE, pp. 1–8. IEEE (2014)
32. Mamais, G., Tsiodras, T., Lesens, D., Perrotin, M.: An ASN. 1 compiler for embedded/space systems. *Embedded Real Time Software and Systems ERTS* (2012)
33. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring Timing Properties Using VDM++ on an Industrial Application. In: J.C. Bicarregui, J.S. Fitzgerald (eds.) Proceedings of the Second VDM Workshop (2000). Available at www.vdmportal.org
34. Nielsen, C.B.: Dynamic Reconfiguration of Distributed Systems in VDM-RT. Master's thesis, Aarhus University (2010)
35. OMG: The Common Object Request Broker: Core Specification. (2002)
36. Ortega, R.B., Borriello, G.: Communication synthesis for distributed embedded systems. In: Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on, pp. 437–444. IEEE (1998)
37. Perrotin, M., Conquet, E., Delange, J., Schiele, A., Tsiodras, T.: TASTE: A Real-Time Software Engineering Tool-Chain Overview, Status, and Future. In: *SDL Forum*, pp. 26–37. Springer (2011)
38. Riedl, M., Diedrich, C., Naumann, F., Simon, R.: An object based approach for distributed automation. In: AFRICON, 2004. 7th AFRICON Conference in Africa, vol. 2, pp. 1253–1260. IEEE (2004)
39. Riedl, M., Zipper, H., Bangemann, T., Meier, M.: Distributed automation system supports process monitoring and control. In: *Industrial Electronics (ISIE), 2011 IEEE International Symposium on*, pp. 1711–1715. IEEE (2011)
40. Riley, G.F., Henderson, T.R.: *The ns-3 Network Simulator*, pp. 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
41. Satyanarayanan, M.: Pervasive computing: Vision and challenges. *IEEE Personal Communications* **8**(4), 10–17 (2001)
42. Schreiner, D., Schordan, M., Goschka, K.M.: Component based middleware-synthesis for AUTOSAR basic software. In: *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC'09. IEEE International Symposium on*, pp. 236–243. IEEE (2009)
43. Singh, D., Kaiser, W.J.: The atom leap platform for energy-efficient embedded computing (2010)
44. Sun: Java Remote Method Invocation Specification (2000)
45. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: The into-cps co-simulation framework. *Simulation Modelling Practice and Theory* **92**, 45 – 61 (2019). DOI <https://doi.org/10.1016/j.simpat.2018.12.005>. URL <http://www.sciencedirect.com/science/article/pii/S1569190X1830193X>
46. Titzer, B.L.: Virgil: Objects on the head of a pin. *ACM SIGPLAN Notices* **41**(10), 191–208 (2006)
47. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2009)
48. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: J. Misra, T. Nipkow, E. Sekerinski (eds.) *FM 2006: Formal Methods, Lecture Notes in Computer Science* 4085, pp. 147–162. Springer-Verlag (2006). DOI 10.1007/11813040_11
49. Wandeler, E., Thiele, L., Verhoef, M., Lieverse, P.: System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer* **8**(6), 649–667 (2006)
50. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. *ACM Computing Surveys* **41**(4), 1–36 (2009). DOI <http://doi.acm.org/10.1145/1592434.1592436>