SCHOOL OF
COMPUTING

The 15th Overture Workshop: New Capabilities and Applications for Model-based Systems Engineering

J. S. Fitzgerald, P. W. V. Tran-Jørgensen, and T. Oda (Editors)

TECHNICAL REPORT SERIES

No. CS-TR- 1513-2017

TECHNICAL REPORT SERIES

No. CS-TR- 1513 October, 2017

The 15th Overture Workshop: New Capabilities and Applications for Model-based Systems Engineering

J. S. Fitzgerald, P. W. V. Tran-Jørgensen, and T. Oda

Abstract

The 15th in the “Overture" series”of workshops on the Vienna Development Method (VDM), its associated tools and applications, was held at Newcastle University, UK, on 15th September, 2017. The workshop provided a forum for discussing and advancing the state of the art in formal modelling and analysis using VDM and its family of associated formalisms including extensions for describing distributed, real-time and cyber-physical systems. The workshop reflected the breadth and depth of work supporting and applying VDM. Contributions covered topics as diverse as architectural frameworks of autonomous Cyber-Physical Systems, tool building techniques such as interpreter design, automated code generation, testing and debugging and case studies of formal modelling and tool migration.

Keywords: model-based engineering, formal methods, VDM

Printed and published by Newcastle University,
School of Computing, Urban Sciences Building,
Newcastle upon Tyne, NE4 5TG, England.

Bibliographical details

Title and Authors

NEWCASTLE UNIVERSITY
School of Computing. Technical Report Series. CS-TR- 1513

Abstract:
The 15th in the “Overture" series” of workshops on the Vienna Development Method (VDM), its associated tools and applications, was held at Newcastle University, UK, on 15th September, 2017. The workshop provided a forum for discussing and advancing the state of the art in formal modelling and analysis using VDM and its family of associated formalisms including extensions for describing distributed, real-time and cyber-physical systems. The workshop reflected the breadth and depth of work supporting and applying VDM. Contributions covered topics as diverse as architectural frameworks of autonomous Cyber-Physical Systems, tool building techniques such as interpreter design, automated code generation, testing and debugging and case studies of formal modelling and tool migration.

About the authors:
John S Fitzgerald, PhD, FBCS is Professor of Formal Methods in Computing Science at Newcastle University.

Peter W. V. Tran-Jorgensen, PhD, is a postdoctoral research Scientist with the Department of Engineering at Aarhus University, Denmark.

Tomohiro Oda, PhD, is a Formal Specification Engineer with Software Research Associates, Inc.

Suggested keywords:
Keywords: model-based engineering, formal methods, VDM

Proceedings of the 15th Overture Workshop

The 15th in the "Overture" series of workshops on the Vienna Development Method (VDM), its tools and applications, was held at Newcastle University, UK, on the 15th September 2017. The workshop provided a forum for discussing and advancing the state of the art in formal modelling and analysis using VDM and its associated formalisms including extensions for distributed, real-time and cyber-physical systems.

VDM is one of the longest established formal methods, and has a lively community of researchers and practitioners in academia and industry that has grown around the modelling languages VDM-SL, VDM++, VDM-RT, and CML. Research in VDM and Overture is driven by the need to develop industry practice as well as fundamental theory. Consequently, the provision of tool support has been a priority for many years. Starting from VDMTools and the Overture platform, the Overture initiative provides a community and open technical platform for developing in modelling and analysis technology, including static analysis, interpreters, test generation, execution support and model checking. The fruit of the initiative in recent years has included industry-strength tools for co-modelling and co-simulation in embedded systems design (Crescendo), System-of-Systems modelling, verification and testing (Symphony), and latterly design of cyber-physical systems (INTO-CPS). At the time of writing, the tools are being applied in the INTO-CPS project as well as the TEMPO, CPSBuDi and iPP4CPPS experiments supported by the CPSE-Labs project.

The communitys growth has been greatly assisted by the Overture workshop series. The 15th Workshop reflected the breadth and depth of work supporting and applying VDM. Contributions covered topics as diverse as architectural frameworks of autonomous cyber-physical systems, tool building techniques such as interpreter design, automated code generation, testing and debugging and case studies of formal modelling and tool migration. A special session was led by Graeme Young of Newcastle University, including a structured discussion on future exploitation of Overture technology.

We would like to thank the authors, PC members, reviewers and participants for their help in making this a valuable and successful workshop.

John Fitzgerald
Peter W. V. Tran-Jørgensen
Tomohiro Oda

Program Committee

- Keijiro Araki, Kyushu University, Japan
- Nick Battle, Fujitsu, United Kingdom
- Luis Diogo Couto, UTRC, Ireland
- John Fitzgerald, Newcastle University, United Kingdom (Chair)
- Peter Gorm Larsen, Aarhus University, Denmark
- Fuyuki Ishikawa, National Institute of Informatics, Japan
- Nico Plat, Thanos, The Netherlands
- Peter W. V. Tran-Jørgensen, Aarhus University, Denmark (Chair)
- Tomohiro Oda, Software Research Associates, Inc., Japan (Chair)
- Paolo Masci, Universidade do Minho, Portugal
- Kenneth Pierce, Newcastle University, United Kingdom
- Marcel Verhoef, European Space Agency, The Netherlands
- Sune Wolff, Unity, Denmark

Table of Contents

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

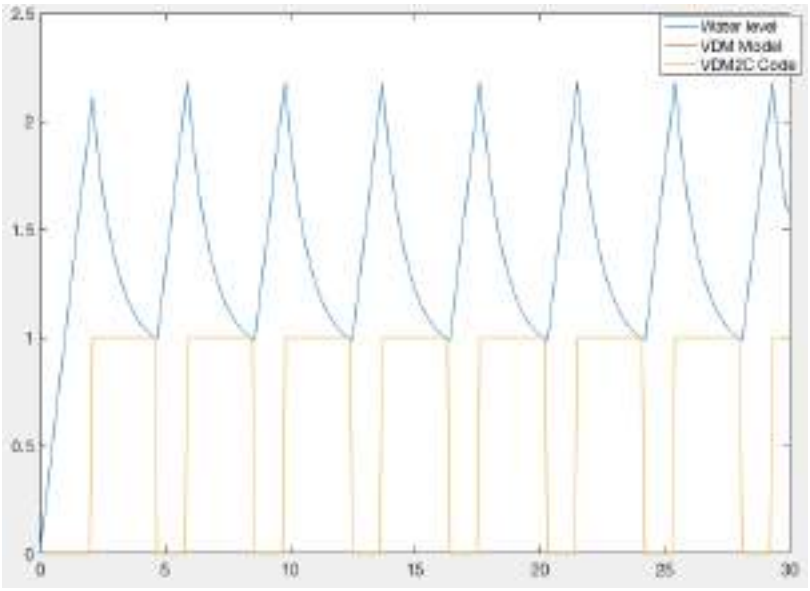


Fig. 1: Comparison of the value outputs of the tool-wrapper and standalone FMUs.

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

Transitioning from Crescendo to INTO-CPS

Kenneth Lausdahl[1], Kim Bjerge[1], Tom Bokhove[2], Frank Groen[2], and Peter Gorm Larsen[1]

[1] Aarhus University, Denmark
[2] Controllab Products, Netherlands

Abstract. In the development of Cyber-Physical Systems, interoperability between tools supporting heterogeneous models is essential. In the Crescendo technology, a direct connection between simulation in Overture and solving in 20-sim was established. In the Integrated Tool Chain for Model-based Design of Cyber-Physical Systems (INTO-CPS) project, this connection was generalized to include any number of constituent components. The connections between the different simulations were established using the Functional Mock-up Interface. This paper explores the portability of a model of a trolley conveyor system from a Crescendo setting to the INTO-CPS technology.

1 Introduction

Whenever it is desirable to combine models described in different formalisms that do not share the same semantic foundations, it is worthwhile to consider technologies that enable a collaborative simulation of the different constituent models. Such simulations are called co-simulations [13]. In the Design Support and Tooling for Embedded Control Software (DESTECS) project [4], such a solution, called Crescendo [12], was enabled by coupling the Discrete Event (DE) formalism VDM, supported by the Overture tool [16], with the Continous-Time (CT) formalism bond graphs, which is supported by the 20-sim tool [15]. This work was subsequently generalised in the INTO-CPS project to any number of DE and CT models using the Functional Mock-up Interface (FMI) to connect the constituent models, represented as Functional Mock-up Units (FMUs) [9,10,17,19].

The conversion from a collaborative model produced using Crescendo to a corresponding INTO-CPS solution has already been reported [22]. However, the case study we present in this paper is novel in the sense that it also uses event-driven control, which are not supported by the FMI standard. The event-driven control which takes place when something happens oppose to time-driven control which occurs as predefined time intervals. In addition, this work also attempts to turn the 3D animations made inside 20-sim into a special 3D FMU so that it is possible to demonstrate system model behaviour in a user-friendly manner.

The case study used is a trolley conveyor system in which the trolley can be tilted. This functionality is used, for example, to move parcels on the trolleys to their desired destination. The case study originates in a collaboration with BEUMER Group, who deliver such conveyor systems, for example, for baggage handling in airports all over the world. In this work we are interested in tilting the trolleys as they pass around a

bend in the conveyor in order to compensate for the centrifugal force generated at high travel speeds.

This paper is structured as follows. Section 2 introduces the different co-simulation technologies used in this work. Section 3 introduces the case study used here. This is followed in Section 4 by an explanation of the transition from the Crescendo technology to the INTO-CPS setting. Finally, Section 5 explains the transition of the 3D animation from 20-sim to Unity, while Section 6 provides a number of concluding remarks.

2 The Co-simulation Technologies used

The technology initially used to realise the case study was the Crescendo Tool. It combined two tools, Overture and 20-sim, allowing co-simulation of DE and CT models. The INTO-CPS technology – the target of this paper – allows for not only two, but n models of any combination of DE and CT to be simulated. The project is focused on the FMI standard.

2.1 Crescendo

The Crescendo tool is the outcome of the European Framework 7 project DESTECS, (`www.destecs.org`) which ran from early 2010 until 2013 [4,11]. Its aim was to develop methods and tools that combine continuous time models of systems (in tools such as 20-sim) with discrete event controller models in the Vienna Development Method (VDM) through co-simulation [25,23]. The approach was intended to encourage collaborative multidisciplinary modelling, including modelling of faults and fault tolerance mechanisms. The Crescendo tool, in combination with Overture and 20-sim, provides a platform for co-simulation. The platform only supports two simulators using a variable step size algorithm to progress simulation time. The intention in the project was to have one discrete controller, which typically did not support roll-back, and allow the plant model to be in charge of providing future synchronization time intervals. In Crescendo, the controllers were modelled in VDM using the Real-Time dialect, which added timing information to the execution of each expression and statement [24]. The physical plant model was intended to be modelled using 20-sim in continuous time using block diagrams or bond graphs. The tool shown in Figure 1 provides a single user interface based on Eclipse for VDM modelling and co-simulation, and good integration with 20-sim. The co-simulation is defined based on a contract which consists of *shared design parameters* used for sharing constant parameters between models; *monitored* variables used for providing inputs to the controller model; *controlled* variables which are controller outputs and *events* which are used like interrupts in the controller.

2.2 The Functional Mock-up Interface

The tool-independent standard FMI was developed within the MODELISAR project [14]. It supports both model exchange and co-simulation and exists in two versions. Version 1.0, released in 2010 and Version 2.0, released in 2014. It was developed to improve

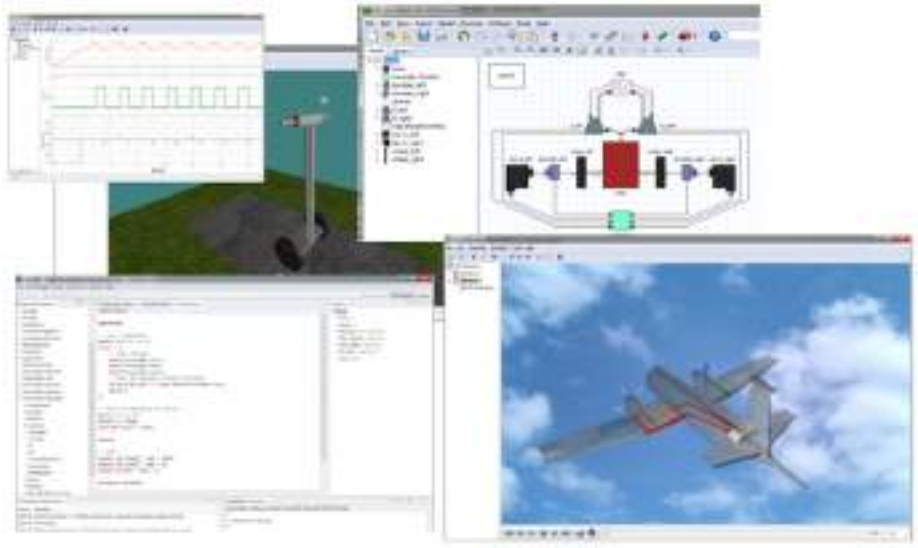


Fig. 1: Different screen shoots from the Crescendo tool

exchange of simulation models between suppliers and Original Equipment Manufacturers (OEM). The standard describes how simulation units are to be exchanged as ZIP archives called Functional Mock-up Unit (FMU)s and how the model interface is described in an XML file named *modelDescription.xml*. The functional interface of the model is described as a number of C functions that must be exported by the library that implements the model inside the FMU. Since the FMU only contains a binary implementation of the model, it offers some level of intellectual property protection. The standard lists a number of constraints on how the C functions in FMI can be called without an explicit co-simulation algorithm.

2.3 The INTO-CPS Technology

The vision of the INTO-CPS[3] consortium is that Cyber-Physical Systems (CPS) engineers should be able to deploy a wide range of tools to support model-based design and analysis, rather than relying on a single *factotum*. The goal of the project is to develop an integrated "tool chain" that supports multidisciplinary, collaborative modelling of CPSs from requirements, through design, to realisation in hardware and software, enabling traceability through the development. The project integrates existing industry-strength baseline tools in their application domains, based centrally around FMI-compliant co-simulation [3]. The project focuses on the pragmatic integration of these tools, allowing for extensions in areas where a need is recognised. The tool chain is underpinned by well-founded semantic foundations that ensures the results of analysis can be trusted [26,7].

[3] See `http://into-cps.au.dk/`.

The tool chain provides powerful analysis techniques for CPS models, including generation and static checking of FMI interfaces; model checking; Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation, supported by code generation. This enables both Test Automation (TA) and Design Space Exploration (DSE) of CPSs.

The INTO-CPS project provides a lightweight application interface for managing the development of constituent system models for co-simulations and for configuring and running such co-simulations. The project offers the following FMI-enabled modelling tools: Overture, 20-sim, OpenModelica and RT Tester. However, it is an open tool chain, so in general any FMI 2.0 enabled tool can be used.

The project also provides high-level system design facilities using the Systems Modelling Language (SysML) through the Modelio tool [20]. A co-simulation configuration can be modelled in SysML using block diagrams and a custom INTO-CPS profile for FMI. Co-simulation in INTO-CPS conforms to FMI version 2.0 and the master algorithm supports both fixed and variable step sizes. The variable step size algorithm supports constraints to adjust the step size. There are four types of constraints that can be utilized:

Zero Crossing: A zero crossing constraint is a continuous constraint. A zero crossing occurs at the point where a function changes its sign. In simulation, it can be important to adjust the step size such that a zero crossing is revealed as accurately as possible. For instance, a ball should rebound from a wall exactly when the distance between the ball and the wall hits zero and not before or after. A solver in a tool such as Simulink can adjust the step size using iterative approaches, but in a co-simulation a roll-back of the participating models' internal states would be required, but is in general not possible or efficient. Hence, the variable step size calculator bases its step size adjustments on the prediction of a future zero crossing. It uses extrapolation and derivative estimation to estimate changes and therefore reduce the need for roll-back.

Bounded Difference: A bounded difference constraint is also a continuous constraint. A bounded difference ensures that the minimal and maximal value of a set of values do not differ by more than a specified amount (the underlying assumption is that this difference becomes smaller when the step size is reduced). The bounded difference problem is distinct from the zero-crossing problem in that there is not a specific time instant (the zero crossing) to hit, but rather a specific time difference (the step size that keeps the difference bounded).

Sampling Rate: A sampling rate constraint is a discrete constraint. It constrains the step size such that repetitive, predefined time instants are exactly hit. This can be useful in co-simulation, for instance, when a modelled control unit reads a sensor value every x milliseconds.

FMU Max Step Size: This constraint implements the `getMaxStepSize` method from [6,8] providing a good prediction of the maximal step size that a given FMU instance can perform at a given point in time. It limits the need to roll back a simulation at a given point. It is enabled by default and it constrains the step size as follows:

$$size = min(\{(getMaxStepSize(i) \mid \forall i \in instances\})$$

These constrains can be used to improve the overall co-simulation result by reducing the step size at important time points during a simulation. This can be illustrated by a small example of a water tank which has a constant inflow and a drain valve where the aim is to keep the water level between a minimal and maximal marker as shown in Figure 2.

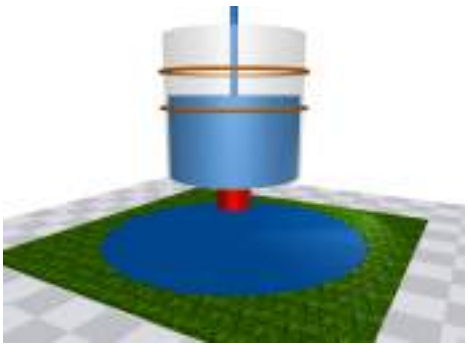


Fig. 2: Water tank with min and max marker

If the controller is implemented as a periodic check then the constraint *FMU Max Step Size* can be added to make sure that the co-simulation does not perform steps larger than the period of the controller. However, if the controller is implemented at event-driven control then the *Zero Crossing* constraint will result in a faster co-simulation while retaining the same simulation precession since the controller only deals with points in time where the level is about to cross the min and max markers. Defining a zero crossing constraint between (`level` and `max`) and (`level` and `min`) will result in a reduction of the step size when approaching the zero crossing and making sure that a step is taken as close to the crossing as possible.

The Overture FMU Extension The Overture tool is extended with an FMU exporter that conforms to FMI 2.0. The exporter relies on a VDM FMI library and modelling guideline. The library consists of a `Port` class and the following specialisations: `BoolPort`, `IntPort`, `RealPort`, and `StringPort`, each mapping to the corresponding scalar variable type in FMI. These all have `getValue` and `setValue` methods. Integration with FMI is modelled using a special `HardwareInterface` class where parameters, inputs and outputs are modelled using this port hierarchy. Through the explicit modelling of FMI scalar variables as ports, it is possible to generate FMUs without requiring additional input. A simple mapping can be made from *shared design variables*: `sdp` into parameters, `controlled` into input on the plant side and output on the controller.

The tool requires the user to follow a design pattern where an instance of `HardwareInterface` is declared in the `System` class, and a `World` class is responsible for blocking the initial thread to ensure infinite execution. This class must contain all

ports and custom annotations that set the type and name of the exported port. Listing 1.1 shows a small example of three ports exported as parameter, input and output, respectively.

```
class HardwareInterface
values
  -- @ interface: type = parameter;
  public v : RealPort = new RealPort(1.0);

instance variables
  -- @ interface: type = input;
  public distanceTravelled : RealPort := new RealPort(0.0);
  -- @ interface: type = output;
  public setAngle : RealPort := new RealPort(0.0);

end HardwareInterface
```

Listing 1.1: The `HardwareInterface` class

While these all map with the same semantics as in Crescendo, this does not apply for `events`, which are not supported in INTO-CPS.

20-sim FMU Extension The 20-sim tool is extended with FMI support through a custom code generator template[4] that is able to export a module in a 20-sim model as an FMU. This solution does not support event detection and external library usage (Dynamic-Link Libraries, or DLLs), which could be used,for example, for collision detection using a physics library. It has support for the following integration methods: Discrete Time, Euler, Runge Kutta 2, Runge Kutta 4, CVODE[5], and MeBDFi[6]. A tool wrapper is also available which does not have these limitations and which thus works much like in Crescendo, where the full potential of the simulator can be used during simulation.

3 Case Study

The trolley conveyor case study aims to investigate the behaviour of different tilt strategies when a parcel follows a specific track. The model describes the environment for the path of the conveyor system, which is composed of both straight and curved segments, as illustrated in Figure 3. On this conveyor, parcels are transported on a trolley at a constant speed. In the bends of the conveyor, the trolley is tilted to ensure that parcels are kept in the same position on the trolley [2]. Actuators on the trolley control the tilt to

[4] The FMU generator template for 20-sim is available at `https://github.com/controllab/fmi-export-20sim/`.

[5] `https://computation.llnl.gov/projects/sundials/cvode`

[6] `http://www.netlib.org/ode/mebdfi.f)`

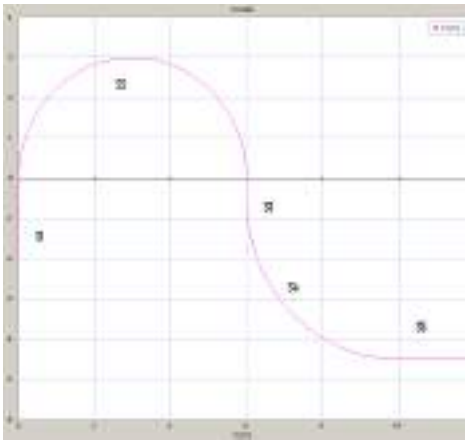


Fig. 3: Conveyor Path and Segments

the desired position such that the centrifugal force experienced by the parcel is neutralized. Only one trolley is modelled, with the focus being on transporting a single parcel. The CT model is composed of a number of blocks modelling the environment, which includes the conveyor path, trolley and parcel. These modelling blocks are described briefly in this section.

A CT model consists of components connected by power and signal ports. Power ports are bidirectional and represent energy flow, while signal ports are unidirectional, from one component to another, and carry information. In the SysML internal block diagram these signals are described with arrows on edges between the component blocks. The `SetAngle` in Figure 4 is such a signal that simulates the current desired tilt angle. A power port is always characterized by two domain independent variables [1]. The product of these variables has the dimension of power, measured in Watts. Therefore, they are called *power conjugated variables*. In the mechanical domain the variables are force and velocity, for the electrical domain the variables are voltage and current. In 20-sim diagrams, like those we use in this CT model, a power port is a port where power can be exchanged between a component and its environment in terms of these.

The purpose of the DE model is to control the motion of the tilting trolley. Exploring different motion curves for the tilting trolley device has been the primary purpose of this part of the model. In the DESTECS project, the DE model was linked to the CT model in terms of the co-model, interface as described in the DESTECS methodological guidelines [5].

4 Transitioning to INTO-CPS

To perform a translation from a Crescendo model into an INTO-CPS co-simulation project, both simulation tools must be capable of exporting FMI 2 compliant FMUs as described in section 2.3. The original model of the Conveyor-Path (CP) uses events that are not supported in INTO-CPS. The Crescendo co-simulation contract for the CP

model is shown in Listing 1.2. It describes the interface between the DE controller and the CT plant model. The first section of the contract lists Shared Design Parameters (SDP), that is, shared constants that are made available to both simulators. Controlled variables set by the controller become monitored variables that are observed by the controller. Finally, events generated by the CT model and handled by the DE model.

```
sdp real v;
sdp real r2;
sdp real r4;
sdp real l1;
sdp real l3;
sdp real trayPitch;
sdp real p;

controlled real setAngle := 0.0;

monitored real distanceTravelled := 0.0;
monitored real distCTB1 := 0.0;
monitored real distCTB2 := 0.0;
monitored real distCTB3 := 0.0;
monitored real distCTB4 := 0.0;

event eventCTB1;
event eventCTB2;
event eventCTB3;
event eventCTB4;
```

Listing 1.2: Crescendo contract

The first step in a transition to INTO-CPS is to convert the contract to establish the interface for the co-models for the controller and plant. The *controlled* variables from Crescendo can be mapped directly to output ports, and any *monitored* variables can be mapped directly to input ports. The SDP can be mapped to properties of both the controller and plant block. FMI does not have a notion of SDP, but individual FMUs can have parameters. Once block creation is completed, the connections are made and this mapping preserves the same overall co-simulation semantics as in Crescendo. In Figure 4 the SysML connection diagram of the CP model is shown. Note that events are also converted to ports and connections.

The diagram is made in the Modelio[7] tool using the INTO-CPS profile for SysML. This tool can export an initial model description for each FMU containing the required scalar variables (a.k.a. ports). It can also export the connection information in a format supported by the INTO-CPS Application, which is the interface used for performing co-simulations.

4.1 Updating the DE Controller model

The model description exported from Modelio is imported into Overture, resulting in the `HardwareInterface` class shown in Listing 1.3. It lists all the ports used by

[7] `https://www.modelio.org/`.

Fig. 4: SysML connection diagram for the CP model.

the controller. The remainder of the VDM model must also be updated to use the ports from the new FMI library instead of the local instance variables used in Crescendo.

The events cannot be directly mapped because FMI does not have a corresponding notion. The events are not essential for the controller in the CP model, as they mainly serve a logging purpose. It is however still possible to achieve the same effect using a real port which is set to change its value at the same time as the events in the original model when used in combination with the variable step algorithm. The variable step algorithm can be configured with constraints if needed to make sure that a step is performed on every value change.

```
class HardwareInterface

values

  -- @ interface: type = parameter;
  public v : RealPort = new RealPort(2.0); -- {m/s}
  -- @ interface: type = parameter;
  public  r_max : RealPort = new RealPort(5.0); -- {m} maximum radius
  -- @ interface: type = parameter;
  public  r2 : RealPort = new RealPort(3.0); -- {m} radius first curve
  -- @ interface: type = parameter;
  public  r4 : RealPort = new RealPort(4.0); -- {m} radius second curve
  -- @ interface: type = parameter;
  public  l1 : RealPort = new RealPort(2.0); -- {m} lenght of first straight line
  -- @ interface: type = parameter;
  public  l3 : RealPort = new RealPort(0.5); -- {m} lenght of straight line
    -- between curves
  -- @ interface: type = parameter;
  public  trayPitch : RealPort = new RealPort(0.9); --{m}
  -- @ interface: type = parameter;
  public  p : RealPort = new RealPort(0.5); -- Percentages accelerating

instance variables
    -- @ interface: type = input;
    public distanceTravelled : RealPort := new RealPort(0.0);--Distance since start
    -- @ interface: type = input;
    public distCTB1 : RealPort := new RealPort(0.0);  -- Position of CTB1
    -- @ interface: type = input;
    public distCTB2 : RealPort := new RealPort(0.0);  -- Position of CTB2
    -- @ interface: type = input;
    public distCTB3 : RealPort := new RealPort(0.0);  -- Position of CTB3
    -- @ interface: type = input;
    public distCTB4 : RealPort := new RealPort(0.0);  -- Position of CTB4
    -- @ interface: type = input;
    public switchNumber : RealPort := new RealPort(0.0);
```

```
    -- @ interface: type = output;
    public setAngle : RealPort := new RealPort(0.0);  -- Desired bank angle

end HardwareInterface
```

Listing 1.3: Hardware interface for case study

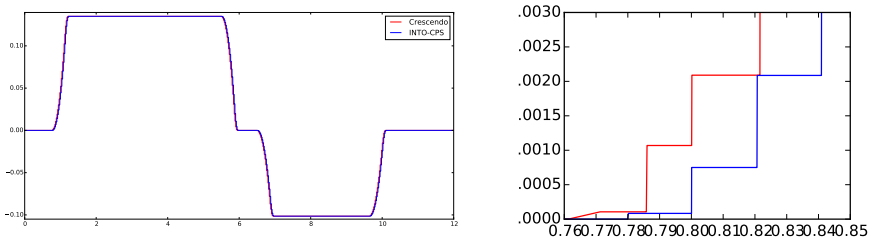
4.2 Updating the CT plant model

There were two options that could be used for updating the 20-sim model: 1) Use the standalone FMU exporter with an updated version of the model, 2) Use a tool wrapper FMU that uses 20-sim for simulation without requiring model changes, except for handling events.

The first option is already supported, since 20-sim has been updated to support FMI using code generation, similar to how 20-sim can generate code for deployment. The result is a stand-alone FMU that does not require 20-sim during co-simulation. The stand-alone FMU can be generated from any submodel. The case study was then changed by replacing the block that encoded the shared DESTECS variables with a block that was created by importing the interface description from Modelio into 20-sim to ensure that the interface matched the overall design.

The second option did not require changes to the model, since a tool wrapper FMU support feature would interact with the model in its current state. To enable this, 20-sim was in fact updated with support for tool wrapper FMUs. It uses the same import and export variables as the original Crescendo model, which means that this model can be used directly without modification for the INTO-CPS co-simulation. The generated tool wrapper FMU lists the import and export variables as input and output scalar variables. The original 20-sim model is also stored inside the FMU. This model is extracted when the co-simulation is initialized, and the FMU tool wrapper then starts 20-sim and requests it to open the 20-sim model from the tool wrapper FMU. If the model opens successfully, the tool wrapper communicates with the 20-sim instance using the same interface as the Crescendo tool, thus the tool wrapper FMU acts as a proxy between INTO-CPS and the 20-sim model.

The model has further been improved by using an external collision detection package to improve the detection of collisions between the parcel and the tray. Because the collision detection package uses an external Dynamic-Link Library (DLL), the first option of using a stand-alone FMU through code generation could not be used. The tool wrapper FMU export was the only option, then, for this model. Furthermore, events were replaced by a `switchNumber` variable that is set to the segment that is approaching at the same moment that the event was triggered in the Crescendo co-simulation. The event handlers in the DE model can be triggered by casing on the `switchNumber` variable. It should be noted that this event handling is not critical to the controller's ability to perform correct control, but is mainly used for user feedback. If these events were critical they should be constrained by e.g. a zero crossing constraint.



(a) Comparison of desiredBankAngle

(b) Crescendo offset.

Fig. 5: Plot of simulated desiredBankAngle in Crescendo verses INTO-CPS.

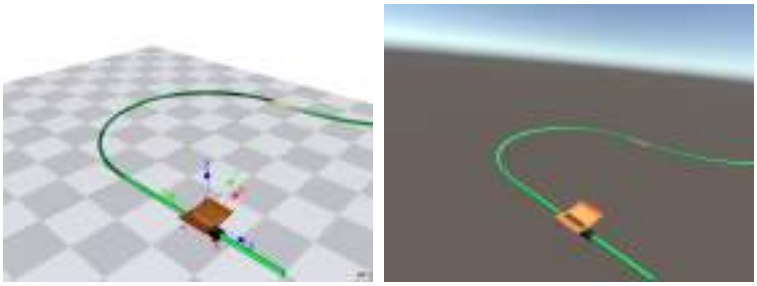
4.3 The INTO-CPS Co-Simulation

The INTO-CPS co-simulation has been carried out using the controller FMU exported from Overture and the tool wrapper plant FMU exported from 20-sim. The simulation was configured in the INTO-CPS Application based on the SysML connection diagram of the connection definitions. The variable step size algorithm was used with the *FMU Max Step Size* constraint enabled. This algorithm configuration is nearly identical to the one used in Crescendo. In Figure 5a the desired bank angle from the controller is plotted from a simulation performed both in Crescendo and INTO-CPS. It shows that the behaviour of the controller is the same but with a small offset. The model itself behaves correctly in both simulations and the shape of the bank angle is also as expected.

The initial offset, illustrated in Figure 5b, seems to be caused by the plant model integration method not being able to precisely match the desired point in time (multiple of 0.02 seconds) during the first two steps of the simulation. It can be seen that the INTO-CPS simulation always keeps a span of 0.02 seconds between control changes, while the first two steps in Crescendo do not match the period of the controller because of this offset. It has not been possible to determine why the Crescendo simulation has these deviations. It could be that the newer version of the tools have improved the ability to hit the precise point in time, or that it is related to model enhancements that come with a new collision detection library. The Crescendo simulation was performed with Crescendo version 2.0.0 (build GF1B8817) and 20sim version 4.4.1 (build 4356). The INTO-CPS simulation was performed with COE version 0.2.6, Overture FMI Exporter version 0.2.8 and 20sim 4.6.4 (build 8007).

5 Moving the Animation into Unity

As opposed to most FMUs, an animation FMU does not add any simulation data to a co-simulation. However, its purpose lies in giving another view of the data generated by the other components of the co-simulation. It becomes possible to see the behaviour of a co-simulation in a different way than simply observing signal graphs. This is useful for physical systems like robots or the trolley conveyor system, because it becomes



(a) 20-sim 3D animation (b) Unity 3D animation

Fig. 6: 20-sim versus Unity 3D animation of the trolley conveyor system.

possible to have direct visual feedback of the actual physical system simulated in the co-simulation.

5.1 Providing 3D animation for INTO-CPS simulations

During the DESTECS project, 20-sim was used for 3D visualization of the models during simulation. To bring the same level of visualization to the INTO-CPS project, a custom FMU was developed based on the 3D animation feature of 20-sim. This enabled a generic solution for 3D animations in any co-simulation in INTO-CPS. This solution provides the same feature as in DESTECS. Although Unity is a cross-platform animation framework, the link to 20-sim currently makes it impossible to provide, in a 3D animation FMU, the platform independence that FMI has the ability to support, since 20-sim only supports Microsoft Windows platforms.

5.2 Converting the Animation into Unity

To improve upon the render quality of the animation used in the DESTECS project, a game engine called Unity [21] was employed in the INTO-CPS setting. A conversion tool has also been developed during the project that converts a 3D scene produced in 20-sim into a 3D scene usable in Unity. The 3D scenery of the CP case study from section 3 was converted from 20-sim to Unity. The original 3D animation and the result of the conversion to Unity is shown in Figure 6.

The conversion is not yet completely covered. Some of the objects, lighting and cameras, among others, are implemented differently in Unity with respect to 20-sim, making it difficult to convert them. This can also be observed from the illustrations in Figure 6, as the camera angle is different between both animations. The difference in lighting causes colours to also look different. The colours, however, have been verified to be identical between 20-sim and Unity.

Unity has additional features to make an animation look more polished. Examples include support of virtual and augmented reality, the production of complex particle effects and the possibility to add assets like skyboxes and floor textures. As an example, a skybox and floor texture were added to the original Unity animation in Figure 6, resulting in Figure 7.

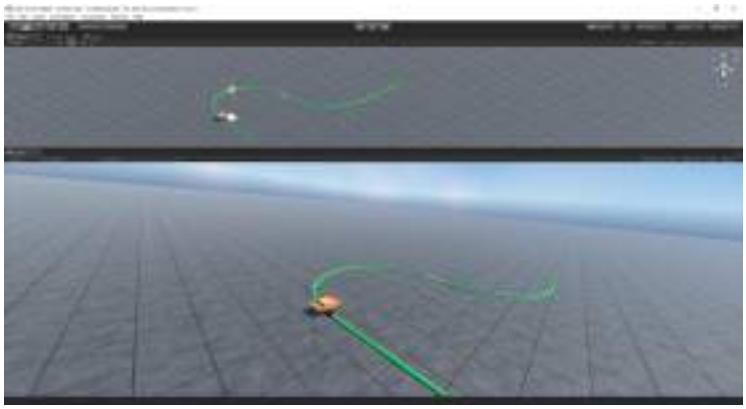


Fig. 7: Improved Unity 3D animation of the trolley conveyor system, as part of the Unity Editor environment.

5.3 FMI Support for Unity Animations

A new tool has been created for Unity that enables arbitrary Unity models to be exported as tool wrapper FMUs that can be used in co-simulation on the platforms supported by Unity. It include a thin FMI implementation that provides communication to the Unity animation stored inside the resource folder of the FMU. Upon instantiation it will launch the Unity application and establish a communication link between it and the FMI functions. During simulation, the new values are sent to the animation, which in turn renders the animation in relation to these new values.

The tool is implemented as a Unity plug-in that enables the FMU to be exported from within Unity at the press of a button. It has an editor extension that provides an interface to select the axes of objects that should be included in the FMU. It also enables the user to name the axes that become the inputs of the FMU. Among others, the supported axes are position, rotation and scaling of a 3D object, but also Unity-specific features like texture sliding, blend shapes and showing or hiding 3D objects.

Finally, to use the animation from section 5.2 during co-simulation, it has been exported as an FMU and the *Plant* FMU has been updated to export the signals required to drive the animation. The following signals have been added as outputs from the *Plant* and connected to the animation:

- `TrayPositionCtrl_z_sphere2`
- `TrayPositionCtrl_y_sphere2`
- `TrayPositionCtrl_x_sphere2`
- `TrayPositionCtrl_z_sphere3`
- `TrayPositionCtrl_y_sphere3`
- `TrayPositionCtrl_x_sphere3`
- `TrayPositionCtrl_z_sphere1`
- `TrayPositionCtrl_y_sphere1`
- `TrayPositionCtrl_x_sphere1`
- `TrayModel_Tray_y`
- `TrayModel_Tray_x`
- `TrayModel_Tray_angle`
- `TrayModel_Parcel_y`
- `TrayModel_Parcel_angle`
- `TrayModel_Parcel_x`
- `TrayModel_Slider_z`
- `TrayModel_Slider_y`
- `TrayModel_Slider_x`
- `TrayModel_Slider_angle`
- `Tray_Loader_angle_y`

6 Concluding Remarks

This paper has illustrated that it is possible to efficiently transfer collaborative models developed using the Crescendo technology to the INTO-CPS FMI-based setting. From a debugging perspective, the Crescendo technology still has some advantages over the new INTO-CPS tool chain. However, there are many limitations of Crescendo that are eliminated with the INTO-CPS tool chain [18]. Most importantly, INTO-CPS represents an open tool chain that enables users to include any tools that are able to provide FMI version 2.0 compliant co-simulatable FMUs. It is also worth noting that it supports the full development life cycle, from initial requirements towards the final realisations of all constituents included. We have also illustrated that the case study can be simulated in INTO-CPS and determined that events were not critical for the control of the model, while showing how constraints can be added to handle event-controlled models.

Acknowledgments. 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. We would like to thank all the participants of this project for their efforts making this a reality, and Victor Bandur for feedback on this article. We would also like to thank BEUMER Group for the ability to analyse this case study.

References

1. van Amerongen, J.: Dynamical Systems for Creative Technology. Controllab Products, Enschede, Netherlands (2010)
2. Bjerge, K., Larsen, P.G.: Using VDM in a Co-Simulation Setting for an Industrial Conveyor System. In: The Modelling, Identification and Control – MIC 2013 conference. IASTED, Innsbruck, Austria (February 2013)
3. 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 (September 2012)
4. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., Wouters, F.: Design Support and Tooling for Dependable Embedded Control Software. In: Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems. pp. 77–82. ACM (April 2010)
5. Broenink, J.F., Fitzgerald, J., Pierce, K., Ni, Y., Zhang, X.: Methodological guidelines 1. Tech. rep., The DESTECS Project (INFSO-ICT-248134) (December 2010)
6. Broman, D., Brooks, C., Greenberg, L., Lee, E., Masin, M., Tripakis, S., Wetter, M.: Determinate composition of FMUs for co-simulation. In: Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on. pp. 1–12 (2013)
7. Cavalcanti, A., Woodcock, J., Amálio, N.: Behavioural models for fmi co-simulations. In: Sampaio, A.C.A., Wang, F. (eds.) International Colloquium on Theoretical Aspects of Computing. Lecture Notes in Computer Science, Springer (2016)
8. Cremona, F., Lohstroh, M., Broman, D., Natale, M.D., Lee, E.A., Tripakis, S.: Step revision in hybrid co-simulation with FMI. In: MEMOCODE. pp. 173–183. IEEE (2016)
9. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)

10. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In: Proc. INCOSE Intl. Symp. on Systems Engineering. Edinburgh, Scotland (July 2016)
11. Fitzgerald, J., Larsen, P.G., Pierce, K., Verhoef, M.: A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems. Mathematical Structures in Computer Science 23(4), 726–750 (2013)
12. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2013)
13. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: State of the art. Tech. rep. (feb 2017), http://arxiv.org/abs/1702.00686
14. ITEA Office Association: Itea 3 project 07006 modelisar. https://itea3.org/project/modelisar.html (December 2015), (Visited on 12/06/2015)
15. Kleijn, C.: Modelling and Simulation of Fluid Power Systems with 20-sim. Intl. Journal of Fluid Power 7(3) (November 2006)
16. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
17. 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: CPS Data Workshop. Vienna, Austria (April 2016)
18. 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 (September 2017)
19. Larsen, P.G., Fitzgerald, J., Woodcock, J., Lecomte, T.: Trustworthy Cyber-Physical Systems Engineering, chap. Chapter 8: Collaborative Modelling and Simulation for Cyber-Physical Systems. Chapman and Hall/CRC (September 2016), iSBN 9781498742450
20. Quadri, I., Bagnato, A., Brosse, E., Sadovykh, A.: Modeling Methodologies for Cyber-Physical Systems: Research Field Study on Inherent and Future Challenges. Ada User Journal 36(4), 246–253 (December 2015), http://www.ada-europe.org/archive/auj/auj-36-4.pdf
21. Technologies, U.: Unity. https://unity3d.com/ (December 2016)
22. Thule, C., Nilsson, R.: Considering Abstraction Levels on a Case Study. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 16–31. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
23. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2009)
24. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)
25. Verhoef, M., Visser, P., Hooman, J., Broenink, J.: Co-simulation of Distributed Embedded Real-time Control Systems. In: Davies, J., Gibbons, J. (eds.) Integrated Formal Methods: Proc. 6th. Intl. Conference. pp. 639–658. Lecture Notes in Computer Science 4591, Springer-Verlag (July 2007)
26. Woodcock, J., Foster, S., Butterfield, A.: Heterogeneous Semantics and Unifying Theories, pp. 374–394. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-47166-2_26

Modelling Network Connections in FMI with an Explicit Network Model

Luis Diogo Couto[1] and Ken Pierce[2]

[1] United Technologies Research Center, Ireland
CoutoLD@utrc.utc.com
[2] Newcastle University, United Kingdom
kenneth.pierce@newcastle.ac.uk

Abstract. The Functional Mock-up Interface (FMI) is an emerging standard for co-simulation of models packaged as Functional Mock-up Units (FMUs). The FMI standard permits FMUs to share data during simulation using a simple set of types (Booleans, integers, reals and strings). FMI was primarily designed to connect continuous-time models (described as differential equations) for simulation of physical processes. The application of FMI to cyber-physical systems (CPSs) requires models of networked, discrete-event (DE) controllers. This presents a challenge for modellers as the simple FMI interface does not provide abstractions for modelling network communications. In this paper we present a discussion of the limitations of FMI in modelling networked controllers, and demonstrate a proof-of-concept of one possible solution in the form of an explicit model of the network medium called the *ether*. We describe an example ether FMU implemented in the VDM (Vienna Development Method) modelling language and its application in a smart building case study. We discuss the limitations of the approach and consider its potential application in other cases.

Keywords: Functional Mock-up Interface (FMI), co-modelling, co-simulation, network modelling, ether pattern, Vienna Development Method (VDM), Overture

1 Introduction

The design of cyber-physical systems (CPSs) requires engineers from diverse disciplines to co-operate to build and integrate a wide range of physical and software components. Model-based design is increasingly popular for building individual components, but each engineering discipline has its own paradigms and formalisms. Software engineers typically use discrete-event (DE) formalisms such as VDM (Vienna Development Method), whereas mechanical engineers use continuous-time (CT) formalisms described in differential equations. One way to allow these engineers to work together is through collaborative modelling, in which constituent models are connected together to form a *multi-model*. Such multi-models allow for analysis of CPSs at a system-level.

The Functional Mock-up Interface (FMI) is an emerging standard for co-simulation of multi-models, supported by platforms such as INTO-CPS[3] [3, 4, 6–8]. In FMI, each

[3] https://into-cps.github.io/

constituent model is packaged as an executable unit called a Functional Mock-up Unit (FMU). A multi-model comprises a set of FMUs and a configuration file describing the composition of the FMUs, their connections, and details of the simulation (e.g. duration).

FMI is an attractive proposition for industry due to its simple interface and increasing number of tools that support FMU generation[4], however FMI has certain limitations for software engineers. FMI was originally designed for co-simulation of CT models of vehicle components and because of this, it lacks abstractions to support networked, discrete-event (DE) controllers. In this paper we present a possible solution to overcome these limitations, without extending the FMI standard, by introducing an FMU that explicitly models a network medium, following a approach that we call the *ether* pattern.

In the remainder of this paper, we discuss the limitations of communications modelling in FMI in Sect. 2. We describe the ether pattern and a proof-of-concept FMI implementation using VDM in Sect. 3. We present a building case study of a Fan Coil Unit (FCU) using the ether pattern in Sect. 4. Finally we present conclusions in Sect. 5.

2 Limitations of Communication Modelling in FMI

In this section we first consider the current limitations of modelling communications with FMI. For the purposes of this paper, we consider *communication* to be messages being exchanged between two or more entities (each of which is packaged as a distinct FMU), as illustrated in Fig. 1. These messages must conform to some user-specified structure, whether it be something low-level such as packet structure or something higher level such as API calls.

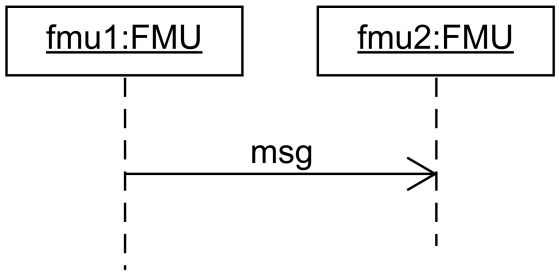


Fig. 1. Communication between two FMUs.

Before proceeding, it is worth explaining why we reason about communication *between* FMUs. The VDM-RT notation has native support for reasoning about communication using constructs like the `BUS` class and implicit remote operation calls [9, 12]. However, it is not currently possible to export only parts of a VDM model (such as classes deployed to a particular CPU) as an FMU and maintain communications. The only way to export a VDM-RT model is to export the entire model including the **`system`**

[4] A list of FMI compatible tools can be found here: `http://fmi-standard.org/tools/`

class. As such, the distribution of entities to CPUs is not visible at the FMI level. In a pure FMI setting, it is only possible to leverage the communication reasoning abilities of VDM-RT (or other modelling notations) to analyse intra-model communication. Thus, if one is interested in communication between models/FMUs, it is necessary to model the communications at the FMI level in some way.

The largest issue affecting FMI communication modelling is a fundamental one in the design of FMI— it only supports the exchange of continuous signals. This is related to the purpose of FMI, providing a means to connect simulation tools of different physical domains, all of them continuous. As such, the features and capabilities of the standard are geared towards the exchange of physical signals. The communications that we are interested in modelling are discrete phenomena: message pairs, packets, etc. However, at the multi-model level, all connections between FMUs provide continuous exchange of data. At every time in the simulation there is a value for each signal (even if that value is zero). To model concepts such as the sending and receiving of discrete-event messages, we must do so inside the models used to generate the FMU themselves. This is unintuitive as communication is primarily an inter-FMU phenomenon.

Modelling of communication can be distributed and embedded across the various FMUs, but this brings its own issues. Firstly, it will pollute the various models with communication concepts (such as message delays, response handling, etc.). Secondly, it makes it difficult to reason about communication since it is distributed across various models and hidden at the co-simulation level, where we can only observe the values of exchanged signals. We need an explicit model of the communication medium to enable reasoning. As we will see in Sect. 3, the ether pattern provides one way to do this. Another fundamental issue is that currently FMI connections are one-to-one (direct connections between two FMUs). When modelling communications, we are often interested in modelling many entities communicating between each other. This leads us to very large connection mappings between FMUs that become hard to maintain.

Assuming one has decided to model communications inside an FMU, either using the ether or distributing the communications, there are still some issues to consider, most of which have to do with the limited amount of data types supported by FMI. FMI does not support structured data types (only primitive types such as integers, reals or Booleans). However, structured data is important for modelling both low-level concepts (receiver identifier, timestamp, message data) and high-level concepts (API endpoints, payloads). Even if we model these aspects using structured data types in each constituent model, without structured data types in FMI, these aspects cannot be easily lifted to the co-simulation level.

There are two workarounds to achieve structured data types in FMI co-simulations. The first is use a collection of interdependent ports to represent each part of the data, the second is to encode/decode the structured data using strings. Both approaches have limitations. For the multi-port approach, the interdependence between the ports cannot be captured in the FMI standard, so it must be disseminated manually to the team members which leads to potential coordination issues. In addition, it also scales poorly, as complex messages will require several additional ports per FMU, which increases the complexity and effort of connecting FMUs to set up co-simulation. A high number of ports may also degrade co-simulation performance when compared to an approach using a single port.

The string-based approach scales better as everything is contained in one signal (per one-way communication channel). But this approach also has coordination issues. Namely, it is essential to share the encoding and decoding between all the models. This can be simplified by adopting a standard notation such as ASN.1 (which includes a set of Generic String Encoding Rules) as in Fabbri et al. [2]. However, the encoding and decoding functionality must still be implemented in all models. Since FMI is necessary to couple these models, they are unlikely to be able to reuse each other's functionality which will lead to duplicate implementations. Furthermore, coding and decoding messages is the kind of low-level task that we wish to avoid when modelling. This approach also requires some built-in support for strings from the modelling tools (both in terms of exporting FMU with string ports, and processing and manipulating strings within the model).

Overall, support for communication modelling in FMI is rather limited and we must make trade-off decisions between several approaches, each with their own flaws. Each approach introduces additional failure points, particularly in terms of team synchronisation. These failures can affect the co-simulations negatively.

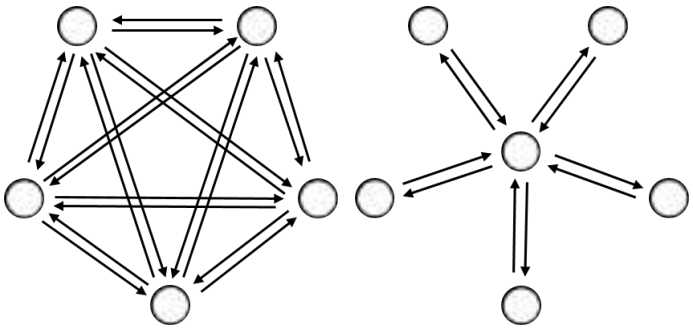
3 Modelling Networked Controllers in FMI

In this section, we address the problem of modelling networked controllers in a multi-model context. The previous section outlined the difficulties and proposed various solutions, and here we report on initial experience using the following of the solutions proposed above: using the *ether pattern* to introduce an explicit model of communication medium to the multi-model; and using string ports to marshal structured data between FMUs. In the remainder of this section we describe the ether pattern in an FMI context, report on an initial proof-of-concept ether FMU using VDM, and finally consider limitations of this FMU and identify next steps.

3.1 The Ether Pattern in FMI

When modelling and designing distributed controllers, it is necessary to model communications between controllers as well. While controller FMUs can be connected directly to each other for co-simulation, this quickly becomes unwieldy due to the number of connections increasing exponentially. For example, consider the case of five controllers depicted in Fig. 2(a). In order to connect each controller together, 20 connections are needed (i.e. for a complete bidirected graph). Even with automatic generation of multi-model configurations, this is in general not a feasible solution.

Through employing the ether pattern, a representation of an abstract communications medium is introduced (called the *ether*). This pattern was described initially in a collaborative modelling context using the Crescendo technology [5]. In an FMI setting (using the INTO-CPS technology, for example), the ether takes the form of an FMU that is connected to each controller that handles message-passing between them. This reduces the number of connections needed, particularly for large numbers of controllers such as swarms. For five controllers, only 10 connections are needed, as shown in Fig. 2(b).



(a) Topology of five controllers connected to each other

(b) Topology of five controllers connected via a shared medium

Fig. 2. Connecting FMUs directly and via a shared medium

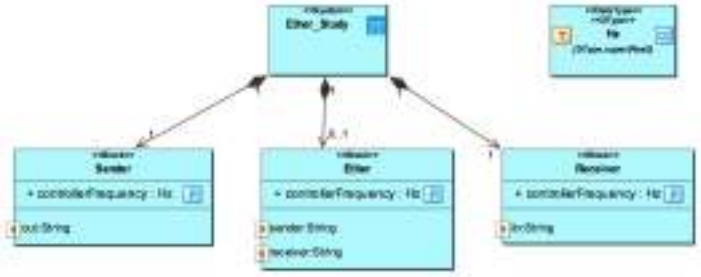
We consider a simple producer/consumer multi-model, in which a *producer* FMU sends structured data via an *ether* FMU to a *consumer* FMU. Fig. 3 shows two diagrams of this example, using views from the INTO-CPS SysML profile. Fig. 3(a) is an Architecture Structure Diagram (ASD) that shows the static structure of the system in terms of FMUs and their ports. The sender has a single output port, and the consumer has a single input port. The ether sits in between and has a corresponding port for each of these. When following the ether pattern in FMI, the ether FMU will need additional ports for new each FMU connected to it. Fig. 3(b) shows the dynamic structure of the example, with the producer connected to the consumer via the ether. This view can be used in the INTO-CPS tool chain to configure co-simulations.

3.2 A Proof-of-Concept Ether FMU using VDM

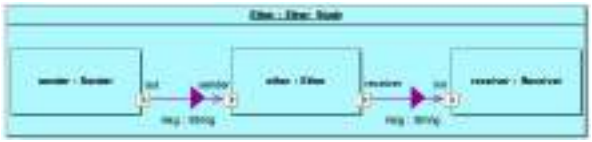
A demonstration of the ether pattern in FMI was produced by implementing the producer/consumer example above using VDM/Overture for all three FMUs (`Sender`, `Receiver` and `Ether`). The INTO-CPS technology was used to combine these FMUs into a multi-model, which is described in detail in the INTO-CPS Examples Compendium [11] and available via the INTO-CPS tool as *Case Study: Ether*.

As described in Sect. 2, connections between FMUs are typically numerical or Boolean types. This works well for modelling of discrete-time (DT) controllers and continuous-time (CT) physical models, however in a cyber-physical setting where we wish to model complex, networked controllers, it is necessary that controllers can communicate with each other using data types that are not part of the FMI specification.

In order to pass structured data between the VDM FMUs (representing the produced and consumed data), string ports were used and the data marshaled to/from their string representations within each VDM model. The Overture FMU export plug-in that enables FMU generation supports string ports, which are part of the FMI standard but not supported by all FMI-compatible tools. To encode/decone the data, the `VDMUtil` standard library was used. This can generate ASCII representation of data and parse ASCII representation back into VDM types. Using this approach it was possible to define



(a) Architecture Structure Diagram (ASD) showing static FMU structure



(b) Connection Diagram (CD) showing dynamic FMU structure

Fig. 3. SysML diagrams describing a producer/consumer multi-model that follows the ether pattern

a simple broadcast ether that receives message strings on its input channel(s) and sends them to its output channel(s). In the producer/consumer example, the three FMUs have the following roles:

Sender Generates random 3-tuple messages of type **`real * real * real`**, encodes them as strings using the `VDMUtil` library and puts them on its output.

Receiver Receives strings on its input port and tries to convert them to single messages of type **`real * real * real`** or to a sequence of messages of type `seq of (real * real * real)`.

Ether Has an input port and output port, each assigned a unique identifer, i.e. as a **`map`** `Id` **`to`** `StringPort`. It also has a mapping of input to output ports as a set of pairs: **`set of`** `(Id * Id)`. It has a list that holds messages for each output destination, because multiple messages might arrive for one destination. It gathers messages from each input and passes them to the outputs defined in the mapping.

In this simple example the sender and receiver are asymmetrical, but in more complicated examples controllers can be both senders and receivers by implementing both of the behaviours described above. This example served as a baseline for the industrial case study described in Sect. 4 which also follows the ether pattern.

3.3 Going Beyond the Basic Ether FMU

The proof-of-concept ether FMU presented above is basic. In each update cycle, it passes values from its input variables to their respective output variables. This essentially replicates the shared variable style of direct FMU-FMU connections. It also works as

a broadcast medium, so message senders are not identified. While it demonstrates the feasibility of using the ether pattern for modelling networked controllers in FMI, there are a number of areas in which experience is needed before the ether pattern can be fully understood in this context:

Addressing This basic ether is a broadcast medium, so messages are not tagged or addressed to any specific receiver. A realistic communications medium should allow this. The basic ether does however assign an identifier to each channel however, so adding addressing is primarily a matter of adding functionality for FMUs to discover their own identifiers and those of others connected on the ether.

Timing The ether FMU in the above has a naive approach to timing, it simply passes on data that it finds on its input channels. The relative update speeds of the FMUs will affect messages, for example if a value is not read by the receiver before it is changed, then that value is lost. The introduction of an intermediate FMU means that an extra update cycle is required to pass values from sender to ether and ether to receiver. These issues don't affect the examples in this paper as the messages are high level and not time-critical, since the control loop is slower than communication speed. However if lower-level details must be simulated then a more complicated ether will be required. Useful queueing and timing principles in a co-simulation setting are covered by Verhoef [12]; in the extreme a dedicated network simulator might be required[5].

Quality of Service Realistic communications modelling will require consideration of Quality of Service. For example senders might need to wait to access a medium, or resend messages in the event of collisions. Low-level communications modelling would also require consideration of Maximum Transmission Unit (MTU), the need to break up messages, and the problems of delay, loss and repetition. By controlling for problems introduced by the nature of co-simulation, any reduction in performance of the multi-model can be attributed to the realistic behaviour introduced intentionally into the model of communications.

Reusability We do not claim that ether presented above is general purpose, however it can form the basis for work in that direction. A general purpose FMU would ideally implement the features above and be configurable for various scenarios. Libraries could be created for marshalling data to and from string representations (such as ASN.1 as suggested above).

4 Case Study: Fan Coil Unit

In this section we present an application of the ether pattern to the model-based development of an HVAC system using the INTO-CPS multi-modelling technology [10]. The complete case study covers the modeling and analysis of energy consumption and comfort levels for an HVAC system that controls the temperature of connected rooms or areas inside a building as shown in Fig. 4. It contains models of several concepts including physical rooms and air flow; Fan Coil Units (FCUs); supervisory control of

[5] Example network simulators include OpNet `www.riverbed.com/` and the open-source NS2 `www.isi.edu/nsnam/ns/`.

FCUs; communications between master-slave FCUs; communications between FCUs and a supervisor; air-pipe connections between FCUs and Air-Handling Units (AHUs); and water pipe connections between FCUs and Heat Pump Units (HPUs). For this paper we focus on the communication between FCU controllers and a supervisor.

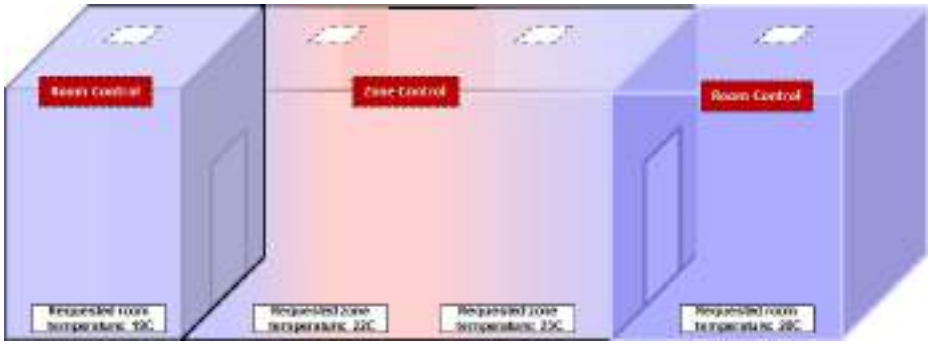


Fig. 4. Rooms and Zone level schematic for the case study

The relevant part of the control strategy for the system is regulation of FCU operations and supervision of FCUs. User input is taken into account from room or zone thermostats and is compared with current room air temperature sensed by the FCUs, triggering the FCUs to take a series of actions to reach the desired temperature by regulating the air flow (using its fan), and regulating the water pipe valves to control the cooled or heated air. The supervisor oversees and coordinates the behaviour of the FCUs and may override local FCU decisions.

In terms of the ether pattern, there are two relevant entities to be modelled:

- FCU controllers modelled in MATLAB Simulink
- Supervisor modelled in VDM-RT

We briefly outline the functionality of both entities. The primary task of the FCU controller is to run a PI loop to enable the FCU to control the room temperature. The FCU controllers also support a master-slave behaviour. Broadly speaking, the master overrides the set point of its slaves to ensure more consistent control over a zone. An FCU must be aware of its role as master or slave and work from the correct set point accordingly. The role of the supervisor is to assign an FCU as master (and possibly reassign it, if faults are detected), to distribute set points from masters to slaves, and to ensure that any set point in the FCUs is within certain maximum and minimum bounds.

To enable the supervisor activity, the FCU controllers expose an API for the supervisor so that it may control them using a higher level set of instructions. Namely, FCU controllers must provide getters for the set point and setters for role (master or slave) and the set point of the master.

The first iteration of the case study had a single controller. In the second iteration of the case study, multiple controllers were introduced and these were modelled entirely

within a single VDM-RT model. The supervisor and the FCU controllers (including the PI loops) were modelled as separate classes and deployed to different `CPU` objects in the model. This enabled the use of the native communication facilities of VDM-RT. In the third iteration of the case study, the FCU controller model was replaced with a Simulink model, due to performance of the PI controllers. The supervisor remained in the VDM model. As such, it was necessary to connect four Simulink FMUs and a VDM model.

We adopted the ether pattern (basing our implementation on the example described in the previous section), centralising all communication in the VDM model, but adapting the pattern to also include supervisory behaviour within the ether model. Our motivation for embedding the supervisor in the ether is to reduce the amount of FMI communications. In addition, we are not particularly interested in reasoning about communication details, since their impact on temperature variation in this use case is minimal [1]. In spite of this, we found the ether to be useful as a way of structuring our multi-model and centralising communication between the Simulink and VDM FMUs.

In our case study, we implement the supervision API as a protocol between the supervisor and FCU controller. This protocol is represented by three FMI signals, as shown in Fig. 5. Originally, the entire protocol was encoded in a single port using String FMI signals. However, the FMI Toolbox for MATLAB/Simulink does not support exporting FMUs with String ports, so the values were separated as follows:

- *Mode* (supervisor to FCU): An integer that indicates the operating mode of the FCU (0 for OFF, 1 for master, 2 for slave)
- *SuperGet* (FCU to supervisor): A string that encodes the measured temperature and set point of the FCU. Format: `mk_(TEMP,SETPOINT)`.
- *SuperSet* (supervisor to FCU): A real that overrides the FCU set point.

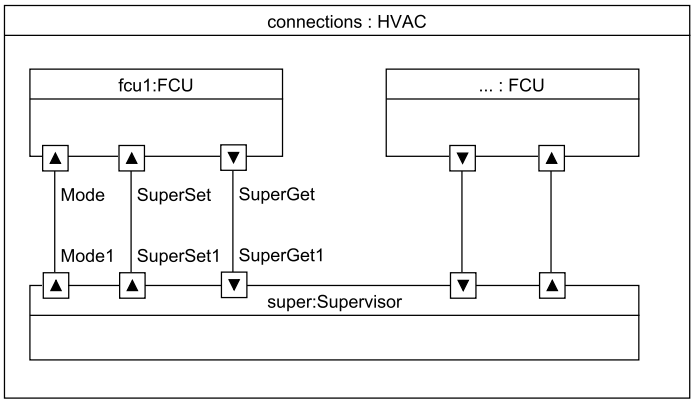


Fig. 5. FMUs communicating over FMI with the Supervisor embedded in the ether (all FCUs have the same connections).

In the Supervisor model, each FMI port is represented in the special FMI class called `HardwareInterface`, as defined in the Overture FMI export tool. Each port is then connected onto a variable of an `FCU`, which is deployed onto a separate CPU, as shown in the instantiation of the `System` class below:

```
system System

instance variables
public static super : [Supervisor] := nil;
public static sr1 : [FCU] := nil;
public static sr2 : [FCU] := nil;
public static z1 : [FCU] := nil;
public static z2 : [FCU] := nil;

public static hwi: HardwareInterface :=
  new HardwareInterface();

cpu1 : CPU := new CPU(<FP>, 5E3);
cpu2 : CPU := new CPU(<FP>, 5E3);
cpu3 : CPU := new CPU(<FP>, 5E3);
cpu4 : CPU := new CPU(<FP>, 5E3);
cpu5 : CPU := new CPU(<FP>, 5E3);
bus : BUS := new BUS (<FCFS>, 960,
  {cpu1, cpu2,cpu3,cpu4,cpu5});

ioperations
public System : () ==> System
System () ==
(
  sr1 := new FCU(1);
  sr1.primeFmi(hwi.sr1_spIn,hwi.sr1_spOut,hwi.sr1_mode);
  cpu1.deploy(sr1,"FCU_SR1");

  sr2 := new FCU(2);
  sr2.primeFmi(hwi.sr2_spIn,hwi.sr2_spOut,hwi.sr2_mode);
  cpu2.deploy(sr2,"FCU_SR2");
  -- ...
  super := new Supervisor({sr1,sr2,z1,z2});
  cpu5.deploy(super,"Supervisor");
);
end System
```

The `FCU` class is a very thin wrapper around the FMI ports which provides the supervisor with read and write access to the FMI values and hides all FMI aspects and side-effects (such as updating the FMI output port when a new set point is set). The `Supervisor` runs a continuous loop where it controls and distributes the set points of the FCUs and also performs fault detection, as shown in the abbreviated listing below.

```
ihhclass Supervisor
public control :()==>()
control()== (
  for all fcu in set {fcu | fcu in set dom fcus & fcu.isOn}
  do (
    if not fcus(fcu).faulty then checkDrift(fcu);
    distributeTemps(fcu);
  )
);

private distributeTemps : FCU ==> ()
distributeTemps(fcu) == (
  if fcu.role=<MASTER> then (
    let
      masterTemp = fcu.getMeasuredTemp(),
      nsp = getAdjustedSetPoint(fcu)
    in (
      fcu.setSetPoint(nsp);
      for all s in set fcu.getSlaves() do (
        s.setSetPoint(nsp); -- new value flows
                            -- to FMI variable as
      )                     -- side-effect
    )
  );
);
-- ...
end Supervisor
```

5 Conclusions

We have presented an approach to modelling networked control of CPSs in FMI. The approach consists of an explicit model of the network medium called the ether. We described the ether pattern and presented a basic, proof-of-concept implementation using a VDM model. We demonstrated its usage through a case study from the smart building domain, specifically the HVAC system.

The ether pattern is an effective way of modelling networked controllers within the context of FMI, without requiring changes or extensions to the standard. However, as we have seen from our case study, limitations in the FMI tools can prevent us from deploying it fully. In a setting where we are using VDM/Overture for all controllers, this is not a concern since Overture supports FMI strings. However, when using more heterogeneous tool chains, this may be a concern.

As our case study shows, the ether pattern can be used not only to model communication, but also to structure the FMUs of the co-simulation, particularly in terms of how it affects VDM modelling. By following the ether pattern when developing the multi-model, we can also easily extend it in the future with a more detailed model of the communication medium including concepts such as addressing, reliability and speed.

On the other hand, the usage of the ether does bring non-trivial burdens to the modeller and pollutes models with low-level communication concepts such as encoding and decoding of messages that might not be of interest. In addition, it requires support for strings, which is not certain for all FMI tools. While we have shown that the ether pattern can overcome some limitations in the modelling of networked controllers in FMI, further work is needed to investigate features for realistic communications modelling: sender identification, confirmation of delivery, maximum transmission unit, message timing and so on. In addition, the question of how multiple communication channels can exist together in a multi-model should be considered, i.e. should they be modelled within one ether FMU or should multiple ether FMUs be introduced. It is possible that some of these aspects can only be addressed directly at the level of the FMI itself through modifications to the standard, however a richer standard "ether FMU" and libraries could go a long way to helping multi-modellers work with networked controllers.

Acknowledgements

We wish to thank Pasquale Antonante and Hajer Saada for their assistance in modelling the case study. This work is supported by the INTO-CPS H2020 project: *Integrated Tool Chain for Model-based Design of Cyber-Physical Systems*. Funded by the European Commission-H2020, Project Number:664047.

References

1. Couto, L.D., Basagianis, S., Mady, A.E.D., Ridouane, E.H., Larsen, P.G., Hasanagic, M.: Injecting Formal Verification in FMI-based Co-Simulation of Cyber-Physical Systems. In: The 1st Workshop on Formal Co-Simulation of Cyber-Physical Systems (CoSim-CPS). Trento, Italy (September 2017)
2. Fabbri, T., Verhoef, M., Bandur, V., Perrotin, M., Tsiodras, T., Larsen, P.G.: Towards integration of Overture into TASTE. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 94–107. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
3. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)
4. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In: Proc. INCOSE Intl. Symp. on Systems Engineering. Edinburgh, Scotland (July 2016)
5. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2014), `http://link.springer.com/book/10.1007/978-3-642-54118-6`
6. 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: CPS Data Workshop. Vienna, Austria (April 2016)

7. Larsen, P.G., Fitzgerald, J., Woodcock, J., Lecomte, T.: Trustworthy Cyber-Physical Systems Engineering, chap. Chapter 8: Collaborative Modelling and Simulation for Cyber-Physical Systems. Chapman and Hall/CRC (September 2016), iSBN 9781498742450
8. Larsen, P.G., Fitzgerald, J., Woodcock, J., Nilsson, R., Gamble, C., Foster, S.: Towards semantically integrated models and tools for cyber-physical systems design. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Proc 7th Intl. Symp. Lecture Notes in Computer Science, vol. 9953, pp. 171–186. Springer International Publishing (2016)
9. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: The VDM-10 Language Manual. Tech. Rep. TR-2010-06, The Overture Open Source Initiative (April 2010)
10. Ouy, J., Lecomte, T., Christiansen, M.P., Henriksen, A.V., Green, O., Hallerstede, S., Larsen, P.G., ger, C.J., Basagiannis, S., Couto, L.D., din Mady, A.E., Ridouanne, H., Poy, H.M., Alcala, J.V., König, C., Balcu, N.: Case Studies 2, Public Version. Tech. rep., INTO-CPS Public Deliverable, D1.2a (December 2016)
11. Payne, R., Gamble, C., Pierce, K., Fitzgerald, J., Foster, S., Thule, C., Nilsson, R.: Examples Compendium 2. Tech. rep., INTO-CPS Deliverable, D3.5 (December 2016)
12. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2009)

Towards Multi-Models for Self-* Cyber-Physical Systems

Hansen Salim and John Fitzgerald

School of Computing, Newcastle University, UK
{h.salim, john.fitzgerald}@ncl.ac.uk

Abstract. The ability of systems to respond autonomously to environmental or internal change is an important facet of performance and dependability, particularly for cyber-physical systems in which networked computational and physical processes can potentially adapt to deliver continuity of service in the face of potential faults. The state of the art in the design of such systems suffers from limitations, both in terms of the capacity to model cyber and physical processes directly. We consider an approach to the modelling of cyber-physical systems that may reconfigure autonomously in accordance with internally stored policies. Specifically, we consider an adaptation of the MetaSelf framework to multi-paradigm models expressed using the INTO-CPS tool chain. We conduct a feasibility study using the benchmark UAV swarm example.

1 Introduction

The term *self-** (“self-star”) is applied to systems in which humans define general principles or rules that govern some aspect of the system, but leave it up to the system itself to make the operational decisions within the bounds dictated by such policies. The class includes systems that are, for example, self-adapting, self-organising, self-healing, self-optimising, and self-configuring systems. Examples of specific applications include highly distributed swarm-like systems, with localised decision-making, or the more centralised architectures of reconfigurable manufacturing plants.

As in the examples above, many self-* systems are cyber-physical systems (CPSs), being composed of networked computational elements that often interact with physical processes have the capacity to adapt in response to external or internal events. Such adaptation may include the dynamic alteration of operating parameters in the cyber or physical elements, as well as modification at the architectural level through the dynamic incorporation of new components, or the establishment of new connections between existing components. Such mechanisms could, for example, provide consistent and enhanced resilience to unanticipated events. The challenge then arises of undertaking model-based engineering for self-* CPSs, which involves the selection of optimal policies governing operation, with the need to maintain assurance of dependability.

The MetaSelf framework [5] was proposed to support the description of dynamically reconfigurable systems, providing a basis for delivering assurance on some properties of self-* capabilities. Originally proposed as a means of supporting the architectural description and verification of dependable self-adaptive and self-organising systems, it has only been explored in a digital (cyber) context. In this paper, we begin to examine the feasibility of engineering self-* CPSs within the INTO-CPS co-modelling and co-simulation framework. We first introduce key concepts in self-* systems and identify related work (Section 2), and we then focus on the specific MetaSelf framework (Section 3). We propose an approach to realising MetaSelf in INTO-CPS (Section 4) and explore the feasibility of this approach by means of a case study based on the UAV swarm example (Section 5). We draw conclusions in (Section 6).

2 Background: Self-* Systems and Reconfiguration

The term "Self-*" refers to systems that exhibit some autonomy [2] and has been described as one of the defining characteristics of CPSs [1], as well as a key to future CPSs [18]. Research in CPSs with self-* properties has demonstrated benefits in areas such as scalability [10], functionality [3], resilience [14,17], and optimization [6]. Examples exist in many domains. For example, it has been shown that self-healing can enable a more reliable smart grid [7,9] in which the self-* mechanism uses real-time control over power sources and switches to reconfigure the distribution network to isolate faults. A Generic Adaptation Mechanism (GAM) has been proposed for the ICT architecture of smart cars [15]. In robotics, modular self-reconfigurable units enable variable morphology [19] by rearranging their connectivity to adapt to unexpected situations, perform new objectives, or recover from faults.

Designing a self-* enabled CPS is far from trivial, especially when considering the interoperability of inherently heterogeneous CPSs. CPS designers must also consider the difference in dynamics between cyber and physical elements, as well as ensuring that autonomy can be achieved within time bounds.

One favourable approach is the use of Service-Oriented Architecture (SOA) to co-ordinate computational and physical processes. In an SOA services are provided by agents/components accompanied by a middleware framework that unifies the network [8,12,16]. SOA appears promising, but research on its application to self-* CPS is still lacking. Current approaches are described for specific domains: it is not yet clear that results are readily transferred to other CPS domains. Leitao suggests a 3-layer framework as for the engineering of adaptive complex CPSs [11], involving the integration of SOA with multi-agent systems for interoperability, using cloud computing and wireless sensor networks to enable ubiquitous environment, and self-* properties through self-organization

techniques. While the study shows the potential of the technology, the research is still at its infancy without (so far as we currently know) a proof of concept.

SOA has also been adopted by Dai et al. [4] to enable "Plug-and-play" in industrial CPSs with a case study on lighting systems. The "Plug-and-play" refers to the controller level software services that integrate loosely coupled automation systems through service discovery and orchestration. However, there is no support to describe this framework outside of industrial CPS.

3 MetaSelf

MetaSelf is an architectural framework that supports dynamically reconfigurable systems, providing a basis for gaining assurance in some self-* capabilities [5]. It exploits metadata and internally stored policies to enforce reconfiguration decisions. MetaSelf utilises SOA and, as mentioned in Section 2, the SOA approach has potential benefits in the design of CPS. MetaSelf was originally proposed as a framework for trustworthy self-adaptive and self-organising systems and in this section, we highlight the potential of adapting MetaSelf for CPSs, looking at how the characteristics of the MetaSelf architecture might be able to tackle certain CPS design challenges. We then describe the MetaSelf Development Process which will be used on our case study in Section 5.

3.1 MetaSelf Architecture

The main elements of MetaSelf include the following:

Self-Describing Components/Services/Agents. Component decoupled from their descriptions (e.g functionalities, policies, QoS) are a primary characteristic of SOA. This approach allows for interoperability as well as run-time planning for solutions to a priori unknown changes.

Acquired, Updated & Monitored Metadata. MetaSelf relies on acquiring, updating and monitoring metadata to enable self-* mechanisms. Metadata in MetaSelf can be related to individual or groups of components (e.g., QoS, availability), description of system components (relates to component interface), or the environment (relates to information received from system sensors). These metadata support decision-making for self-adaptation and/or self-organisation.

Self-* Mechanisms. Self-* mechanisms describe how and when reconfiguration is triggered. A mechanism takes metadata as a basis and adaptation actions are followed by all components. Self-* mechanisms can be implemented as *rules for self-organisation* or *dependability policies*. *Rules for*

self-organisation refer to the design of self-organisation that works bottom-up. Self-organising rules apply to each individual component and the resulting global behaviour emerges from their local interaction. *Dependability policies* refers to self-adaptive approach that works top-bottom, meaning that the run-time infrastructure evaluate its own global behaviour and change it when there is room for improvement.

Enforcement of Policies. The run-time infrastructure needs services that are able to enforce the policies as described by self-* mechanisms. These services may replace and reconfigure components and act directly on components.

Coordination/Adaptation. The list of components within the system is managed by this service. It connects and activates components to follow the automated reconfiguration actions following the rules set by the policies.

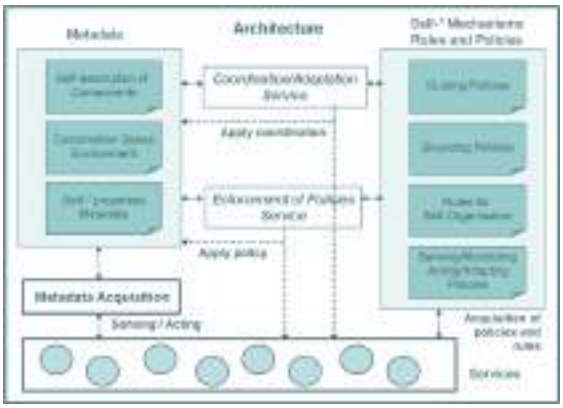


Fig. 1. MetaSelf run-time generic infrastructure

3.2 MetaSelf Development Process

The MetaSelf Development Process is a guide to designing the run-time infrastructure of MetaSelf-enabled systems. It includes four activities or phases . The *Requirement and analysis phase* describes system functionality and its self-* requirements. It should specify concrete trigger conditions of the desired reconfiguration and may include QoS. The *Design Phase D1* selects self-* mechanisms and architectural patterns along with the definition of generic rules/policies. The *Design Phase D2* includes description of policies and required metadata, component design, and any adaptation to the self-* mechanisms. The *Implementation phase* is the result of the run-time infrastructure based on the specified

design. A *Verification phase* covers activities intended to increase confidence that global properties and requirements are respected. Analysis and improvement may be identified and carried out in this phase.

4 Modelling MetaSelf-enabled Systems in INTO-CPS

INTO-CPS[1] is a tool chain that supports co-modelling of CPSs from requirements, through design, to realisation in hardware and software, enabling traceability at all stages. The INTO-CPS approach uses co-simulation based on the FMI standard, encapsulating models from different tools in FMUs, enabling inter-FMU communication and importing into hosting tools. The tool chain allows developers to build system models from diverse modelling tools and collaborate on multi-models.

As a first step towards integration of MetaSelf-enabled systems in INTO-CPS multi-models, we propose a stand-alone MetaSelf FMU that contains the main elements of the MetaSelf architecture. This FMU can be included as part of INTO-CPS multi-model design to enable reconfiguration and provide self-* characteristics. The MetaSelf FMU is a VDM-RT discrete-event (DE) model that requires metadata as its input, and outputs commands to other FMUs within the simulation for architectural reconfiguration purposes (Fig. 2). The MetaSelf FMU supports the realisation of MetaSelf architecture through VDM-RT files: `controller.vdmrt`, `component.vdmrt`, and `service.vdmrt`.

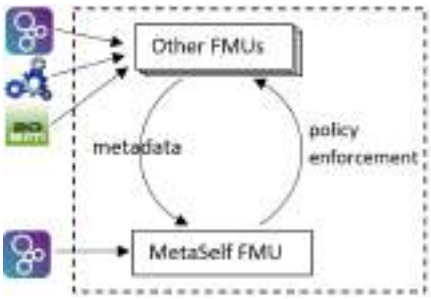


Fig. 2. Interaction between MetaSelf FMU with other FMUs in INTO-CPS multi-model

The file `component.vdmrt` defines a superclass that supports the description of generic components. The `Component` class contains methods to acquire its metadata and to enforce policies upon it that must be defined by its

[1] `http://projects.au.dk/into-cps/`

subclasses (See Fig. 3). This approach decouples the component from its description (i.e., metadata), supporting the *Self-Describing Components* element of MetaSelf.

```
class Component

operations
public getMetadata : token ==> real
getMetadata(t) == is subclass responsibility;

public enforcePolicies : Service ==> ()
enforcePolicies(c) == is subclass responsibility;

end Component
```

Fig. 3. Snippet of component class

```
system System
operations
public System : () ==> System
System () ==
 (
  --Attach connection to other FMUs to each corresponding component object
  let metadata1 =  new Sensor(hwi.metadata),
      enforcement1 = new Command(hwi.enforcement),
      myComponent1 = new Component(metadata1,enforcement1),

      metadata2 =  new Sensor(hwi.metadata),
      enforcement2 = new Command(hwi.enforcement),
      myComponent2 = new Component(metadata2,enforcement2)
  in
  (
    controller := new Controller(myComponent1, myComponent2);
    mainthread := new Thread(25, controller, myComponent1, myComponent2);
  );
  -- deploy the controller
  cpu.deploy(mainthread);
 );
end System
```

Fig. 4. FMU shared variables handled in `system.vdmrt`

The `service.vdmrt` defines a superclass that supports the description of metadata. It simply carries a token to identify the type of metadata it is describing. This superclass is especially useful when describing multiple type of metadata within a component.

The `controller.vdmrt` is where the main control loop will be handled. Metadata published by external FMUs will be acquired and saved as a local variable here. To support policy enforcement, local variables for component class objects are also defined and connected to the output port. *listOfComponents*

```
class Controller
instance variables
--SA/SO Adaptation
  public listOfComponents: set of Component;
  public registry : set of Component ;

operations
  -- constructor for Controller
  public Controller : Component * Component ==> Controller
  Controller (myComponent1, myComponent2) ==
  (
    --Component
    listOfComponents := {myComponent1,myComponent2};
    registry := {};
  );
```

Fig. 5. Snippet of controller.vdmrt (instance variable and constructor)

holds the active components within the system, each mapped with an identifier, while *registry* holds the non-active components (Fig. 5).

As described in Section 3.1, the pattern for a self-* mechanism is split into *rules for self-organisation* and *dependability policies*. A designer can define the desired self-* mechanism at *SOrules* and *dependabilityPolicies* method which will be executed at every simulation step (Fig. 6). Fig. 7 shows how each MetaSelf element is represented by the FMU.

5 A Feasibility Study: UAV Swarm

This section shows how the MetaSelf FMU approach has been applied to enhance the "Swarm of UAV" example from the INTO-CPS compendium [13] with self-* characteristics. The example concerns a group of Unmanned Aerial Vehicles (UAVs) that communicate to achieve global behaviour. Each UAV can adjust its pitch, yaw and roll to move in 3D space using rotors, and a central controller dictates the desired movements of the UAVs in the swarm.

5.1 Requirements and Analysis Phase

The goal of our swarm is to coordinate UAVs to provide wireless coverage over a designated area. UAVs serve as wireless routers and have fixed range of effective coverage radius. The UAVs should be coordinated to cover a rectangular area. For this case study, the area can be fully covered by aligning four UAVs (Fig. 8).

Each active UAV has a limited battery life and will have to leave the formation when this drops below a certain level. Our goal is to design a self-adaptive swarm using *dependability policies*. Requirements include:

```
-- POLICY
public Step : () ==> ()
Step() ==
(
  dependabilityPolicies();
  SOrules();
);

public dependabilityPolicies: () ==> ()
dependabilityPolicies() ==
(
  --Define the dependability policies here
  --E.g if components.metadata = faulty() then (replaceComponent();moveToRegistry();)
  return
);

public moveToRegistry : token ==> ()
moveToRegistry(t) ==
(
  registry := registry ++ {t |-> listOfComponents(t)};
  listOfComponents := {t} <-: listOfComponents;
);

public moveToComponentList : token ==> ()
moveToComponentList(t) ==
(
  listOfComponents := listOfComponents ++ {t |-> registry(t)};
  registry := {t} <-: registry;
);

public SOrules: () ==> ()
SOrules() == (
  --SOrules typically applies to all components and dictates its global behaviour
  --E.g if components.metadata = faulty() then replaceComponent()
  return
);
```

Fig. 6. Snippet of controller.vdmrt (Self-* mechanisms)

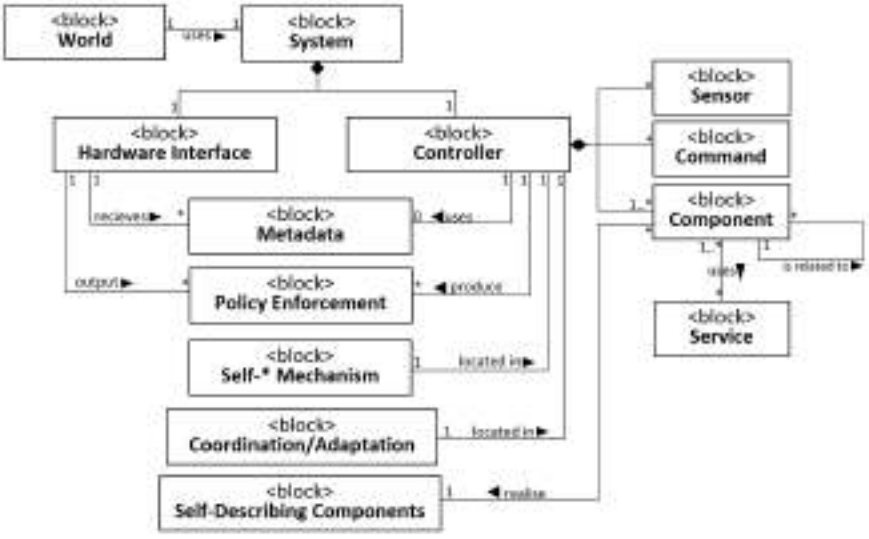


Fig. 7. MetaSelf elements in INTO-CPS FMU (SysML block definition diagram)

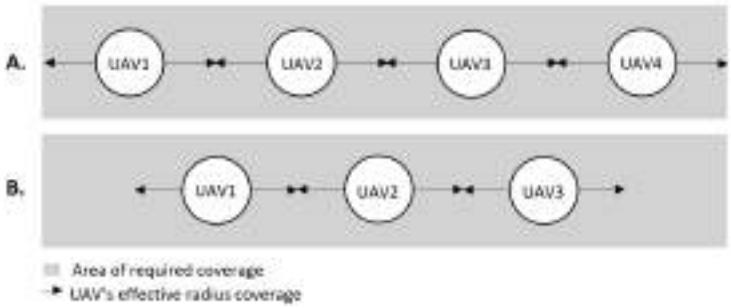


Fig. 8. UAV swarm. A: Optimum coverage with 4 UAVs. B: Example formation with limited UAVs.

1. The system could be equipped with fully charged backup UAVs. When there is a backup UAV, the system should be able to autonomously replace low battery UAV with the backup UAV.
2. When the number of available UAVs does not allow for optimum coverage (e.g., only 3 UAVs available), the system should be able to coordinate the UAVs to maximise the coverage, with highest priority being the center of the area.

For this study, we set up a scenario that starts with 5 fully charged UAVs and eventually with only 3 UAVs remaining at the end of the simulation.

5.2 Design and Implementation Phases

The multi-model has four FMUs. The MetaSelf FMU requires the battery level of each UAV as metadata, and will output coordinates to enforce the policies. In this phase, we design the SysML architecture diagram and connection diagram to show the architecture as well as the metadata flow of the swarm (Figs. 9 & 10).

To achieve the requirements fromSection 5.1, we need to decide how a self-* mechanism will be designed for the UAV swarm. The existing swarm model is made up of centrally controlled UAVs which are not equipped with sensors to monitor their environment, and therefore they are not able to make self-organising decisions. Consequently, the self-* mechanism design for this case study will follow a self-adapting pattern. Rules for self-adapting system (the *Dependability Policy*) are expressed in the `dependabilityPolicies` method. Metadata regarding the list of UAVs and battery life will be available to the policy, and it may call methods from *policy enforcement* to achieve self-adaptation. *Policy enforcement* has a method to activate (`moveToComponentList`) and

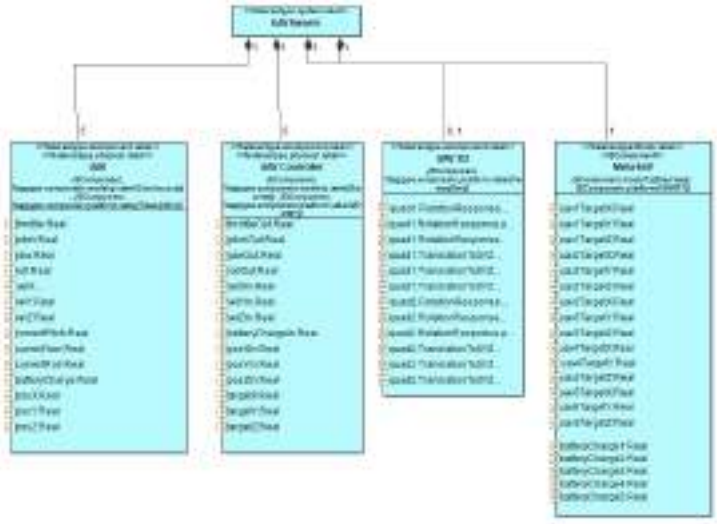


Fig. 9. SysML architecture diagram of UAV swarm

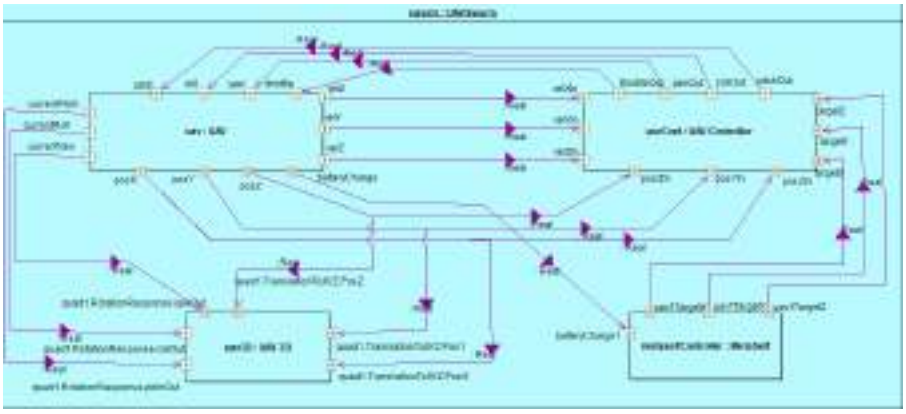


Fig. 10. SysML connection diagram of UAV swarm

deactivate (`moveToRegistry`) a component (Fig. 6) that will result in UAV replacement and method to set the UAV target coordinate (Fig. 11).

To represent the UAVs, we create a subclass UAV containing a method that provides metadata on battery life, and that may receive coordinates from the policy enforcement (Figs. 11). Figs. 12 and 13 refer to the controller class where *Dependability Policy* is described and enforced.

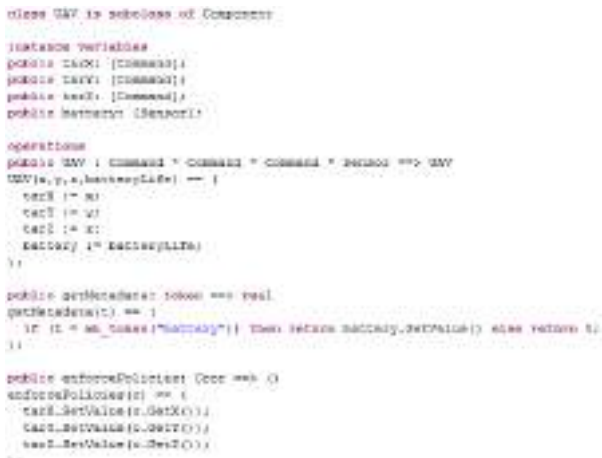


Fig. 11. Snippet of UAV class

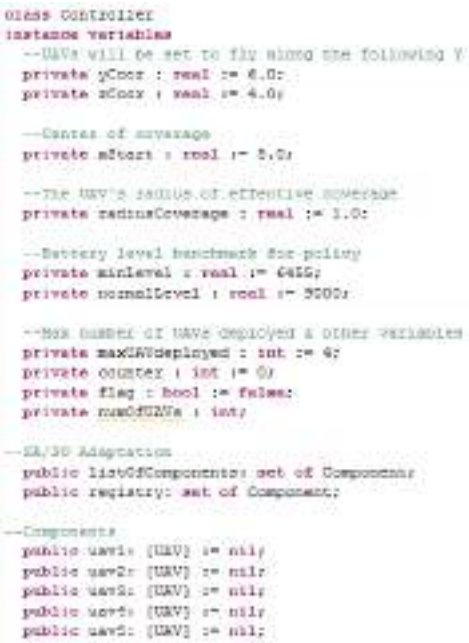


Fig. 12. Variables of controller class

Fig. 13. Self-* Mechanisms: Dependability policies to enable self-adaptive UAV swarm

5.3 Verification phase

We simulate the multi model and observe whether the MetaSelf functionalities meet the requirements stated in Section 5.1. Fig. 14 shows the actual x coordinate of all the UAVs in the swarm against simulation time. The graph shows that initially UAVs1-4 were coordinated around the x axis until around time 14 when UAV3 is replaced by UAV5. This shows that the first requirement has been met whereby backup UAVs are able to replace low battery UAVs. UAV2 eventually ran out of battery at around time 25, and the remaining UAVs spread out accordingly to maximize coverage area, suggesting that the UAVs can adapt to the number available, meeting the second requirement.

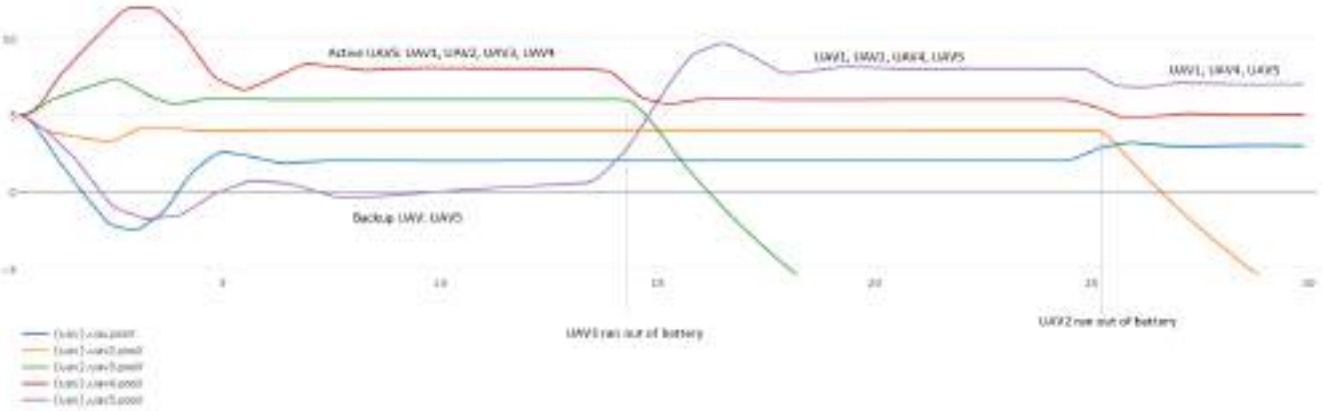


Fig. 14. INTO-CPS UAV swarm simulation graph showing the actual x coordinate of all the UAVs

6 Conclusions

Here we discuss the feasibility of our approach in addressing several challenges in the design of self-* CPSs as briefly mentioned in section 2: *Cyber and Physical integration*, *Heterogeneous components*, *Level of Autonomy*, and *Real time*.

In our study, the elements that enable self-* are discrete-event processes that receive metadata and perform physical reconfigurations; the challenge lies in capturing both discrete and continuous dynamics. Our approach addresses this by separating the design of discrete and continuous processes into separate models. This means that, provided the designer is able to create a multi-model that captures the properties of physical processes into metadata, the discrete-event policies can describe appropriate reconfiguration actions.

In our example, the swarm is homogeneous. However, we suggest that our approach can readily support heterogeneous structures. Both Metaself and INTO-CPS follow component-based approaches specifically intended to support the design of heterogeneous systems. Their integration means that on the system level, different components can communicate as long as the metadata are properly defined. On the architecture level, different components can be designed as separate FMI-conformant communicating FMUs.

Metaself allows some autonomy, but has not yet been defined to accommodate models of machine learning (ML). The Metaself process suggests that guiding policies defined at design time will not change after deployment, but an ML-enabled CPS would likely have dynamically mutable policies. This might be achieved my adding a service layer in the infrastructure that monitors performance and modifies the guiding policy accordingly.

In our proposed approach reconfiguration actions will have to go through cyber, network and physical processes. The inability to schedule real-time performance from policy decision to execution makes this form of Metaself unsuitable for real-time CPSs. Metaself should be enhanced with a form of real-time scheduling to ensure reconfiguration in real time performances.

The applicability of our approach for self-* CPSs is limited by its lack of support for real-time operation, and further work is required to demonstrate its application to CPSs with heterogeneous elements. However, it has shown potentia as it addresses certain challenges in self-* CPS design, and we anticipate that solution to some weaknesses can be added on the existing infrastructure. Further work is also required to assess its feasibility on a wider range of CPS characteristics such as human-in-the-loop and security that are not yet addressed.

Acknowledgements We acknowledge assistance from the EC's INTO-CPS project (H2020 Project 644047) and are grateful to the Overture Workshop reviewers for their comments on our earlier draft.

References

1. Antsaklis, P.: Goals and Challenges in Cyber-Physical Systems Research Editorial of the Editor in Chief. IEEE Transactions on Automatic Control 59(12), 3117–3119 (Dec 2014)
2. Berns, A., Ghosh, S.: Dissecting Self-* Properties. In: Third IEEE Intl. Conf. on Self-Adaptive and Self-Organizing Systems. pp. 10–19 (Sept 2009)
3. Bharad, P., Lee, E.K., Pompili, D.: Towards A Reconfigurable Cyber Physical System. In: 2014 IEEE 11th Intl. Conf. on Mobile Ad Hoc and Sensor Systems. pp. 531–532 (Oct 2014)
4. Dai, W., Huang, W., Vyatkin, V.: Enabling plug-and-play software components in industrial cyber-physical systems by adopting service-oriented architecture paradigm. In: IECON 2016 - 42nd Annual Conf. IEEE Industrial Electronics Society. pp. 5253–5258 (Oct 2016)
5. Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A.: MetaSelf: An Architecture and a Development Method for Dependable Self-* Systems. In: Proc. 2010 ACM Symp. Applied Computing. pp. 457–461. SAC '10, ACM, New York, NY, USA (2010)
6. Duarte, R.P., Bouganis, C.S.: Variation-Aware Optimisation for Reconfigurable Cyber-Physical Systems, pp. 237–252. Springer International Publishing, Cham (2016)
7. Elgenedy, M.A., Massoud, A.M., Ahmed, S.: Smart grid self-healing: Functions, applications, and developments. In: 2015 First Workshop on Smart Grid and Renewable Energy (SGRE). pp. 1–6 (March 2015)
8. Feljan, A.V., Mohalik, S.K., Jayaraman, M.B., Badrinath, R.: SOA-PE: A service-oriented architecture for Planning and Execution in cyber-physical systems. In: 2015 Intl. Conf. on Smart Sensors and Systems (IC-SSS). pp. 1–6 (Dec 2015)
9. Gao, X., Ai, X.: The Application of Self-Healing Technology in Smart Grid. In: Power and Energy Engineering Conference (APPEEC), 2011 Asia-Pacific. pp. 1–4 (March 2011)
10. Jatzkowski, J., Kleinjohann, B.: Towards Self-reconfiguration of Real-time Communication within Cyber-physical Systems. Procedia Technology 15, 54 – 61 (2014), 2nd Intl. Conf. on System-Integrated Intelligence: Challenges for Product and Production Engineering
11. Leitão, P.: Towards Self-organized Service-Oriented Multi-agent Systems, pp. 41–56. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
12. Park, S.O., Do, T.H., Jeong, Y.S., Kim, S.J.: A dynamic control middleware for cyber physical systems on an IPv6-based global network. Intl. Jnl. of Communication Systems 26(6), 690–704 (2013)
13. Payne, R., Gamble, C., Pierce, K., Fitzgerald, J., Foster, S., Thule, C., Nilsson, R.: Examples Compendium 2. Tech. rep., INTO-CPS Deliverable, D3.5 (December 2016)
14. Ratasich, D., Höftberger, O., Isakovic, H., Shafique, M., Grosu, R.: A Self-Healing Framework for Building Resilient Cyber-Physical Systems. In: 2017 IEEE 20th intl. Symp. on Real-Time Distributed Computing (ISORC). pp. 133–140 (May 2017)
15. Ruiz, A., Juez, G., Schleiss, P., Weiss, G.: A safe generic adaptation mechanism for smart cars. In: 2015 IEEE 26th Intl. Symp. on Software Reliability Engineering (ISSRE). pp. 161–171 (Nov 2015)
16. Schuh, G., Pitsch, M., Rudolf, S., Karmann, W., Sommer, M.: Modular Sensor Platform for Service-oriented Cyber-Physical Systems in the European Tool Making Industry. Procedia CIRP 17, 374 – 379 (2014)
17. Seiger, R., Huber, S., Heisig, P., Assmann, U.: Enabling Self-adaptive Workflows for Cyber-physical Systems, pp. 3–17. Springer International Publishing, Cham (2016)
18. Thomson, H., Paulen, R., Reniers, M., Sonntag, C., Engell, S.: D2.4 Analysis of the State-of-the-Art and Future Challenges in Cyber-physical Systems of Systems . CPSoS Project Deliverable, CPSoS (2015)
19. Yim, M., Shen, W.M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., Chirikjian, G.S.: Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics]. IEEE Robotics Automation Magazine 14(1), 43–52 (March 2007)

Debugging Auto-Generated Code with Source Specification in Exploratory Modeling

Tomohiro Oda[1], Keijiro Araki[2], and Peter Gorm Larsen[3]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] Kyushu University (araki@ait.kyushu-u.ac.jp)
[3] Aarhus University, Department of Engineering, (pgl@eng.au.dk)

Abstract. Automated code generation is one of the most powerful tools supporting VDM. Code generators have been used for efficient and systematic realisation during the implementation phase. In the exploratory phase of the formal modeling process, automated code generation can be a powerful tool for users to validate integration of generated code with existing system modules and graphical user interfaces. This paper discusses debugging possibilities in the exploratory specification phase, and introduces a debugger for auto-generated code.

1 Introduction

Animation of formal specification languages is a powerful technique in both modeling and testing of formal specifications [13] as a lightweight use of formal methods. Animation plays an important role in archieving a common understanding between developers and clients in the exploratory phase of formal specification [17]. Dialects of the VDM-family have executable subsets supported by interpreters [10,11]. We have been expanding the possible use of VDM interpreters [11,14,15].

Automated code generation [4,7,8,16,19] is another technique to animate formal specifications. Code generators have been used in the implementation phase to produce the program source code in a systematic and efficient way. In general, interpreters have the advantage of flexibility, debugging support and handiness, and thus they are used for animating VDM specifications as prototypes.

Code generators can also be used as an animation mechanism for the exploratory modeling. Code generators, in general, provide better performance and flexibility with a Graphical User Interface (GUI) and external components, which are beneficial in the exploratory specification phase as well as in the implementation phase. Use of code generators in the exploratory specification phase typically brings the following challenges:

- The user has to manage consistency between two sources, namely the VDM source specification and the generated source code.
- The user has to be fluent in both VDM and the target programming language.
- When debugging the generated program code, the user also has to locate the erroneous counterpart in the source specification and fix it.

These challenges need to be addressed in order to make code generators more practical in the exploratory phase.

The rest of this paper is organised as follows: section 2 introduces the exploratory specification phase, and describes debugging tasks in the exploratory phase. Afterwards, section 3 explains the ViennaVDMDebugger, a debugger for auto-generated code integrated with a conventional debugger, and section 4 gives technical issues and possibility of the debugger. Finally, section 5 describes related work and section 6 provides concluding remarks.

2 Debugging Auto-Generated Code with Exploratory Specification

The formal specification phase targets the production of a rigorous specification based on a deep understanding of the target system and its application domains. However, specification engineers sometimes have only limited knowledge of the application domains at the beginning of the specification phase. The earlier stage of the specification phase involves two parallel activities, to produce the specification and to learn the application domains from domain experts. We call this stage of the formal specification phase as exploratory specification [17].

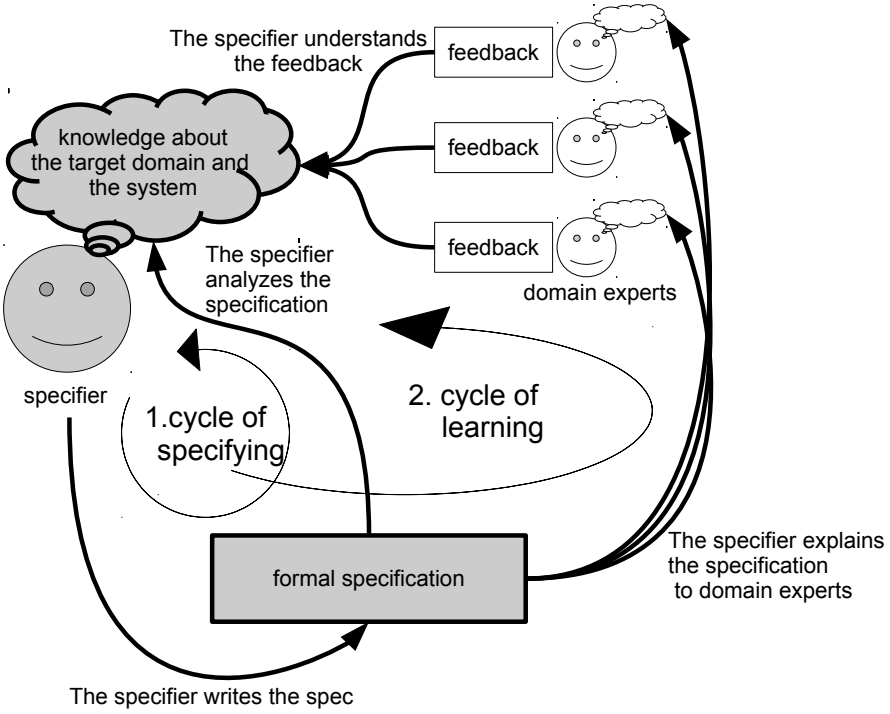


Fig. 1. The process of exploratory modeling

Table 1. Available ways of debugging animation by interpreters and code generators

Steps	Interpreters	Code generators
aware	watch variables, assertions	assertions
locating	step execution, call stacks, evaluating expressions	locating the bug in the target language and then tracing to the corresponding part in the spec
modifying	editor	editor
testing	reloading the spec and evaluating expressions	regenerating, recompiling, and restarting the code

Figure 1 illustrates the process of the exploratory specification phase. There are two cycles in the process. The first cycle starts with the specifier writing a specification based on the specifier's own knowledge of the target system and application domains. The specifier then analyses it with tools to obtain a deeper understanding. For example, a specification may suffer from an unexpected invariant violation error when animating the specification. The unexpected error is evidence that the specifier had an insufficient understanding of the target system. By debugging the specification, the specifier corrects both the specification and the understanding.

Another cycle is to learn from the domain experts. The specifier writes the specification and then explains it to the domain experts, including both application domains as well as technical domains. Animation is a strong tool to demonstrate the behaviour of the specification to domain experts who are not fluent in formal specification languages. Code generators enable the specification to be equipped with a native GUI and/or to be linked with external modules, which make the animation understandable and realistic to the application/technical domain experts.

Since the specifier needs to visualise the specifications to domain experts, the specifications must live in the larger context of GUI and networking libraries. We thus expect the specifier to understand both VDM and a programming language (Smalltalk, in our case). The goal of the exploratory specification is that the specification is validated by stakeholders, covering all functionalities and feasible to implement. The specification should be agreed by the client, application domain experts and also engineers.

In either cycle, animation often exhibits unexpected behaviour and raises errors because the specification in the exploratory phase is incomplete and error prone. The unexpected behaviour and errors are not only failures but also opportunities to understand the system and domains better. A debugger is a vehicle that provides the specifier with a new and deeper knowledge of the system. Debugging an animated specification typically involves the following steps:

1. The specifier or a domain expert becomes aware of an unexpected behaviour or an error.
2. The specifier locates the cause.
3. The specifier modifies the specification.
4. The specifier and the domain expert test the modified specification.

Available tool support varies by the method of animation. Table 1 summarises what the specifier does in each step of debugging an interpreted and code-generated animation.

Interpreters have powerful functionalities to help users to debug in each step. However, debugging a specification with a code generator is not straightforward. The specifier often has to locate the bug in the target programming language, and then trace to its counterpart in the specification. If the generated code was modified by hand, the locating task becomes more complicated; the specifier has to determine whether the unexpected behaviour or the bug was caused by the hand modification or by the source specification.

While automated code generators have advantages in performance and connectivity to external components, debugging support relies on the target language, and it is often the specifier's burden to trace the bug in the generated code to the source specification. A debugger together with a code generator that integrates two levels of debugging, namely the generated code and the source VDM specification, can reduce the burden of the specifier. The integration requires traceability from the bytes in the executable binary emitted by the target language's compiler to the syntactic fragments of the source VDM specification.

3 The ViennaVDMDebugger

In this section, we describe a debugger of the ViennaTalk environment [17,18]. ViennaTalk is a live VDM-SL development environment built on Pharo Smalltalk[4] [1]. Realisation of tool support for debugging auto-generated Smalltalk code at the source VDM-SL specification level involved two tasks: getting traceability from bytecodes to VDM-SL AST nodes and adding a UI for the source VDM-SL specification to the Smalltalk debugger. Each task will be explained in the following subsections.

3.1 Traceability

This subsection explains how traceability from bytecodes to VDM source fragments is realised. Figure 2 shows the architecture of ViennaTalk's code generators [16].

`ViennaVDMParser` parses a VDM source specification into a VDM AST (Abstract Syntax Tree), and `ViennaVDMFormatter` generates a VDM source specification from a VDM AST. `ViennaVDM2SmalltalkClass` generates Smalltalk classes and methods using `ClassBuilder` and `Compiler` that are provided in the standard library of Pharo Smalltalk. Note that ViennaTalk does not write a source file of the generated code, but creates a class instance in the Smalltalk environment.

Pharo Smalltalk provides traceability from bytecodes in a compiled method to a Smalltalk AST node. `ViennaVDM2SmalltalkClass` creates a mapping from Smalltalk AST nodes to their corresponding VDM AST nodes and stores it in the generated class. `ViennaVDM2SmalltalkClass` also stores a mapping between VDM AST nodes and VDM source ranges. Along with the Pharo Smalltalk's mapping from bytecodes to Smalltalk AST nodes, Smalltalk bytecodes can be traced to VDM AST nodes. To avoid modification to the Smalltalk `Compiler` class, the mapping from Smalltalk AST nodes to VDM AST nodes is composed of two mappings;

[4] `http://pharo.org/`

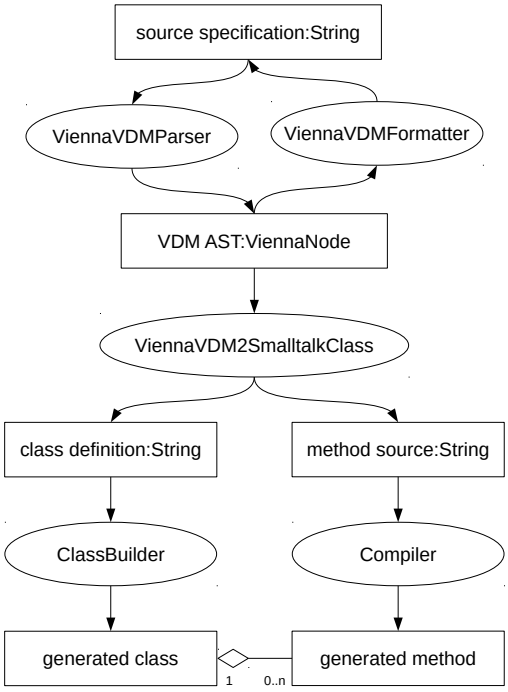


Fig. 2. Overview of ViennaTalk's code generators

one is between Smalltalk AST and Smalltalk source ranges created by the Smalltalk parser, and the other is between VDM AST and Smalltalk source ranges created by `ViennaVDM2SmalltalkClass`.

ViennaTalk defines `Vienna TracingString` as a subclass of the `String` class to implement the traceability between VDM/Smalltalk source strings and VDM AST nodes. `ViennaTracingString` holds links from its subranges to a source object.

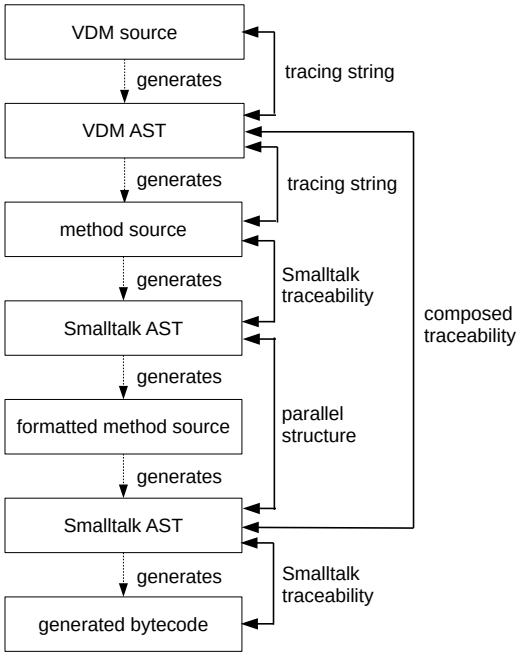


Fig. 3. Traceability composition

Figure 3 shows the composition of mappings along the chain of intermediate products from a VDM source to Smalltalk bytecodes. Traceability realised by `Vienna TracingString` is labelled as 'tracing string' in Figure 3. `ViennaVDMFormatter` generates a VDM source string as an instance of `ViennaTracingString` that has links from its subranges to the source VDM AST nodes. `ViennaVDM2Smalltalk-Class` generates a method source as an instance of `ViennaTracing String` that has links from its subranges to the source VDM AST nodes. ViennaTalk generates pretty-printed Smalltalk source code using Pharo Smalltalk's functionality via the Smalltalk AST nodes. The two Smalltalk source code items can be mapped by tracing the parallel structure of the two Smalltalk ASTs because the original source and the pretty-printed source are parsed to ASTs in the same structure. Along with Pharo Smalltalk's traceability from the Smalltalk AST nodes to corresponding Smalltalk

source subranges, ViennaTalk composes a mapping from the Smalltalk AST nodes to the VDM AST nodes. At total, an arbitrary bytecode in the generated methods can be traced up to its corresponding source VDM fragment.

`ViennaTranspiledObject` is the base class of all auto-generated Smalltalk classes. Each subclass of `ViennaTranspiledObject` manages its own specification dictionary that maps method names to VDM AST nodes in a similar way; each Smalltalk class has a method dictionary that maps method names to compiled methods. Each subclass of `ViennaTranspiledObject` has a mapping from Smalltalk AST nodes to their source VDM AST nodes. In Smalltalk, a program context is an object that can answer the object (the message receiver) and the method name (the message selector) that the program context is running in. Thus, the VDM AST node can be traced from a program context if and only if an instance of a subclass of `ViennaTranspiledObject` is in the call stack of the program context.

3.2 The UI of the Debugger

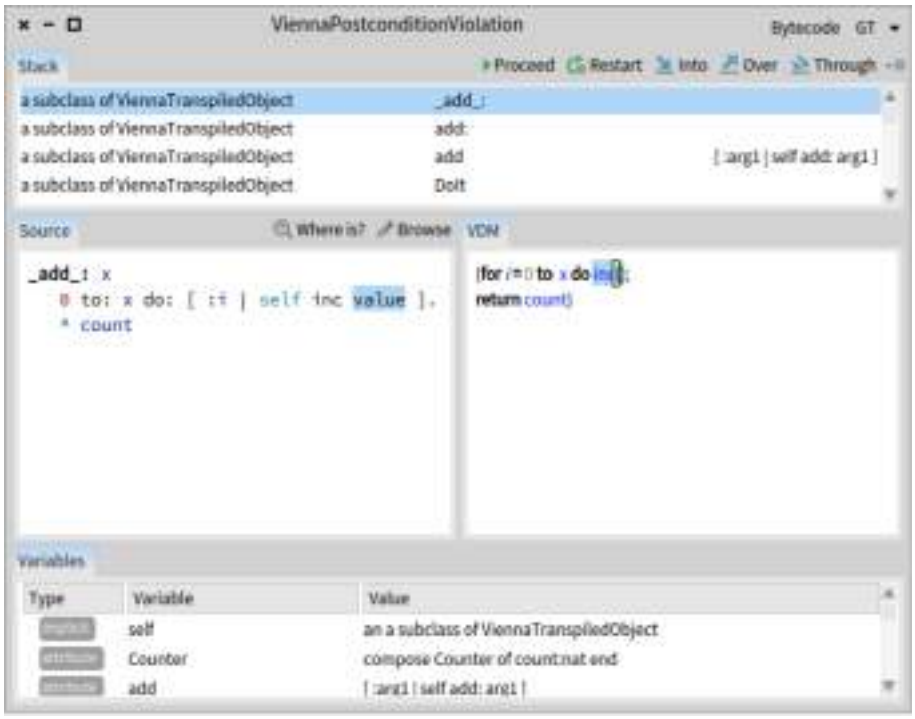


Fig. 4. A screenshot of VDMDebugger on ViennaTalk

Having traceability from program context to a VDM source fragment, ViennaTalk provides a debugger UI that displays the source VDM specification and the current executing fragment from the current program context. Figure 4 shows a screenshot of the VDMDebugger on top of ViennaTalk.

At the top, the debugger window has window operation buttons on the left, a window label `ViennaPostconditionViolation` in the middle and debugger selector buttons on the right. The window label indicates that the program execution is suspended because a post-condition is violated. The debugger selector buttons will be explained later. The second row is a series of buttons labeled `Proceed`, `Restart`, `Into`, `Over` and `Through` for step execution operations of Smalltalk program. The third row shows a call stack of the current program context where the user can select the method of concern. In the fourth row, Smalltalk source code and VDM source specification are displayed side by side, and the fragment of source that is currently to be executed is highlighted in each source. At the bottom, a list of variables and their values are shown.

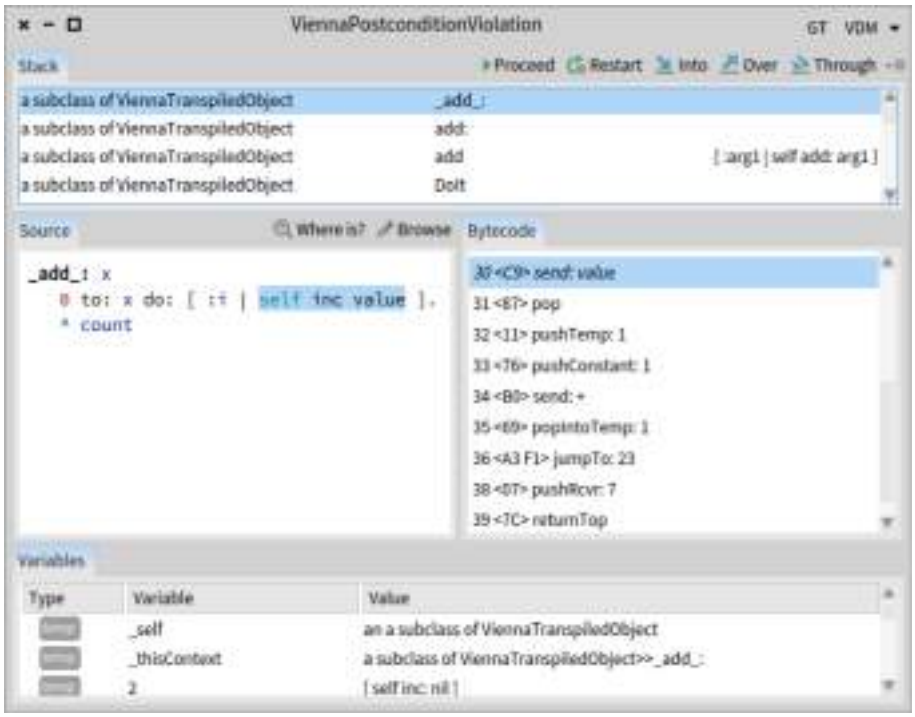


Fig. 5. A screenshot of bytecode presentation of the debugger

The debugger is integrated with the standard debugger in the Pharo Smalltalk system using a framework called a *moldable debugger* [2]. With the moldable debugger,

domain specific or purpose specific presentation can be added to the standard debugger in a pluggable manner.

The standard Pharo Smalltalk provides three presentations of the moldable debugger. The first is the generic presentation that displays the Smalltalk source code of the current program context. The second is the bytecode presentation that displays the Smalltalk source code and the bytecodes of the method being executed. The third is the unit test presentation that can visualise the difference between the expected and actual results of unit testing. ViennaTalk adds a presentation: the display of the source VDM specification of auto-generated classes as well as the Smalltalk source code.

The user can select a presentation to use via the debugger selector button on the right-top corner of the debugger window. Figure 5 shows a screenshot of bytecode presentation of the moldable debugger. By clicking on the `VDM` button at the right-top corner of the window, the debugger changes its presentation to VDMDebugger as shown in Figure 4. With these presentations, the user can see which particular part of the VDM source generated what bytecode in the generated class.

Given a program context, the VDMDebugger updates its view by the following steps:

1. The debugger scans the call stack of the program context for a context whose receiver is an instance of a subclass of `ViennaTranspiledObject`. The context is called *specification context*.
2. The debugger looks up a VDM AST node in the specification dictionary of the receiver class of the specification context. The VDM AST node is called *method spec node*.
3. The debugger generates a VDM source form the method spec node. The generated VDM source is a tracing string whose subranges are linked to their source AST nodes. The VDM source is called *method spec source*.
4. The debugger gets Smalltalk AST node from the current Program Counter (PC) value of the specification context, and looks up the VDM AST node in the AST-AST mapping of the receiver class. The VDM AST node is called *active spec node*.
5. The debugger finds the subrange of the method spec source linked to the active spec node. The subrange is called *active spec subrange*.
6. The debugger displays the active spec source and highlights the active spec subrange.

Each time when a program context proceeds, VDMDebugger shows the VDM source with the currently executed fragment highlighted along with the auto-generated Smalltalk source.

4 Discussions

In this section, we walk through a typical scenario of the debugging task on an auto-generated code to illustrate how VDMDebugger support the debugging task, and then discuss VDMDebugger in the cycle of specifying and the cycle of learning in the exploratory modeling process.

4.1 Example: debugging an erroneous counter specification

Figure 6 shows an erroneous VDM-SL specification of Counter. The bug is at the **for** statement in the definition of the `add` operation which repeats the `inc()` operation call `x + 1` times. When an expression `add(2)` is evaluated on a workspace of VDM-Browser in the transpiler mode, a post-condition violation exception is reported. VDMDebugger opens when the `Debug` button is pressed (See Figure 7). The user can also set a breakpoint at a generated method to bring the debugger.

```
state Counter of
    count : nat
init s == s = mk_Counter(0)
end

operations
    inc : () ==> nat
    inc() ==
        (count := count + 1;
        return count)
    post  count - count~ = 1;

    add : nat ==> nat
    add(x) ==
        (for i = 0 to  x do inc();
        return count)
    post  count - count~ = x;
```

Fig. 6. An erroneous VDM-SL specification of counter

ViennaTalk's transpiler generates three methods, namely `_add:`, `add:` and `add` from the `add` operation. The `_add:` method implements the body of the operation. The `add:` method checks the dynamic type of the actual arguments, checks the pre-condition, invokes the `_add:` method, checks the post-condition, and then returns the resulting value. The `add` method wraps `add:` as a closure object. The VDMDebugger opens with the source code of the `add:` method that signals the post-condition violation exception and the definition of the `add` operation with the post-condition expression highlighted to notify the post-condition expression has failed as shown in Figure 8. The execution of the `add:` method can be restarted by clicking the `Restart` button. The user clicks the `Over` button several times to step the execution forward until the invocation of the `_add:` method. The user then clicks the `Into` button to drill into the `_add:` method and repeat step execution to see how the execution proceeds and how the state changes. The user may understand the `inc()` is invoked 3 times by step execution of the `_add:` method. Figure 4 is a screenshot of VDMDebugger stepping in the `_add:` method.

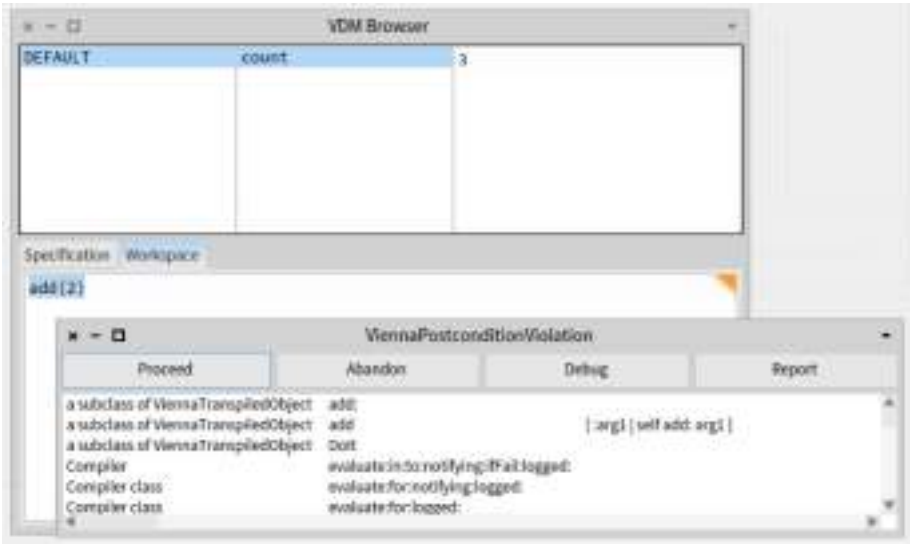


Fig. 7. A screenshot of an exception notifier raised by a post-condition violation

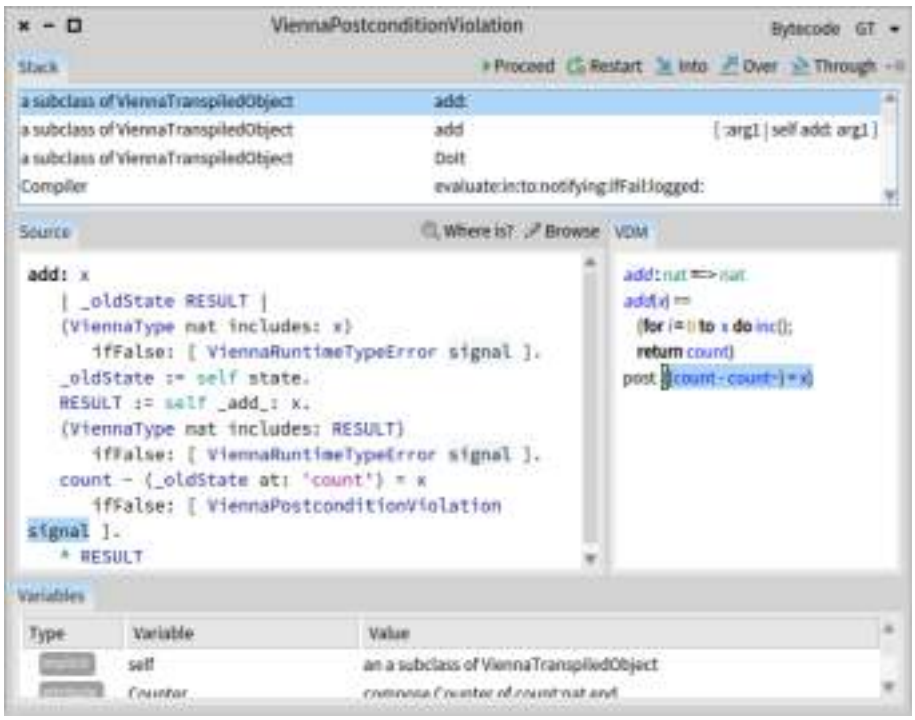


Fig. 8. A screenshot of VDMDebugger at post-condition violation

The debug execution commands such as Restart, Into and Over are based on Smalltalk's execution. Runtime type checking, pre-condition checking and post-condition checking can be debugged by step execution in the `add` method. It gives finer granularity of step executions compared to those at the VDM specification level, but it also entails extra workload for the user. For example, to reach the execution state shown in Figure 4 from the execution state shown in Figure 8, the user has to click the `Restart` button once, the `Over` button 10 times, the `Into` button once, and then the `Over` button four times. Step execution in the granularity of the VDM-SL specification is desired, and this will be the focus of future work.

4.2 VDMDebugger in the exploratory modeling

VDMDebugger is designed to support the specifiers in the cycle of specifying and the cycle of the learning presented in Figure 1. Each cycle imposes its own goal and constraints, and therefore has different requirements on the debugger.

In the cycle of specifying, the specifier is basically working alone and knows both VDM-SL and the target language, e.g. Smalltalk. The most frequent task is to understand the behaviour of the specified system rather than connectivity to a GUI or external systems. The code generator is used as a high performance animation engine. The specifier's concern is the behaviour of the specification rather than the behaviour of the Smalltalk program. The step execution operations are based on Smalltalk execution, which is a shortcoming of the current implementation of `VDMDebugger`. The specification and its state variables shown in the debugger is the minimum requirement, which the `VDMDebugger` satisfies.

In the cycle of learning, the specifier works with other stakeholders, such as application domain experts and technical leaders, who typically do not have background knowledge of VDM-SL and/or Smalltalk. The advantage of the code generator is not performance, but connectivity with GUI and external subsystems. When an unexpected behaviour such as a run-time error is observed, the specifier tries to locate the cause. The first call to make is whether the cause is in the source VDM specification or not. If the cause is in the source VDM specification, the task is to locate the bug in the VDM specification and fix it. Otherwise, the task is to locate the bug in Smalltalk code, such as GUI or connection to external systems. The specifier needs to check the VDM specification and Smalltalk code back and forth in a program context. For example, when a VDM operation is invoked by an external system via a HTTP connection and an assertion on the VDM state is violated, the cause may be either an invalid HTTP request, a bug in the code that invokes the VDM operation or a bug in the VDM specification. `ViennaVDMDebugger` provides a good support for checking both VDM specification and Smalltalk code in the same execution context.

ViennaTalk's traceability is also useful in the maintenance phase. The source VDM specification can be delivered embedded within the executable. Traceability is of key importance to formal methods. When an executable causes problematic behaviour, it is important the problematic instruction in the executable can be tracked to a particular part of the specification.

5 Related Work

VDMTools [9] has a feature for combining VDM-SL specifications with code written in C++ [5]. However, this support is only available from the interpreter and not from the generated code. On the other hand, the code generators of VDMTools are able to generate the code such that if a run-time error is encountered it is able to refer to the exact location from which the construct was generated [6].

For the Overture tool numerous features have been developed enabling a combination of VDM specifications and code written in different programming languages. The focus here has been at the interpretation level where a capability similar to that in VDMTools was presented in [14], which also enabled control to be distributed to an external application. Numerous code generators have also been developed for Overture. Of particular interest is the ability to bridge such external code between an interpretation level and a code generation level [3]. However, none of the efforts in Overture have enabled the kind of dual debugging presented in this paper.

Eiffel [12] is an Object-Oriented programming language with assertion features for Design by Contract. One can specify invariants among instance variables and pre-condition and post-condition of a method; VDM-SL also has invariants on types and states and pre-conditions and post-conditions of operations. As a general purpose programming language, Eiffel has rich and reliable library for GUI construction and network programming. The Eiffel compiler generates a native executable binary from source code. Debuggers are provided, with either GUI or command line, where the user can watch variables, set breakpoints, and execute the code by steps.

One interesting feature of Eiffel is that pieces of specifications such as invariant, pre-condition and post-condition program code are embedded within program code and they are organised by a class hierarchy. ViennaTalk also embeds pieces of specifications into auto-generated classes. The difference is that Eiffel compiler generates native executable binaries while ViennaTalk generates Smalltalk classes that resides upon Smalltalk Virtual Machine. As a result, Eiffel has advantages for producing deliverables that run efficiently on native OSs while Smalltalk has liveness that is beneficial in the exploratory modeling.

6 Concluding Remarks

Users validation and feasibility study in the exploratory specification phase is important to efficient application of lightweight formal methods. Code generators and interpreters are both mechanisms to animate formal specifications, and with the traceability of ViennaTalk, generated code can be efficiently debugged.

The foundation of ViennaTalk's traceability described in this paper relies upon the image based design of the Smalltalk environment where everything including program contexts and compiled methods are first class objects that programmers can extend in a manner. ViennaTalk pursues liveness of VDM-SL specifications. The VDMDebugger adds a further level of liveness to ViennaTalk by bringing VDM-SL source specifications into Smalltalk's live debugger.

We plan to enhance the integration between VDM-SL and the Smalltalk environment to fill the gap between formal specifications and program code. We believe such

integration will benefit the usability of both VDM-SL and Smalltalk: Smalltalk's flexibility will be made available to development of VDM-SL specifications, and rigorous construction will be made available to Smalltalk programming.

Acknowledgments

The authors would like to thank Paul Chisholm and Christoph Reichenbach for valuable feedback on drafts of this paper, and Tudor Gîrba for technical advice on the moldable tools. The authors also thank anonymous reviewers for insightful comments.

References

1. Black, A., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland (2009), http://pharobyexample.org
2. Chiş, A., Gîrba, T., Nierstrasz, O.: The moldable debugger: A framework for developing domain-specific debuggers. In: International Conference on Software Language Engineering. pp. 102–121. Springer (2014)
3. Couto, L.D., Lausdahl, K., Plat, N., Larsen, P.G., Pierce, K.: Decoupling validation UIs using Publish-Subscribe binding of instance variables in Overture. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 123–136. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
4. Diswal, S.P., Tran-Jørgensen, P.W., Larsen, P.G.: Automated Generation of C# and .NET Code Contracts from VDM-SL Models. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 32–47. Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
5. Fröhlich, B., Larsen, P.G.: Combining VDM-SL Specifications with C++ Code. In: Gaudel, M.C., Woodcock, J. (eds.) FME'96: Industrial Benefit and Advances in Formal Methods. pp. 179–194. Springer-Verlag (March 1996)
6. Group, T.V.T.: The vdm-sl to c++ code generator. Tech. rep., Kyushu University (January 2008), http://www.fmvdm.org/doc/
7. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (January 2015)
8. Kanakis, G., Larsen, P.G., Tran-Jørgensen, P.W.: Code Generation of VDM++ Concurrency. In: Proceedings of the 13th Overture Workshop. pp. 60–74. Center for Global Research in Advanced Software Science and Engineering, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan (June 2015), http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf, gRACE-TR-2015-06
9. Larsen, P.G.: Ten Years of Historical Development: "Bootstrapping" VDMTools. Journal of Universal Computer Science 7(8), 692–709 (2001)
10. Larsen, P.G., Lassen, P.B.: An Executable Subset of Meta-IV with Loose Specification. In: VDM '91: Formal Software Development Methods. VDM Europe, Springer-Verlag (March 1991)

11. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (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 (October 2011), http://dl.acm.org/citation.cfm?id=2075089.2075107, ISBN 978-3-642-24558-9
12. Meyer, B.: Object-oriented Software Construction. Prentice-Hall International (1988)
13. Nielsen, C.B.: Dynamic Reconfiguration of Distributed Systems in VDM-RT. Master's thesis, Aarhus University (December 2010)
14. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0
15. Oda, T., Araki, K., Larsen, P.G.: VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In: Plat, N., Gnesi, S. (eds.) FormaliSE 2015. pp. 33–39. In connection with ICSE 2015, Florence (May 2015)
16. Oda, T., Araki, K., Larsen, P.G.: Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 48–62. Aarhus University, Department of Engineering, Aarhus University, Department of Engineering, Cyprus (November 2016), ECE-TR-28
17. Oda, T., Araki, K., Larsen, P.G.: A formal modeling tool for exploratory modeling in software development. IEICE Transactions on Information and Systems 100(6), 1210–1217 (June 2017)
18. Oda, T., Araki, K., Larsen, P.G.: ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4. In: Proceedings of the International Workshop on Smalltalk Technologies. pp. 4:1–4:7. Prague, Czech Republic (Aug 2016)
19. 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 pp. 1–25 (2017), http://dx.doi.org/10.1007/s10009-017-0448-3

Automated Test Procedure Generation from Formal Specifications

Tomoyuki Myojin[1] and Fuyuki Ishikawa[2]

[1] Hitachi, Ltd., Japan
[2] National Institute of Informatics, Japan

Abstract. As software systems become more complicated, their test conditions also become complicated. This makes it hard for developers involved in test design to create test procedures with which to set test conditions. Thus, in this paper, we propose an automated test procedure generation method that uses state machines derived from formal specifications. We developed a tool that generates test procedures automatically from formal specifications written in VDM++. The evaluation results show that our method enables us to generate test procedures for integration testing of actual systems in practical time.

Keywords: Testing, Test Procedure, Formal Specification, VDM++, SPIN

1 Introduction

High quality software systems are developed through rigorous testing processes such as unit testing, integration testing, and system testing. As software systems have been increasing in scale, their software needs to be verified in various conditions in accordance with various use-cases in software testing processes. In particular, in the integration testing process, in which combinations of functions are checked, software needs to be tested while various combinations of conditions are considered. Therefore, these tests put a strain on testing cost and are an obstacle in software development.

Among the testing processes, the test design process is one of the most difficult. In the test design process, on the basis of function specifications of a target system, a developer makes a test procedure that contains setup test conditions that should be satisfied when executing a target function to be tested. In accordance with the procedure, a tester sets up a test condition, executes a target function to be tested, and obtains test results that shows whether the test succeed or not. In these testing processes, the more complicated the system, the more complicated the test conditions. In turn, these complicated test conditions make test procedures complicated. These complicated test procedures are difficult to develop, which causes the cost for testing to increase. To solve this problem, test procedures need to be created efficiently.

In this paper, we automate generation of a test procedure to decrease the cost for testing. To achieve this, we propose a method that automatically generates a test procedure from formal specifications written in VDM++. Several studies have reported tests being generated by using model checking [1][2]. Following these, the proposed method uses model checking to search for a test procedure to reach a state that satisfies the test

conditions from specifications. To apply VDM++ specifications to model checking, we translate VDM++ specifications into a Promela model used for model checking by using the SPIN tool. To assess the effectiveness of the proposed method, we evaluate the method by using the specifications of a network switch as a practical software system.

In Section 2, we use the example of a network switch to describe our motivation for generating test procedures. In Section 3, we describe how to generate a test procedure from formal specifications. In Section 4, we explain how our tool generates a test procedure from VDM++ [3] specifications. In Section 6, we evaluate the feasibility and practicability of the proposed method in a practical software development and describe our conclusion and future directions.

2 Motivating Example

In this paper, we use a network switch system [4] as a practical software system. A network switch is a gateway device that constructs a network between computers and has various functions such as high redundancy, stability, and virtualization. For example, the switch has physical ports connected to other devices via a cable, and some functions are realized by giving the port a special role such as setting speed. These functions have implicit or explicit specifications written either in a natural language or a formal specification language. Network switch users construct a network by setting the switch properly by referring to a user's guide written on the basis of the specifications.

In the integration testing process for a network switch system, to test a function, an environment must be constructed in which the function can be executed. For example, to test a VLAN (Virtual LAN) function, ports must be set to construct an environment in which a VLAN function can be executed. Figure 1 shows an example of a virtual network constructed on a physical network by using a VLAN function. To execute a function on the network switch, a command corresponding to a function is used. To test the constructed VLAN network, VLAN and port setting commands should be implemented in the test procedure (Figure 2). Commands to construct VLAN should be determined on the basis of functional specifications of the network switch. For example, Figure 1.1 shows functional specifications of commands written in VDM++, which is a formal specification language.

```
1  class switch
2
3  types
4  public VlanID = nat
5  public PortID = nat
6
7  instance variables
8  public vlanexist : seq of bool;
9  public switchportmode : seq of nat;
10
11 operations
12 public configure_vlan : VlanID ==> ()
13 configure_vlan(vlan) == vlanexist(vlan) := true
```

```
14 pre vlanexist(vlan) = false;
15
16 public interface_port_switchport_mode : PortID * ModeID ==> ()
17 interface_port_switchport_mode(port, mode) ==
18   switchportmode(port) := mode;
```

Listing 1.1. Formal Specifications for VLAN Function

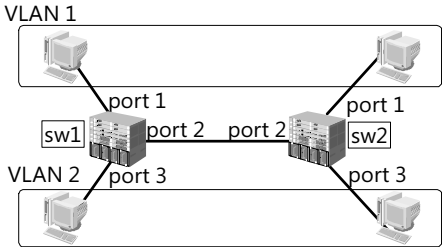


Fig. 1. Network Structure using VLAN Function

```
1 vlan 1
2 interface range port 1
3 switchport mode trunk
4 interface range port 2
5 switchport mode trunk
```

Fig. 2. Commands to construct VLAN Network

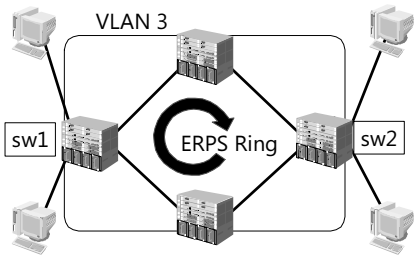


Fig. 3. Network Structure using ERPS Function

```
1 vlan 3
2  ...
3 ethernet ring g8032 ring 1
4  ...
5 ring 1 r-aps channel-vlan 3
6 ring 1 activate
```

Fig. 4. Commands to construct VLAN Network

As an example of another function, a redundant ringed network constructed on a virtual network is shown in Figure 3. This function is called ERPS (Ethernet Ring Protection Switching) [5]. To create an ERPS ring, the same commands are executed as for VLAN. To test the constructed ERPS ring, the commands to construct a virtual network need to be executed before the commands to construct an ERPS ring because ERPS needs a virtual network. Figure 4 shows the commands to construct an ERPS ring.

In this way, we need to find out how the ERPS function depends on the VLAN function and is implemented properly during the ERPS test procedure. In an actual development project, experienced developers create an appropriate test procedure on the basis of their domain knowledge. However, when a network structure and parameters for the network, i.e., test conditions, become complicated, the number or combinations

of the commands to satisfy test conditions increase. For example, to test the ERPS function in an actual development, several ERPS rings that each have a VLAN network are estimated to be necessary. This might cause development cost to increase and errors to occur in test design.

3 Proposed Method

3.1 State Machine Model based on Formal Specification

As explained above, a test procedure is created on the basis of functional specifications. To create a test procedure, developers choose and refer to necessary specifications from all specifications and combine them. Therefore, if all combinations of specifications can be enumerated, a test procedure can be derived that satisfies the desired test conditions.

In this paper, we use a state machine model to express that the system's internal state is modified by executing functions written in functional specifications. In other words, the state machine is translated from functional specifications. To translate the state machine, the specifications should be formalized. Each function has preconditions to be met before being executed. For example, the ERPS function can be executed after the VLAN function. This means that executing the VLAN function is a precondition for executing the ERPS function. Figure 5 shows state transition by ERPS functions and VLAN functions. In this state machine, transition paths from an initial state to a state satisfying test conditions are obtained by executing several functions. These transition paths represent the test procedures. Note that the shortest path is the shortest test procedure.

3.2 Searching for a Transition Path

The method to express the state machine model and search for an execution path that satisfies the test conditions on the state machine is described below. First, one function is selected from all functions that satisfy their preconditions. Second, the state is transited in accordance with a selected function. By transiting all paths exhaustively, all states that can be reached from an initial state and a path can be enumerated. By describing test conditions using temporal logic, a model checker shows counter conditions that show a path from an initial state to the state that satisfies test conditions. The path represents a test procedure.

4 Implementation

4.1 Test Procedure Generation Tool

We developed a test procedure generation tool that generates a state machine model from functional specifications and outputs a test procedure using a model checker. In this paper, a SPIN model checker [6] is adopted to handle the state machine model. Figure 6 shows components of the tool. VDM++ is adopted to describe specifications. By using VDM++, a function is described as an operation and precondition, and behavior

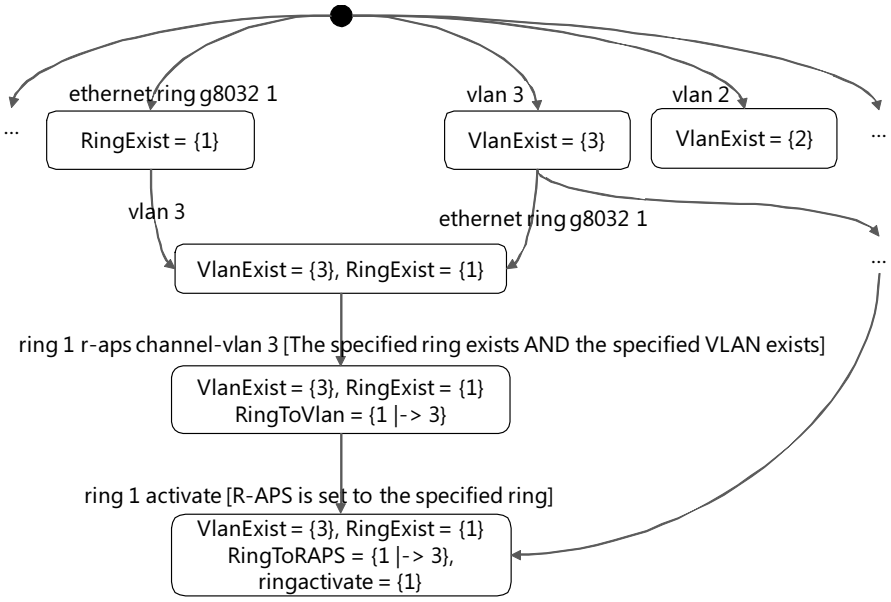


Fig. 5. State Machine Model to Search for Test Procedures

is described for each operation. The reason for using VDM++ is that it has been used in other projects in our company, so the adoption barrier of our proposed method will be lower. The tool consists of a VDM-Promela translator and a counter example processer. The VDM-Promela translator translates a VDM++ specification into a Promela model, which is processed by the SPIN model checker. The counter example processor outputs a test procedure from a counter example. To analyze VDM++ specifications, class libraries (ast.jar and interperter.jar) provided by the Overture tool are used.

4.2 Relationships between VDM++ and Promela

VDM++ specifications correspond to the Promela model translated by the proposed tool. For example, the VDM++ specifications shown in Listing 1.2 are input to the tool, and then the Promela model shown in Listing 1.3 is obtained. The relationships between VDM++ specifications and Promela model are explained below.

Types `Types` do not have direct relationships. However, a type invariant expresses a range of instance variables and operation arguments.

Values A constant described in `values` is expressed as `#define`(Listing 1.3, l.1–2).

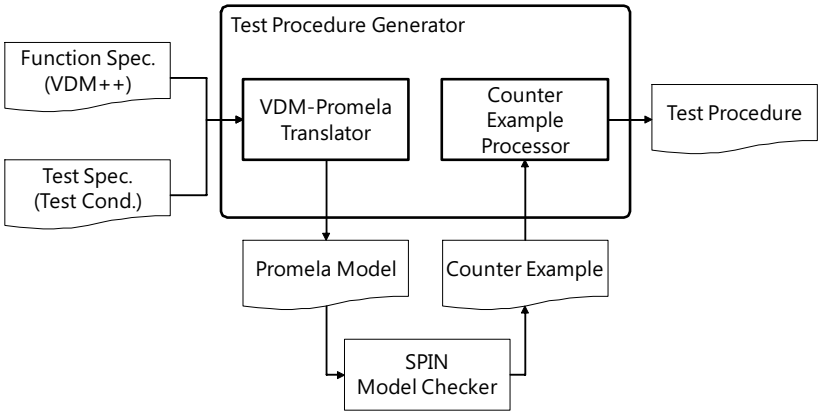


Fig. 6. Test Procedure Generation Tool

Instance variables A variable described in **`instance variables`** is expressed as a global variable (Listing 1.3, l.4). Initial values are set in initial processes (Listing 1.3, l.6–12).

Operations A behavior described in **`operations`** is expressed as an inline definition (Listing 1.3, l.17–20). A precondition in **`operations`** is also expressed with **`#define`** (Listing 1.3, l.14–15). In addition to these, **`operations`** construct the main process, which expresses the state machine model shown in section 3.1.

```
1  class switch
2
3  types
4  public VlanID = nat
5  inv vid == vid <= MAX_VLAN;
6
7  values
8  public NOT_ASSIGNED = 99;
9  public MAX_VLAN = 2;
10
11 instance variables
12 public vlanexist : seq of bool :=
13   [false | x in set {1,...,MAX_VLAN} & true ];
14
15 operations
16 public configure_vlan : VlanID ==> ()
17 configure_vlan(vlan) ==
18   vlanexist(vlan) := true
19 pre
20   vlanexist(vlan) = false;
```

```
21
22 end switch
```

Listing 1.2. Simple VDM++ Specifications of Network Switch

```
1  #define NOT_ASSIGNED 99
2  #define MAX_VLAN 2
3
4  bool vlanexist[2]
5
6  init
7  {
8   atomic {
9    byte i;
10   for(i : 0 .. 2 - 1) { vlanexist[i] = false; }
11  }
12 }
13
14 #define pre_configure_vlan(vlan) \
15   ((vlanexist[vlan] == false))
16
17 inline act_configure_vlan(vlan) \
18 {
19  vlanexist[vlan] = true;
20 }
21
22 active proctype main()
23 {
24  do
25   ::if
26    ::atomic{pre_configure_vlan(0)->act_configure_vlan(0)}
27    ::atomic{pre_configure_vlan(1)->act_configure_vlan(1)}
28   fi
29  od
30 }
```

Listing 1.3. Promela Model translated from Simple VDM++ Specifications of Network Switch

4.3 Specification Template

To generate a Promela model from VDM++ specifications, additional specifications and restrictions are provided. Additional specifications give initial values for all variables (Listing 1.2, l.12–13). The reason for giving initial values is that the initial state is needed for a state machine but was not defined in our VDM++ specifications. In this paper, initial values when the machine started are given.

Restrictions are explained below. There are two kinds of restriction. One is caused by using model checking. One problem is state explosion, which makes it impossible to search for all states. This problem is due to an increase of transitions and states. If functional specifications are described faithfully with VDM++ without abstracting details, the ranges of variables or arguments of functions increase, so their combinations increase explosively, which causes a state explosion. Thus, we apply two restrictions to VDM++ specifications. One restricts ranges of variables and arguments by type invariant (Listing 1.2, l.4–5), and the other fixes the length of a sequence in the initialization (Listing 1.2, l.12–13). By adding the above restrictions, a Promela model that does not cause a state explosion can be generated from VDM++ specifications.

The other kind of restriction is caused by the language gap between VDM++ and Promela. For example, there is a basic type named **`real`** that expresses a real number in VDM++. However, Promela does not have any types to express a real number. Therefore, to translate a **`real`** type, the specifications may need to be abstracted. Furthermore, our tool can support just a part of VDM++ language because the tool is a very simple implementation based on replacing text. For example, it supports neither set expressions nor quantified expressions. The descriptions supported on the tool are shown in Table 1.

Table 1. Supported Descriptions

Type	bool, nat, nat1, int
Expression	numeric literals, names, operators(+, −, ⋆, /, <, >, <=, >=, =, <>, and, or, not)
Statement	:= (assignment)

Considering the above additional specifications and restrictions, VDM++ specifications that can be processed with the proposed method satisfy the prescribed form shown in Listing 1.4.

```
1  class ClassName
2
3  types
4  public Tname1 = T1
5  inv pat1 == cond;
6  ...
7
8  values
9  public pat1 = exp1;
10 ...
11
12 instance variables
13 public var1 : Tname1 := exp2;
14 public var2 : seq of Tname2 :=
15   [exp3 | x in set {1,...,pat1} & true ];
16 ...
```

```
17
18 operations
19 public op1 : Tname1 * Tname2 * ... * TnameN ==> ()
20 op1(pat1, pat2, ..., patN) == exp3
21 pre precond;
22 ...
23
24 end ClassName
```

Listing 1.4. Formal Specification Template

4.4 Counter Example Processing

The counter example processor generates a test procedure on the basis of a counter example output by the SPIN model checker. To generate a test procedure, a simulation function of SPIN is used. That function traces a transition path, and the trace shows information that contains inline definitions and their argument values. The processor extracts the information from traces and output as a test procedure.

5 Result

5.1 Evaluation Method

The proposed method was evaluated to see whether it can generate test procedures in the test design process for actual development. We use the network switch for the evaluation target and assume the development of an ERPS function. The functional specifications of the network switch are described in its user's guide [4].

There are two kinds of evaluations: feasibility and practicability. The feasibility evaluation checks whether the proposed method can generate a test procedure from specifications that provide functional specifications and a test condition. Generated test procedures are validated by using the network construction procedure written in the user's guide of the network switch. The practicability evaluation, on the assumption of an integration test for ERPS, checks whether the proposed method can generate a test procedure for the large network used in actual testing for ERPS. In addition, the generation time is measured. VDM++ specifications used in the evaluation are described in Appendix, and Table 2 details the evaluation environment.

Table 2. Evaluation Environment

CPU	Intel Core i7 6600U (2.6GHz)
RAM	16 GByte
SPIN	Spin Version 6.4.5
OS	Windows 10

5.2 Feasibility

In the feasibility evaluation, a redundancy network constructed with an ERPS ring is used. The test condition for this situation is expressed as '$ringactive[0] == true$' and the linear temporal logic (LTL) formula is expressed as '$\Box!(ringactive[0] == true)$'. The test condition indicates that 'a ring is enabled.' When the SPIN model checker is used, the LTL formula must be negated, so the negation operator is added to the top of the LTL formula of the test condition. When the Promela model generated by using VDM++ specifications was checked to see whether it satisfied the test condition, a counter example was found. The test procedures based on the counter example and user guide are shown in Figures 7 and 8. The two procedures do not show any significant difference in terms of values. Therefore, the redundancy network can be feasibly constructed by using the test condition generated by the proposed method.

```
1  configure_vlan(1)
2  interface_port_switchport_mode(1,1)
3  interface_port_switchport_mode(2,1)
4  ethernet_ring_g8032(0)
5  ethernet_ring_g8032_port0_interface_port(0,2)
6  ethernet_ring_g8032_port1_interface_port(0,1)
7  ethernet_ring_g8032_rpl_owner(0,1)
8  ethernet_ring_g8032_raps_channelvlan(0,1)
9  ehternet_ring_g8032_inclusionlist_vlanids(0,0)
10 ethernet_ring_g8032_activate(0)
```

Fig. 7. Test Procedure Generated with Proposed Method

```
1  sw1(config)# vlan 4000
2  sw1(config)# interface range port 1/0/65,1/0/69
3  sw1(config-if-port-range)# switchport mode trunk
4  sw1(config)# ethernet ring g8032 ring1
5  sw1(config-erps-ring)# port0 interface port 1/0/65
6  sw1(config-erps-ring)# port1 interface port 1/0/69
7  sw1(config-erps-ring-instance)# rpl port1 owner
8  sw1(config-erps-ring-instance)# r-aps channel-vlan 4000
9  sw1(config-erps-ring-instance)# inclusion-list vlan-ids 1-1000
10 sw1(config-erps-ring-instance)# activate
```

Fig. 8. Test Procedure obtained from User's Guide

5.3 Practicability

In the practicability evaluation, a redundancy network constructed with three ERPS rings is used. Two are major rings, and the other a sub ring. The major rings can be constructed independently, but the sub ring needs to be constructed using a major ring. In accordance with actual integration testing, we construct a network with one major ring and two sub rings for the evaluation. The test condition P for one major ring and two sub rings is expressed as

$$ringactive[0] == true \;\&\&\; ringactive[1] == true \;\&\&\; ringactive[2] == true \;\&\&\; subring[2] == 0 \;\&\&\; subring[2] == 1,$$

and the LTL formula is expressed as '$\Box!P$'. This indicates that 'ring 0, ring 1, and ring 2 are enabled and ring 0 and ring 1 are sub rings of ring 2.' When with Promela model generated by VDM++ specifications was checked to see whether it can satisfy the test condition in the same way as in the feasibility evaluation, a counter example was found. Finding the counter example takes approximately 3,500 seconds and approximately 12 GBytes of memory.

Table 3. Evaluation Result

	Feasibility	Practicability
Network Redundancy	1 Major Ring	1 Major Ring and 2 Sub Rings
Elapsed Time	131 sec	3500 sec
Usage of Memory	876 MByte	11999 MByte
Counter Example	Found	Found

6 Conclusion

The feasibility evaluation showed that the proposed method can generate a test procedure that satisfies the same test conditions as a test procedure based on the user's guide. Therefore, the proposed method can make appropriate models to generate a test procedure. The proposed method is also practicable because it takes approximately 3,500 seconds to generate a test procedure, which is a practical time.

Automatically generating test procedures decreases the cost for designing test procedures. We empirically estimate that one step in a test procedure takes approximately three minutes to describe. The time includes examining what operations and related operations are needed for testing. According to this estimation, a test procedure takes approximately 4500 seconds to made manually from VDM++ specifications used in a practicability evaluation. It takes 3500 seconds with automated generation, so the proposed method decreases the time required to design a test procedure by approximately 22 percent.

The tool developed in this work has restrictions. Restrictions caused by the gap between VDM++ and Promela can be relaxed or removed by extending the proposed

method. For example, set expressions can be handled by replacing a set with an array. However, other restrictions cannot be relaxed or removed easily. For example, a type of real cannot be handled because Promela does not have a type of floating point number. To solve this problem, another solution such as an SMT (satisfiability modulo theories) solver may be effective.

In this paper, we proposed a method that automatically generates test procedures satisfying test conditions from functional specifications on the basis of formal specifications. The proposed method is based on model checking and is feasible and practicable in integration testing for actual systems.

Future work is as follows. The tool based on the proposed method offers little support for VDM++ syntax. In the future, we will try to extend the method to handle more VDM++ syntax. Moreover, the tool also needs to be validated. In this study, before developing the translation/tool, we generated a test procedure manually, and multiple authors checked that the generated result was valid. We will validate the tool by others types of check.

References

1. Dick, J., Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-Based Specifications, FME93: Industrial-Strength Formal Methods. pp. 268-284. Formal Methods Europe, Springer-Verlag (1993)
2. Ammann, P.E., Black, P.E., Majurski, W.: Using model checking to generate tests from specifications, Proceedings Second International Conference on Formal Engineering Methods (Cat.No.98EX241), Brisbane, Qld., 1998, pp. 46-54.
3. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef M.: Validated Designs For Object-oriented Systems. (2005)
4. Apresia Systems: ApresiaNP Series Users Guide. `http://www.apresia.jp/products/ent/np/manual.html`. (accessed 2016-2-10)
5. ITU-T: G.8032/Y.1344 Ethernet ring protection switching. (2015)
6. Mordechai, B.-A.: Principles of the Spin Model Checker. (2008)

A Formal Specification for Evaluation

```
1  class switch
2
3  types
4  public VlanID = nat
5  inv vid == vid <= MAX_VLAN;
6
7  public PortID = nat
8  inv p == p <= MAX_PORT;
9
10 public RingID = nat
11 inv r == r <= MAX_RING;
12
```

```
13 public ModeID = nat
14 inv m == m <= MAX_MODE;
15
16 public RplPort = nat
17 inv r == r <= MAX_RPLPORT;
18
19 values
20 public NOT_ASSIGNED = 99;
21 public MAX_VLAN = 3;
22 public MAX_PORT = 3;
23 public MAX_RING = 3;
24 public MAX_RPLPORT = 2;
25 public MAX_MODE = 2;
26
27 instance variables
28 public vlanexist : seq of bool := [false | x in set {1,...,
       MAX_VLAN} & true];
29 public switchportmode : seq of nat := [NOT_ASSIGNED | x in set
       {1,...,MAX_PORT} & true];
30 public ringexist : seq of bool := [false | x in set {1,...,
       MAX_RING} & true];
31 public ringport0 : seq of nat := [NOT_ASSIGNED | x in set
       {1,...,MAX_RING} & true];
32 public ringport1 : seq of nat := [NOT_ASSIGNED | x in set
       {1,...,MAX_RING} & true];
33 public rplowner : seq of nat := [NOT_ASSIGNED | x in set {1,...,
       MAX_RING} & true];
34 public rapschvlan : seq of nat := [NOT_ASSIGNED | x in set
       {1,...,MAX_RING} & true];
35 public inclist : seq of nat := [NOT_ASSIGNED | x in set {1,...,
       MAX_RING} & true];
36 public ringactive : seq of bool := [false | x in set {1,...,
       MAX_RING} & true];
37 public subring : seq of nat := [NOT_ASSIGNED | x in set {1,...,
       MAX_RING} & true];
38 public port_inuse_port0 : seq of nat := [NOT_ASSIGNED | x in set
        {1,...,MAX_PORT} & true];
39 public port_inuse_port1 : seq of nat := [NOT_ASSIGNED | x in set
        {1,...,MAX_PORT} & true];
40 public vlan_inuse_rapschvlan : seq of nat := [NOT_ASSIGNED | x
       in set {1,...,MAX_VLAN} & true];
41 public vlan_inuse_inclist : seq of nat := [NOT_ASSIGNED | x in
       set {1,...,MAX_VLAN} & true];
42 public ring_inuse_subring : seq of nat := [NOT_ASSIGNED | x in
       set {1,...,MAX_RING} & true];
43
44 operations
45 public configure_vlan : VlanID ==> ()
46 configure_vlan(vlan) ==
47   vlanexist(vlan) := true
```

```
48 pre
49   vlanexist(vlan) = false;
50
51 public interface_port_switchport_mode : PortID * ModeID ==> ()
52 interface_port_switchport_mode(port, mode) ==
53   switchportmode(port) := mode;
54
55 public ethernet_ring_g8032 : RingID ==> ()
56 ethernet_ring_g8032(ring) ==
57   ringexist(ring) := true
58 pre
59   ringexist(ring) = false;
60
61 public ethernet_ring_g8032_port0_interface_port : RingID *
       PortID ==> ()
62 ethernet_ring_g8032_port0_interface_port(ring, port) ==
63 (
64   ringport0(ring) := port;
65   port_inuse_port0(port) := ring
66 )
67 pre
68   ringport0(ring) = NOT_ASSIGNED and
69   port_inuse_port0(port) = NOT_ASSIGNED and
70   port_inuse_port1(port) = NOT_ASSIGNED;
71
72 public ethernet_ring_g8032_port1_interface_port : RingID *
       PortID ==> ()
73 ethernet_ring_g8032_port1_interface_port(ring, port) ==
74 (
75   ringport1(ring) := port;
76   port_inuse_port1(port) := ring
77 )
78 pre
79   ringport1(ring) = NOT_ASSIGNED and
80   port_inuse_port0(port) = NOT_ASSIGNED and
81   port_inuse_port1(port) = NOT_ASSIGNED;
82
83 public ethernet_ring_g8032_rpl_owner : RingID * RplPort ==> ()
84 ethernet_ring_g8032_rpl_owner(ring, rplport) ==
85   rplowner(ring) := rplport
86 pre
87  ringexist(ring) = true and
88  (rplport <> 0 or (rplport = 0 and ringport0(ring) <>
       NOT_ASSIGNED)) and
89  (rplport <> 1 or (rplport = 1 and ringport1(ring) <>
       NOT_ASSIGNED));
90
91 public ethernet_ring_g8032_raps_channelvlan : RingID * VlanID
       ==> ()
92 ethernet_ring_g8032_raps_channelvlan(ring, vlan) ==
```

```
93  (
94    rapschvlan(ring) := vlan;
95    vlan_inuse_rapschvlan(vlan) := ring
96  )
97  pre
98    ringexist(ring) = true and vlanexist(vlan) = true and
99    vlan_inuse_rapschvlan(vlan) = NOT_ASSIGNED and
100   vlan_inuse_inclist(vlan) = NOT_ASSIGNED;
101
102 public ehternet_ring_g8032_inclusionlist_vlanids :  RingID *
        VlanID ==> ()
103 ehternet_ring_g8032_inclusionlist_vlanids(ring, vlan) ==
104 (
105   inclist(ring) := vlan;
106   vlan_inuse_inclist(vlan) := ring
107 )
108 pre
109   ringexist(ring) = true and
110   vlanexist(vlan) = true and
111   vlan_inuse_rapschvlan(vlan) = NOT_ASSIGNED;
112
113 public ethernet_ring_g8032_activate : RingID ==> ()
114 ethernet_ring_g8032_activate(ring) ==
115  ringactive(ring) := true
116 pre
117   ringexist(ring) = true and
118   rapschvlan(ring) <> NOT_ASSIGNED and
119   rplowner(ring) <> NOT_ASSIGNED and
120   (ringport0(ring) = NOT_ASSIGNED or switchportmode(ringport0(
          ring)) = 1) and
121   (ringport1(ring) = NOT_ASSIGNED or switchportmode(ringport1(
          ring)) = 1);
122
123 public ethernet_ring_g8032_subring : RingID * RingID ==> ()
124 ethernet_ring_g8032_subring(mring, sring) ==
125 (
126   subring(mring) := sring;
127   ring_inuse_subring(sring) := mring
128 )
129 pre
130  rapschvlan(sring) <> NOT_ASSIGNED and
131  rplowner(sring) <> NOT_ASSIGNED and
132  (ringport0(sring) <> NOT_ASSIGNED or ringport1(sring) <>
        NOT_ASSIGNED ) and
133  ringactive(mring) = true and
134  ringactive(sring) = false and
135  ring_inuse_subring(sring) = NOT_ASSIGNED;
136
137 end switch
```

Automated Generation of Decision Table and Boundary Values from VDM++ Specification

Hiroki Tachiyama[1], Tetsuro Katayama[2], and Tomohiro Oda[3]

[1] University of Miyazaki, Japan tachiyama@earth.cs.miyazaki-u.ac.jp
[2] University of Miyazaki, Japan kat@cs.miyazaki-u.ac.jp
[3] Software Research Associates, Inc. Japan tomohiro@sra.co.jp

Abstract. The purpose of this research is to improve the efficiency of test process in software development using formal methods. This paper proposes both methods to generate decision tables and boundary values automatically from the specification written in the formal specification description language VDM++ to reduce the burden in designing test cases. Each method has been implemented as VDTable[1] and BWDM[2] in Java. They parse the VDM++ specification by using VDMJ[3]. VDTable treats with the parsed result as internal representation data, generates a truth table, and displays a decision table on the GUI. BWDM analyzes boundary values of the argument types in function definitions and the condition expressions in if-expressions, and outputs the result as a csv file. This paper describes the both tools, shows their results applied to simple VDM++ specifications, and discusses the improved efficiency of designing test cases.

Keywords: VDM, Software Testing, Decision Table, Test Case, Automated Generation.

1 Introduction

In the design of software development, the Decision Table [1] is used to represent the accurate logic of software. The decision table expresses the logic of software by dividing it into three terms: conditions, actions, and combination rules of conditions. It's applied widely not only at the design, but also the testing. However, it is sometimes troublesome and time consuming to manually prepare the decision table, because it is necessary to understand the target system or specification and to extract conditions and actions from specifications.

Also, the software testing plays an important role in software development. The testing involves design of test cases and design documents based on the specification, which costs time and efforts. Techniques to find bugs with fewer test cases and methods to design such test cases will reduce the cost of the testing. Boundary Value Analysis [2, 3] is generally known as a promising testing technique for this purpose.

[1] https://earth.cs.miyazaki-u.ac.jp/research/2016/vdtable_english.html
[2] https://earth.cs.miyazaki-u.ac.jp/research/2016/bwdm_english.html
[3] https://github.com/nickbattle/vdmj

Fig. 2.1. An example of Decision Table

We propose a pair of methods to generate (1) a decision table (2) and boundary values from a formal specification to improve the efficiency of the testing in the software development with VDM++. VDM (Vienna Development Method) [4, 5, 6] is one of the Formal Methods, and VDM++ [7] is the object-oriented dialect of VDM-SL that is the formal specification description language of VDM. Both methods first parse the VDM++ specification. Thereafter, the former extracts conditions and actions from the parsing data, creates truth values and then generates a decision table. The latter extracts the if-expressions and argument types of function definition, performs boundary value analysis, and finally generates test cases. Each of the proposed methods is implemented as tools in Java, namely VDTable [8, 9] and BWDM [10]. VDTable generates a decision table from VDM++ specification. BWDM generates test cases based on the boundary value analysis of the VDM++ specification. The contents of the following chapters are shown below.

Existing techniques that this work is based upon and related work is briefly introduced in Section 2.

In Section 3, two tools implemented based on the proposed methods are explained.

In Section 4, the tools are evaluated.

In Section 5, we summarize this research and show future issues.

2 Background

In this section, we describe the decision table and boundary value analysis. We also explain related research what is improve the efficiency of software testing.

2.1 Decision Table

The decision table [1] summarizes actions on inputs in a table form. It can be used at the software design and the test design. Fig. 2.1 shows a sample of decision table. It consists of three parts.

1. Condition part (1 in Fig. 2.1) describes the conditional statements in software and the possible combinations of these truth values.
2. Action part (2 in Fig. 2.1) describes the operation of the software for the combination that the truth value of the condition part can have.
3. Rule part (3 in Fig. 2.1) crosses the condition part and action part vertically. This part shows the operation of action part for the combination of each truth value of the condition part. For example, in the case of #2, it shows when an input variable *a* that $a < 5$ is True and $12 <= a$ is False, the "a < 5" (String value) is returned. The other actions are not executed.

2.2 Boundary Value Analysis

It is known from empirical studies that many software bugs exist on "boundaries" within software. Therefore, Equivalence Partitioning [11] and Boundary Value Analysis [2, 3], which are software testing methods, are used.

Equivalence partitioning is one of the test design technique, focusing on inputs and behaviors to test objects and summarizing the range of inputs in which the test objects behave in the same way as equivalence classes. The input values in the equivalence class are regarded as the same behavior, and by narrowing down the input values, the total number of test cases can be reduced. Boundary value analysis is also a test design technique focusing on the fact that program bugs are likely to exist near the boundary of equivalence classes. Boundary value analysis is also a test design technique focusing on the fact that software bugs are likely to exist near the boundary of equivalence classes. It is based on an empirical rule that an error tends to occur at the time of judging the end condition of the loop and the branching condition.

2.3 Related Work

In software development using formal methods, research to improve the efficiency of testing process has been actively conducted [12]. In this section, we briefly describe two researches on software testing using VDM specifications.

CEGTest[4] is a tool for automatically generating a decision table. The cause result graph as the input of CEGTest has to be created manually by the user directly from the VDM++ specification. In other words, in order to obtain the decision table, the user needs to understand the contents of the VDM++ specification and to extract conditions and actions, which takes time and labor. On the other hand, VDTable, the decision table generation tool we developed in this research, does not need to manually create new data for generating the decision table by using the VDM++ specification as the tool input. Therefore, compared to CEGTest, the VDTable has the advantage that the time and labor required for generating the decision table is small.

Also, Overture[5] is the IDE for supporting the development using VDM, and it has editor, syntax checker, and many other functions. The support for Combinatorial

[4] http://softest.jp/tools/CEGTest/
[5] http://overturetool.org/

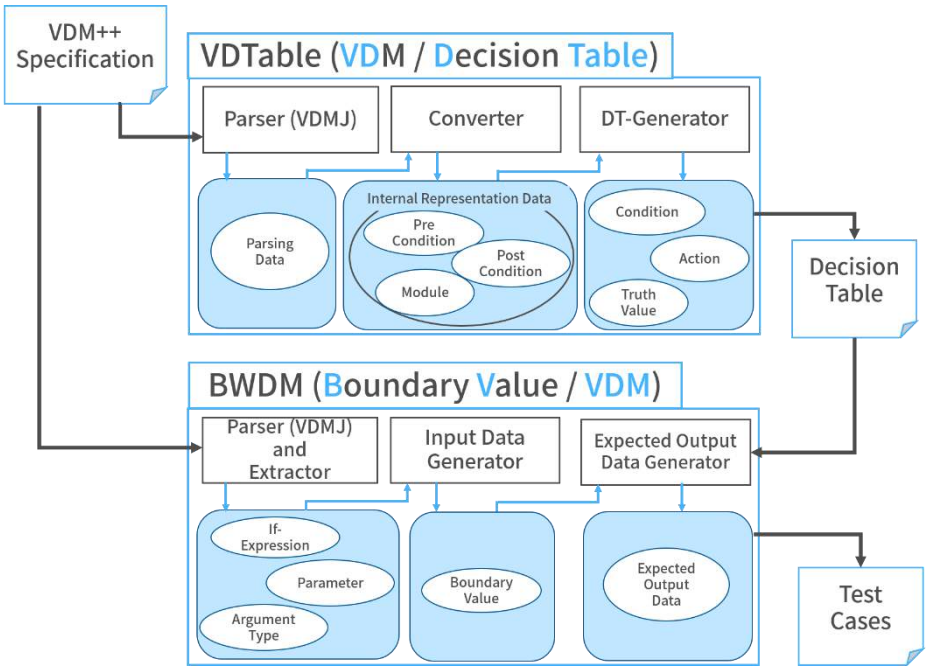


Fig. 3.1. The flow of two tools

Testing [13] is one of the function of Overture. This function enables to automatically generate test cases that is all combinations of input data what is specified by user by regular expression. Also, test is automatically conducted. Input data for this function needs to be prepared by the user. We think that if input data that is generated by BWDM was used for input data of this function, it is possible to perform more effective test against bug finding. In other words, by specifying boundary values generated by BWDM as input data, boundary value analysis and combinatorial testing can be combined and conducted.

The two researches, CEGTest and combinatorial testing, are above highly related to this research, but neither of them has been conducted to reduce the labor and time for creating test cases and decision tables by automatically generating from the VDM++ specification to improve the work efficiency at the time of the test design.

3 Proposed Methods and Implemented Tools

We describe two tools, namely VDTable and BWDM, implemented based on the proposed methods in this research. Fig. 3.1 shows the ovreview of the tools. VDTable generates a decision table from the VDM++ specification automatically, and also BWDM generates, from the decision table and the VDM++ specification, test cases with the boundary values and expected outputs. These test cases can be used for testing of the software implemented from VDM++ specifications.

3.1 VDTable

VDTable [8, 9] is a tool to automatically generate a decision table from the VDM++ specification. VDTable has a GUI to select a VDM++ specification file for input, and display a decision table, and it consists of the following three modules.

- Parser
- Converter
- DT-Generator

The processing procedure of VDTable is shown below.

1. The user runs with a VDM++ specification file, which should pass the syntax and type checking.
2. VDTable parses the given VDM++ specification into an abstract syntax tree.
3. VDTable extracts if-expressions, cases expressions, pre- conditions and post conditions from the syntax tree and generates an internal representation data of those components.
4. VDTable extracts conditions and actions from the internal representation data, and creates truth values from them, and finally composes a decision table.

The current implementation of VDTable has two limitations. Although the parser accepts valid VDM++ specifications, the current implementation of the Converter and the DT-Generator takes only account of if expressions, cases expressions, preconditions and postconditions to generate a decision table. Another limitation is that truth values of conditions and actions are created only based on the control flow of each syntax. VDTable does not always generate all the combinations of truth values.
Below we explain each module.

Parser
VDTable uses VDMJ to obtain an abstract syntax tree of the given VDM++ specification, and outputs only syntax trees of function definitions as parsing data.

Converter
This unit converts the parsing data output by Parser into internal representation data in order to facilitate extraction of conditions and actions and creation of truth values which are necessary in generating a decision table.
Fig. 3.2 shows an example of VDM++ specification, and Fig. 3.3 shows an example of internal representation data generated from the specification of Fig. 3.2. The internal representation data consists of expressions whose keyword tokens, conditions and alternatives are divided into lines, and after each occurrence of “if” and “else” tokens, a number to indicate a corresponding pair is appended.

```
class Sample
functions

  revBin2dec : seq of nat -> nat
  revBin2dec( s ) ==
     if s=[] then 0
     elseif s=[0] then 0
     elseif s=[1] then 1
     else hd s + 2*revBin2dec( tl s );

end Sample
```

Fig. 3.2. An Example of VDM++ Specification

```
if1
(s = [])
then
0
elseif
(s = [0])
then
0
elseif
(s = [1])
then
1
else1
((hd s) + (2 * revBin2dec((tl s))))
```

Fig. 3.3. Internal Representation Data

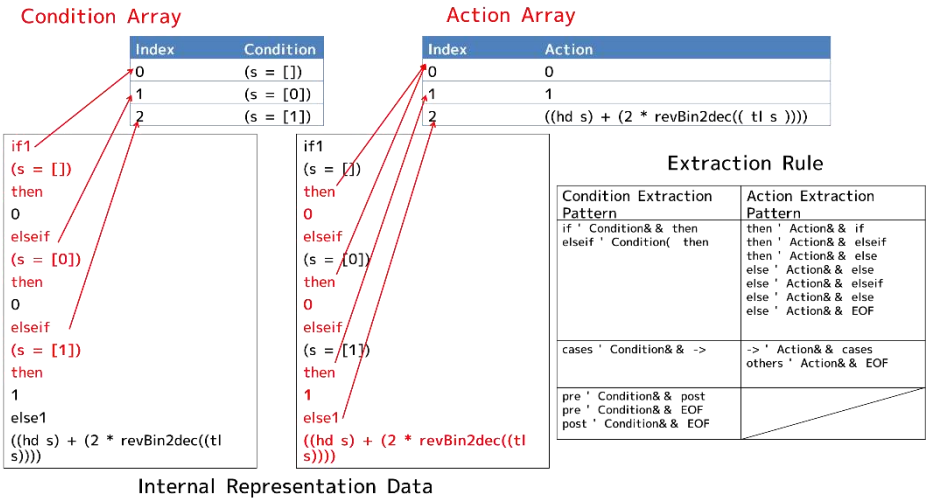


Fig. 3.4. Making of the Condition Array and the Action Array from Internal Representation Data and Extraction Rule

DT-Generator

DT-Generator is an abbreviation of Decision Table Generator. It extracts conditions that is matched with the extraction pattern of the condition from internal representation data, and stores them in a String type array (Condition Array). It also extracts actions that is matched with the extraction pattern of the actions from internal representation data, and stores them in a String type array (Action Array). Each extraction pattern is defined by Extraction Rule. Fig. 3.4 shows examples of Condition Array and Action Array generated by DT-Generator from the VDM++ specification of Fig. 3.2. Also,

Extraction Rule is shown in the Fig. 3.4 too. When DT-Generator extracts conditions and actions, CA-Table is generated. Fig. 3.5 shows the CA-Table generated from

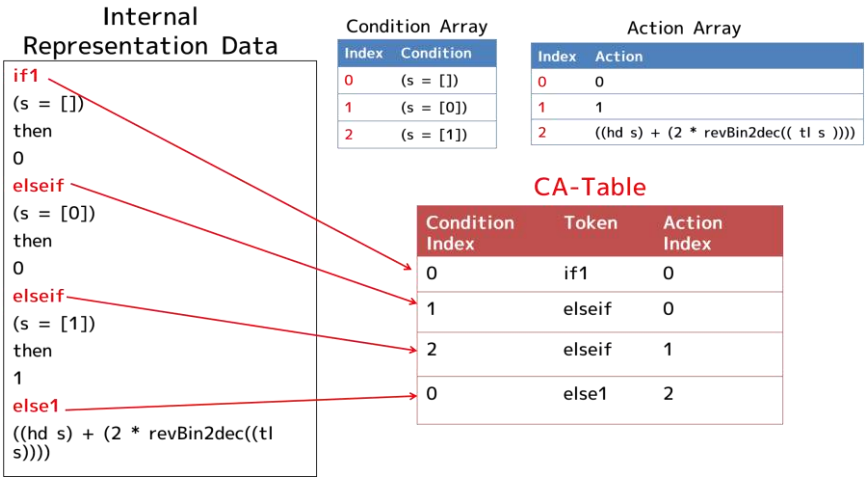


Fig. 3.5. Making of Condition Action Table

Fig. 3.4. CA-Table is an abbreviation of Condition Action Table, and it is a table that corresponds of conditions and actions. CA-Table consists of three columns: condition index, token, and action index.

This unit generates the truth value of the conditions and actions based on this CA-Table. The truth value generation procedure is shown below.

1. Create an array that is two dimensional and stores truth value.
2. Select the first column of created array.
3. Select a row of CA-Table from the first row.
4. Compare tokens and store truth value in the array,

   a. When “if”, “elseif”, “cases” are matched,
      (1) Store "Y" to the column selected for the conditional index row and "N" from the next column to the end of the column.
      (2) Store “X” in action index of selected column.
   b. When “else”, “others” are matched,
      (1) Store "N" to the column selected for the conditional index row and "-" from the next column to the end of the column.
      (2) Store “X” in action index of selected column.

5. If there are unselected rows, select the next column of the array and return to 3. When there is no, truth value is completed.

Two Dimensional Array

Number of Condition and else and others

Index of Condition				
0	Y	N	N	N
1	-	Y	N	N
2	-	-	Y	N

Index of Action				
0	X	X	-	-
1	-	-	X	-
2	-	-	-	X

Fig. 3.6. The Truth Value

Rule	#1	#2	#3	#4
Condition				
(s = [])	Y	N	N	N
(s = [0])	-	Y	N	N
(s = [1])	-	-	Y	N
Action				
0	X	X	-	-
1	-	-	X	-
((hd s) + (2 * revBin2dec((tl s))))	-	-	-	X

Fig. 3.7. The Decision Table

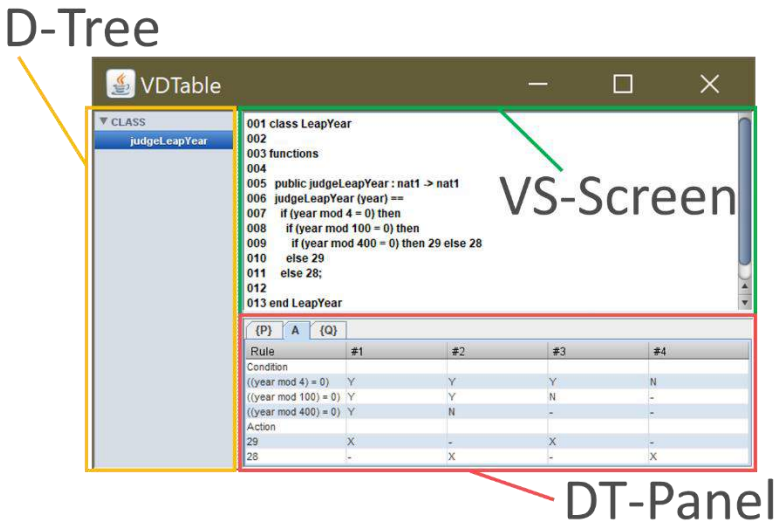


Fig. 3.8. Leap year specification

Fig. 3.6 shows a completed truth table. The finally generated decision table is shown in Fig. 3.7.

GUI parts and display

In this section, we show GUI parts of VDTable and the result of application to leap year specification in VDM++. Fig. 3.8 shows application to leap year specification. VDTable consists of the following three parts.

- VS-Screen (VDM++ Specification Screen)
  Display the VDM++ specification specified by the user along with the line number.

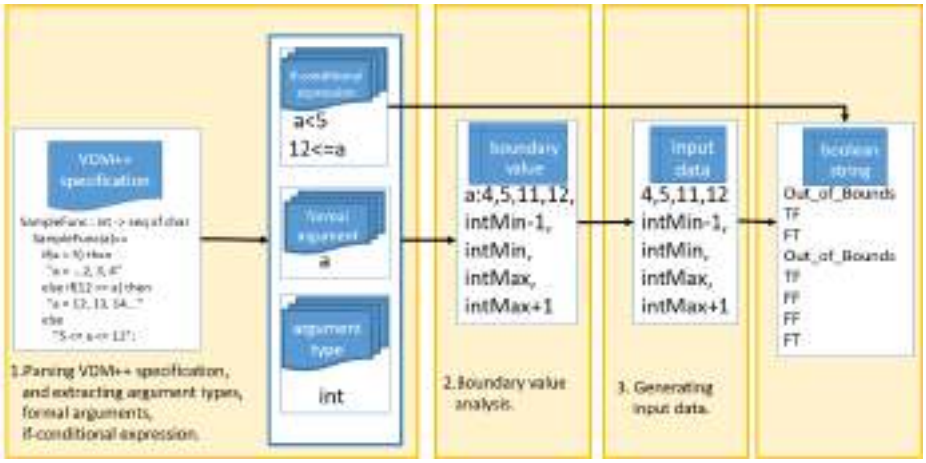


Fig. 3.9. The process of Input Data Generator

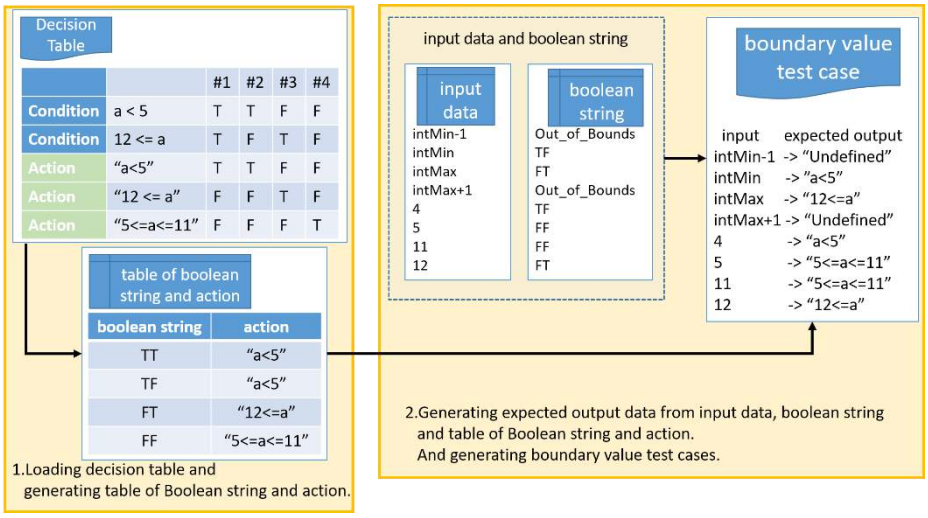


Fig. 3.10. The process of Expected Output Data Generator

- DT-Panel
  Display the decision table of the function definition or operation definition defined in the VDM++ specification. Also, by switching tabs, a decision table of modules, pre- conditions and post conditions is displayed individually.

- D-Tree
  Display the list of definition names in the VDM++ specification.

3.2 BWDM

BWDM [10] analyzes boundary values of function definitions in VDM++ specification, and automatically generates test cases that consists of boundary values and expected output. BWDM is a command line based tool, and it consists of following four processing modules.

- Parser
- Extractor
- Input Data Generator
- Expected Output Data Generator

BWDM takes two inputs: a VDM++ specification and a decision table generated by VDTable. The user previously describes the VDM++ specification and the generates of the decision table by VDTable. The Parser generates the abstract syntax tree from VDM++ specification by using VDMJ. The Extractor extracts if-expressions, parameter, and argument type.

A decision table generated by VDTable is used to generate expected output data. BWDM analyzes the boundary value of the function definition described in the input of VDM++ specification, and finally outputs the generated boundary values and expected output data as test cases. The generated test cases are stored in a CSV (Comma Separated Values) file. Following sections, we describe about Input Data Generator and Expected Output Data Generator.

Input Data Generator

Fig. 3.9 shows the flow from the Parser to the Input Data Generator. The Input Data Generator analyzes of boundary values of if-expressions and type argument types that is extracted from the VDM++ specification.

The target if-expression is limited to inequalities and remainder expressions in which one side is a variable on both sides, and is also not supported for complex conditional expressions. Argument types are limited to integer types (int, nat, nat1).

In the case of if-expressions, BWDM presents the value of the boundary at which the truth value changes for result of boundary value analysis. In the case of argument types, BWDM presents maximum value, maximum value + 1, minimum value, and minimum value - 1 of its type. The generated boundary value is used for deriving the Expected Output Generator. and it finally becomes the Input data of test cases.

Expected Output Data Generator

Fig. 3.10 shows the flow of the Expected Output Data Generator. The Expected Output Data Generator generates expected output data from boundary values generated by the

```
class MixSpecification

functions

 mix: nat * int -> seq of char
   mix( arg1 , arg2 ) ==
     if( arg1 mod 2 = 0) then
       if (0 <= arg2 ) then
         " arg1 :even, arg2 : positive"
       else
         " arg1 : even, arg2 : negative"
     else
       if( arg2 < 0) then
         " arg1 : odd, arg2 : negative"
       else
         " arg1 : odd, arg2 : positive";

end MixSpecification
```

Fig. 3.11. The VDM++ specification that mixed inequality and surplus expression

```
The number of arguments:2

argument type:  argument1:nat   argument2:int

No.     input data      -->     expected output data
No.1    natMin-1        intMin-1        -->     Undefined Action
No.2    natMin-1        intMin          -->     Undefined Action
No.3    natMin-1        intMax          -->     Undefined Action
No.4    natMin-1        intMax+1        -->     Undefined Action
No.5    natMin-1        0               -->     Undefined Action
No.6    natMin-1        -1              -->     Undefined Action
No.38   3               intMin          -->      arg1:odd arg2:negative
No.39   3               intMax          -->      arg1:odd arg2:positive
No.40   3               intMax+1        -->     Undefined Action
No.41   3               0               -->      arg1:odd arg2:positive
No.42   3               -1              -->      arg1:odd arg2:negative
```

Fig. 3.12. The generated test cases by applying the Fig. 3.11 to BWDM

Input Data Generator and the decision table. The pairs of input data and expected output data are presented as test cases.

Application Example

In this section, we show the result of application of BWDM. Fig. 3.11 shows the VDM++ specification that mixed inequality and surplus expression. And Fig. 3.12 shows test cases generated by BWDM from Fig. 3.11. These test cases are possible to use directly, such as combinatorial testing of overture.

4 Evaluation

In this section, we evaluate both tools. For both tools, we measure the time to finish their operation and evaluate what how much they can improve the efficiency of the work.

Table 5.1 Time taken for creation of decision tables for testers and VDTable (sec)

Tester	VDM++ Specification (the number of combinations of the truth values of the conditions)		
	Specification A(4)	Specification B (16)	Specification C (256)
A	252	405	1131
B	213	559	1292
C	169	499	1194
D	240	435	1859
E	134	557	1397
Mean Time	216	498	1629
VDTable	0.012	0.016	0.02

Table 5.2 Execution time of BWDM in each specification (ms)

Times	Specification 1	Specification 2	Specification 3
1^{st} time	325	316	6277
2^{nd} time	283	389	5321
3^{rd} time	500	371	6236
4^{th} time	291	334	5070
5^{th} time	269	371	5700
Average	334	356	5720

4.1 Evaluation of VDTable

We evaluated VDTable by measuring and comparing the both of the time to create the decision table by the test engineer manually and by VDTable automatically.

In the specification used in the experiment, the number of combinations of the truth values of the conditions is different. Each number of it are A: 4, specification B: 16, specification C: 256. The testers are five students, A, B, C, D, and E, belonging to the Department of Information Sciences, the University of Miyazaki.

Table 5.1 shows the average of the measured time and the comparison result of VDTable. Testers took about three minutes to create a decision table for the specification A with the smallest combination of truth values of the conditions. On the other hand, in the case of using the VDTable developed in this research, generation of the decision table was completed in about 20 milliseconds for 256 combinations of the truth values of the conditions, even for the specification C. From this, it can be said that VDTable significantly reduced the labor and time of manual create work when creating a decision table using the VDM++ specification. That is, the VDTable is useful when designing a decision table using the VDM++ specification.

4.2 Evaluation of BWDM

To consider BWDM usefulness, we used 3 specifications. We measured the time to generate test cases. Table 5.2 shows conclusion of measuring. Within the three specification, the if-conditional expressions of inequalities are described. The specification 1 has 5 if-conditional expressions, specification 2 has 10 expressions and specification 3 has 15 expressions. To measure execution time, we used System.nanoTime [14] method of Java. We measured the time that is to give the VDM++ specification and the decision table as an input to the BWDM and to finish outputting the test cases. The unit of time was millisecond, and measurements were made five times for each specification, and we calculated the average of 5 measuring as well. Here, we don't consider the time to prepare the VDM++ specification and the decision table.

For the average time in each specification, specification 1 and specification 2 were less than 0.5 seconds, and specification 3 was less than seven seconds. The number of rules on the decision table for specification 1 (number of columns in the decision table) is 32, specification 2 is 1024 and specification 3 is 1048576. These represent the number of states that the function can take, depending on the Boolean value in the if-conditional expression. When constraint conditions such as preconditions are described in the specification, the number of states actually taken by the function is less than this number. However, it is troublesome and time consuming to manually design test cases for functions that can have as many states as possible. On the other hand, BWDM can automatically generate boundary value test cases in several seconds if the specification is up to condition number 15.

Hence, we think that BWDM is usefulness when testing software that implemented from VDM++ specification.

5 Conclusion

In this research, in order to improve the efficiency of the test process in software development using formal methods, we have proposed the method to generate a decision tables and boundary values from VDM++ specifications and implemented two tools: VDTable and BWDM. First, the both methods parsed the VDM++ specification. Thereafter, VDTable extracted conditions and actions from the parsing data, created truth values and then generated a decision table. And BWDM extracted the if-expressions and argument types of function definition, performed boundary value analysis, and finally generated test cases.

As a result of applying the VDM++ specification to both tools, it was confirmed that decision table generation and boundary value analysis can be executed without problems. In comparison with testers, we confirmed that both tools can reduce time and labor than manual work. From this, it can be said that the work efficiency of the test process, which is the object of this research, was achieved. The future works are shown below.

Common task of both tools

- Correspondence to exponential explosion
  - As conditions increase, the number of the test cases and the state of decision table increases by exponentially. As a correspondence, we are thinking about improving the VDTable GUI and adding options to specify test cases to be generated by BWDM.
- Expanding coverage of the tool
  - The definition of VDM++ what supported by our tools is only function defintion. There are many unsupported syntaxes in the two tools. (operation definition, let be, for, while and others) For Resolving this issue, VDTable needs new extraction rule for unsupported syntaxes. In case of BWDM, it needs that new method to generate boundary values from unsupported syntaxes.
- Evaluation by using huge specification that used in real development
  - In this paper, we used specifications whose scale was small, and prepared by us. For future development, application data according to the specification used in actual development will be useful. The opinion on tools by actual SE will be very helpful too.
- Correspondence to other test design techniques
  - In addition to the decision table and boundary value analysis, many test design techniques are known; for example, pairwise testing. Automatic execution of them based on VDM++ specification also has room for further work.

The task of VDTable

- Correspondence to compound conditional expression
  - Compound conditional expression is a combination of two or more conditions with and, or. Although the current VDTable converts the compound conditional expression as it is to the decision table, decomposition into a simple conditional expression makes it easier to understand the specification. We think that it's possible to resolve this issue by splitting into individual simple conditions by logical operators in compound conditional expressions when VDTable extracts conditional expressions.

The task of BWDM

- Implementation of Symbolic Execution
  - At present, processing base on symbolic execution is being implemented in BWDM in order to make C1 coverage 100 % for structures such as if-then-else expression in function definition.

References

1. “How to use Decision Tables”, http://reqtest.com/requirements-blog/a-guide-to-using-decision-tables/, last accessed 2017/7/31
2. S. C. Reid: An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: Proceedings Fourth International Software Metrics Symposium, pp. 64-73 (1997)
3. Blake Neate: Boundary Value Analysis, University of Wales Swansea (2006)
4. Dines Bjørner and Cliff B. Jones.: The Vienna Development Method: The Meta-Language. In: Computer Science, Lecture Notes, Vol. 61, http://www2.imm.dtu.dk/pubdb/p.php?2256 (1978)
5. Cliff B. Jones.: Systematic Software Development using VDM Second Edition. Prentice Hall International, United States (1995)
6. ISO: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (1996)
7. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef.: Validated Designs for Object-oriented Systems. Springer, New York (2005), ISBN: 1-85233-881-4
8. Tetsuro Katayama, Kenta Nishikawa, Yoshihiro Kita, Hisaaki Yamaba, and Naonobu Okazaki.: Proposal of a Supporting Method to Generate a Decision Table from the Formal Specification. In: Journal of Robotics, Networking and Artificial Life (JRNAL 2014), Vol. 1, No. 3, pp. 174-178 (2014)
9. Kenta Nishikawa, Tetsuro Katayama, Yoshihiro Kita, Hisaaki Yamaba, Kentaro Aburada, and Naonobu Okazaki.: Prototype of a Decision Table Generation Tool from the Formal Specification. In: International Conference on Artificial Life and Robotics 2015 (ICAROB 2015), pp. 388-391 (2015)
10. Hiroki Tachiyama, Tetsuro Katayama, Yoshihiro Kita, Hisaaki Yamaba, and Naonobu Okazaki.: Prototype of Test Cases Automatic Generation tool BWDM Based on Boundary Value Analysis with VDM++. In: International Conference on Artificial Life and Robotics 2017 (ICAROB 2017), pp. 275-278 (2017)
11. “What is Equivalence partitioning in Software testing?”, http://istqbexamcertification.com/what-is-equivalence-partitioning-in-software-testing/, last accessed 2017/7/31
12. Peter W. V. Tran-Jørgensen, Peter Gorm Larsen, and Nick Battle.: Using JML-based Code Generation to Enhance Test Automation for VDM Models. In: Proceedings of the 14th Overture Workshop, pp. 79-93 (2016)
13. Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle.: Combinatorial Testing for VDM. In: 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)
14. “System (Java Platform SE 7)”, https://docs.oracle.com/javase/7/docs/api/java/lang/System.html, last accessed 2017/7/31

Analysis Separation without Visitors

Nick Battle

Fujitsu UK (nick.battle@gmail.com)

Abstract. The design of VDMJ version 3 limits the tool's extensibility because the code for different analyses is tightly coupled with the classes that represent the AST. This situation was improved with Overture version 2 by using visitors. However, while this improves the tool's extensibility, the resulting code is slow to execute and complex to work with. This work presents an alternative approach, implemented in VDMJ version 4, which separates analyses from the AST and from each other, but which avoids the complexity of the visitor pattern while maintaining the performance advantage of VDMJ.

Keywords: design patterns, separation of concerns, code complexity

1 Introduction

VDMJ [3] is a command line tool written in Java, offering support for VDM [1] specifications. As well as parsing all dialects of the VDM specification language, the tool offers three analyses: type checking, model simulation (via an interpreter, with combinatorial testing [7]) and the generation of proof obligations. The source code of VDMJ is compact and efficient, but it is not easily extended, nor would it easily support independently written analysis plugins. This is because the supported analyses are tightly coupled with each other in the same Java classes.

The Overture [2] project has produced an Eclipse [4] based VDM tool. The Overture tool is based on VDMJ, but it has been re-structured internally to de-couple and separate the analyses. This extensibility comes at a cost, however. The internal structure of the Overture code is more complex than VDMJ, and as a result it is harder to develop and maintain. Overture's analyses are also slower to execute.

This work presents recent changes in VDMJ version 4 which add similar extensibility features to Overture, but without adding extra complexity or slowing the analyses. It does this by avoiding the visitor pattern [5], instead separating analyses by producing new data structures dynamically when each analysis is selected.

Section 2 describes the design of VDMJ version 3. Section 3 describes how Overture uses the visitor pattern to separate analyses. Section 4 introduces the alternative to the visitor pattern used in VDMJ version 4. Section 5 looks at the performance characteristics of the new pattern and Section 6 considers its drawbacks. Lastly, Section 7 considers related work, Section 8 possible future work and Section 9 adds a summary.

2 VDMJ Version 3

The design of VDMJ, up to and including version 3, is simple. The parser produces an Abstract Syntax Tree (AST) comprised of objects whose classes include the code of the analyses supported. The execution of those analyses then proceeds by direct recursion over the AST. This is a version of the interpreter pattern [5].

For example, in Figure 1, the parse of the expression "1 + a" produces a tree of three objects. The classes of those objects contain code for type checking (TC), interpretation (IN) and proof obligation generation (PO) for each node, as well as the linkage and primitive data from the parser (AST).

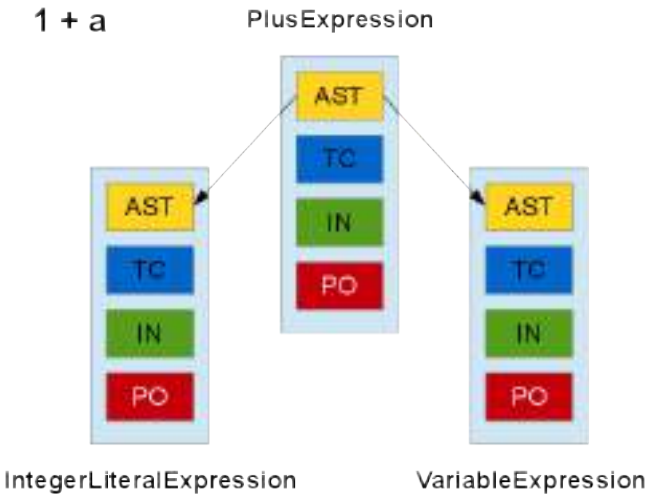


Fig. 1. Analyses in VDMJ version 3

To type check the expression, the TC method of the plus expression at the root of the tree is invoked. This recurses into the literal and variable expression TC methods, which return the types of those sub-expressions. The root then checks whether both sub-types are numeric (as required for the "+" operator) and finally deduces and returns the resulting type to its caller. The same recursive principle is used by the other analyses.

This approach is simple and has several advantages:

- Classes are small and only contain analysis code related to the one AST node concerned.
- Classes can form a hierarchy, with common code in abstract parent classes (eg. NumericBinaryExpression)
- Results from one analysis can be stored in the node and processed by other analyses of the same node.

– The execution of analyses is efficient, because all the required methods are local and can be called directly.

However, if the objective is to produce an extensible tool, this approach is limiting because the code for the different analyses is tightly coupled in the same classes. If different groups want to work on new analyses independently, they would all have to modify the same class files. Supporting a dynamic "plugin" design for analyses would be all but impossible. These points were noted in [6].

3 Overture Version 2

Overture was originally based closely on VDMJ. But to address these extensibility concerns, Overture version 2 was re-designed to use the visitor pattern [5][6]. This pattern separates the data (AST) from the analysis of that data, as well as separating analyses from each other. A simplified view of the new design is illustrated in Figure 2. Note that the AST classes now have no analysis code, and all the code for each type of analysis is collected together into one "visitor" class in separate methods.

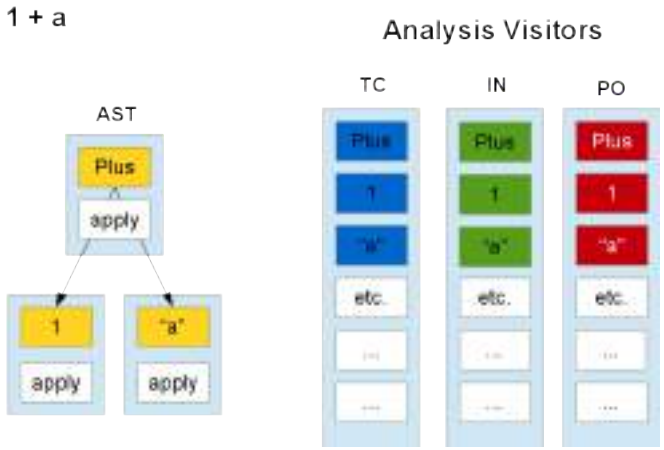


Fig. 2. Overture version 2 Analysis Separation with Visitors

This is a much more extensible design pattern, and allows analyses to be worked on independently, as well as allowing new analyses to be produced without changing the fundamental architecture. It would also support a dynamic plugin architecture in future. But when the visitor pattern is scaled up to support a complex grammar, some problems arise. For example:

– Class sizes can become very large. The VDM grammar has close to 300 node types and therefore naively, a comprehensive visitor would require 300 methods, amounting to several thousand lines of code.

- Visitor methods have no hierarchy, and so there is nowhere obvious to put common code. Private methods could be used, but that only makes the visitor classes even larger.
- Visitors are usually stateless, so there is nowhere obvious to store the node-specific results or working data of analyses.
- The solutions to the previous two points, together with the "double dispatch" semantics of the visitor pattern significantly slow down the execution of analyses.

To address the problem of large class sizes, Overture splits up the main visitors, firstly into large grammatical groups (definitions, expressions, statements etc). Secondly, large visitors are split into separate classes which extend each other. This produces a child class that acquires all the methods needed by inheritance, without including them all in one large source file. But the problem of relatively large source files remains (the largest is still 3751 lines).

To address the issue of where to put common code, Overture uses "assistants". These are usually stateless classes that contain a small group of methods that relate to common processing for a particular analysis, for a particular set of node types. This idea has merit, and keeps the common code away from the visitors (which are already large). But it can be difficult to find the right assistant for the right purpose in a given context. There are 66 assistants currently, and while they have sensible names, they are not related to the visitors and methods they serve in any obvious way (cf. common code in a class hierarchy).

The problem of where to hold analysis state is solved in one of two ways in Overture. Some information that is globally useful (for example the type information from the type checker) is stored in the AST nodes. Other information that is only used by analyses internally (for example, current breakpoint settings in the interpreter) is held by stateful assistants. This data is usually a map from AST nodes to a given type.

Storing type information in the AST may seem an obvious solution, but it implicitly creates a dependency between the type checker and the AST, and between the AST and the consumers of that type information. Consider what would happen if someone produced an alternative type checker with a different representation of types. The AST would have to change or be extended to include both types, and all type dependent analyses would have to change too.

Storing state information in a map is the only other reasonable alternative. But consider that (for example) the interpreter has to consult the current breakpoint setting for every single node that it encounters, and to do each check it has to obtain an assistant, and to do that it has to use an assistant factory. This is a significant performance overhead.

4 VDMJ Version 4

VDMJ version 4 uses an alternative approach to analysis separation that avoids many of the problems associated with the visitor pattern.

As with Overture, the AST is simplified to use objects whose classes only contain parser information. To perform an analysis (the first is usually type checking, but it

need not be), the AST structure is used to create a *separate tree* that is exactly the same shape, but comprised of classes that only contain code for that particular analysis. The analysis then proceeds by recursive descent over the new tree.

When a subsequent analysis is performed that requires the results of the first analysis (for example, the interpreter uses the type checker output), the same tree creation procedure is performed, except transforming the type checker tree - along with its embedded type information - into a tree that is specialized for interpretation. The execution of the interpreter then proceeds by recursive descent over the new tree, as it did with VDMJ version 3. This is illustrated in Figure 3.

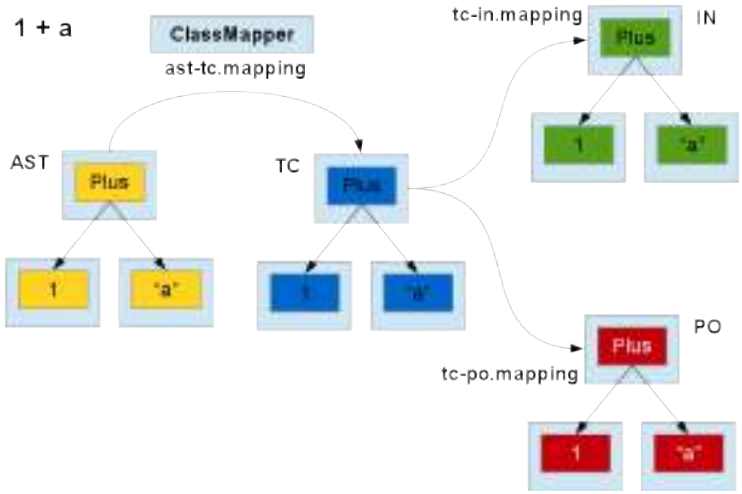


Fig. 3. VDMJ Analysis Separation by Tree Creation

The process of analysis tree creation is performed by a new component called the *ClassMapper*, which is driven by "mapping files" that describe how to create each object in the target tree, given an object in the source tree. For example, the part of the mapping file to create the TC classes from the AST classes used in the example (in the file `ast-tc.mapping`), would be as follows:

```
package com.fujitsu.vdmj.ast.expressions to
        com.fujitsu.vdmj.tc.expressions;

map ASTPlusExpression{left, op, right} to
    TCPlusExpression(left, op, right);
map ASTIntegerLiteralExpression{value} to
    TCIntegerLiteralExpression(value);
map ASTVariableExpression{location, name, original} to
    TCVariableExpression(location, name, original);
...
```

A mapping file consists of *package* lines, which define the source and target Java packages for the *map* lines that follow until the next package line. Map lines define a source class and fields within that class which should be passed to a corresponding target class constructor, with the field arguments identified. Alternatively, a line may be defined as *unmapped*, which means the target class is the same as the source class. A mapping file has a map or unmapped line for every source/target conversion that may be required, which is therefore around 350 lines.

The process of tree conversion starts at the tree root. The source class is identified in the mapping file, and the list of required fields read from the root object using Java reflection. These field values are then used to call the constructor of the target class identified. But *before* the constructor is called, the constructor arguments are themselves converted by the same process. This therefore duplicates the entire tree using the mapping. The mapper additionally keeps a cache of objects that are in the process of being converted and which have already been converted, so that cycles in the source tree can be correctly converted into the same cycle structures in the target tree.

The EBNF grammar of the mapping file is as follows:

```
mappings = { package | map | unmapped }
package = "package", pname, "to", pname, ";"
map = "map", source, "to", destination, ";"
unmapped = "unmapped", fqcn, ";"
source = cname, "{", [ fields ], "}"
destination = cname, "(", [ fields | "this" ], ")"
fields = fname, [ { ",", fname } ]

pname = <a Java package name>
cname = <a Java constructor name>
fname = <a Java field name>
fqcn  = <a Java fully qualified class name>
```

Mapping files are read on demand and cached. The Java reflection *Field* and *Constructor* classes for each map line are created and checked when the file is loaded, so that the process of creating the output tree can proceed as quickly as possible.

This tree mapping approach has several advantages:

- Analysis separation has been achieved. Analysis code is separate from the AST and different analyses are separated from each other.
- The individual class files in the analyses are very small indeed – even smaller than VDMJ version 3 – and they are tightly focussed on a single purpose. The analysis code itself is virtually identical to VDMJ version 3, but split into one class per node per analysis.
- The AST classes are immutable, with fewer fields that VDMJ version 3. This means that their constructors are more efficient and the parser is up to 30% faster than before.
- Code complexity metrics are as expected. The average methods per class is much lower; the average inheritance depth goes up (more classes have the same depth); the average method lines of code decreases because of more simple constructors.

Afferent coupling and McCabe complexity decrease. Most metrics are roughly the same.
- The classes retain the same relative hierarchy as before, and so common code can reside in abstract parent classes.
- The output from an analysis or intermediate state resides within the objects of each analysis tree. If a subsequent analysis wants to use that information, the process of creating its tree will copy a reference to the information from the source tree. Note that the parser output AST is completely immutable.
- There are still dependencies between analyses, but those dependencies are made explicit via the mappings. Multiple analyses of the same kind, such as different type checkers, can co-exist.
- Since there are no visitor redirections, no assistants or assistant factories, and no state map-lookups, the execution of the analyses goes as fast as VDMJ version 3 - that is to say, faster than Overture version 2.

However, these advantages can only be realised once the cost of the analysis tree creation has been paid. Compared to creating assistants, or looking up nodes in a map, this cost may appear prohibitive. But in practice, Java is heavily optimized for object creation, and the overall cost of the tree creation is modest. For most specifications, analysis trees can be created in a small fraction of a second.

5 Performance

The cost of analysis tree creation comes in two parts: the mapping file must be read, and then the source tree must be processed to produce the target tree. The cost of reading the mapping file is fixed (though the actual performance in a multi-process environment depends on whether disk blocks are cached, which other processes are active and so on). The memory footprint is also fixed at a few hundred Kb. However, the cost of the target tree creation depends on the size of the source tree being processed.

Both the visitor pattern and the tree creation pattern incur a performance penalty. But one important difference with tree creation is that the penalty occurs "between" analyses, whereas the visitor pattern penalty is "during" analyses. So to determine the real impact of the tree creation overhead, one important consideration is whether an analysis is one-shot, or whether it is performed repeatedly. For example, a specification need only be type-checked once, and it need only have proof obligations generated once. But many simulations may be performed with the interpreter in a given session - with combinatorial tests, perhaps millions of executions. So in the case of the interpreter, the performance advantage of the VDMJ execution engine will almost certainly outweigh the cost of the production of the interpreter tree.

In practice, the cost of loading the mapping file is negligible. All of the mapping files are roughly the same size, and are read in less than 0.2 of a second. This cost is once-only per session per analysis, so if multiple specifications were processed, this cost would only be paid once. This is not noticeable. In a multi-project workspace like Overture, the cost could effectively be paid once at startup.

The cost of tree duplication is more variable, and depends on the analysis being created as well as the size of the tree. The ClassMapper can create new tree nodes at

a rate of between 200K and 800K nodes per second. Note that 500K AST nodes is roughly equivalent to a 100,000 line specification. So this means that a 10,000 line specification could be prepared for (say) type checking in a fraction of a second. Given that most specifications are less than 10,000 lines, this means that the tree duplication times are usually not noticeable either.

To illustrate this, Figure 4 compares the performance in seconds of the VDMJ version 3 and 4, and Overture type checkers for a series of VDM-SL specifications, from 500 lines to 12,000 lines. The "shmem.vdmsl" specification was used (from the Overture examples), its module being copied to produce ever larger specifications. This specification was chosen because it includes a variety of functions, operations, expressions, statements and patterns.

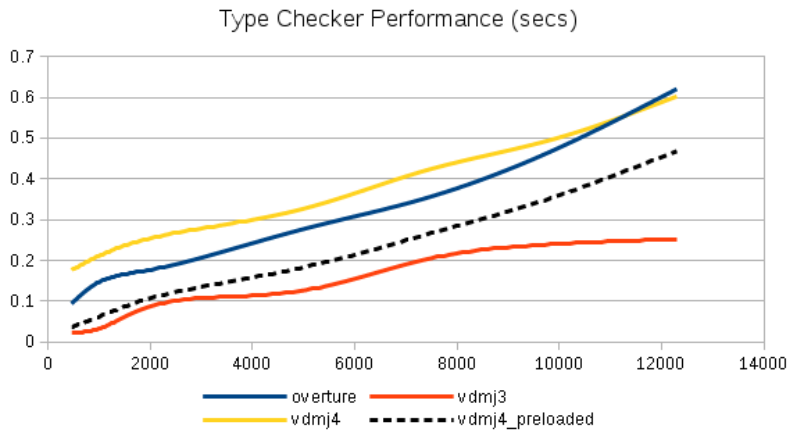


Fig. 4. Type checking performance, lines of specification per second

The figure shows the average performance of 20 consecutive executions[1]. The collection of this data was difficult, since the small differences between the performance of the various tools is comparable to the natural fluctuations that are experienced using Java on a multi-process operating system. Hence the use of averaging over multiple runs.

The execution time of VDMJ version 4 is consistently slower than VDMJ version 3, which is mostly the delay for reading the `ast-tc.mapping` file. It is also possible to see a divergence of the two VDMJ lines. This is caused by the extra delay of the tree creation, which adds between 0.03 seconds for a 500 line specification, and about 0.17 seconds for a 12,000 line specification. As a result, compared to the Overture

[1] All measurements were made on 32-bit Windows 7 running on an i7 laptop, using Java 8 with 1Gb of heap.

2.4.8 performance, VDMJ version 4 is slightly slower below about 11,000 lines in this example (the difference is less than 0.1 secs).

However, if the time to load the mapping file is removed from the VDMJ version 4 times, just leaving the tree creation time (the dotted line), then both VDMJ versions perform better than Overture, and version 4 performance is a few hundredths of a second slower than version 3 below 2000 lines. This may be a fairer comparison, since the mapping file will only be loaded once. Overture performs a type check every time a specification file is saved in the editor; at worst, the cost of reading the mapping file would be incurred once on the first save.

Figure 5 compares VDMJ and Overture type checker performance for considerably larger specifications. Here the fixed delay to load the mapping file is outweighed by the cost of the tree copying. As shown, for an 80,000 line specification, the divergence between VDMJ versions 3 and 4 is just over two seconds. But even with this extra cost, the overall performance advantage of VDMJ type checking means that it is still faster than Overture.

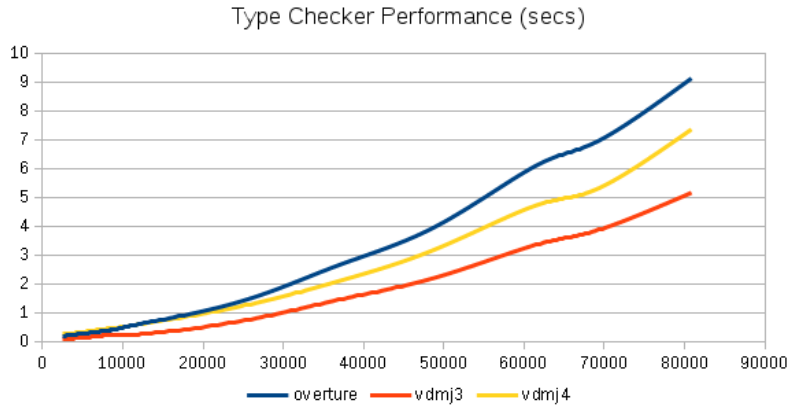


Fig. 5. Type checking performance, lines of specification per second

These figures just show the performance of the type checker tree creation, but the costs of the interpreter and proof obligation tree creations are similar. So for example, a 10,000 line specification might incur a 0.25 second delay, once, before the interpreter could be used. But thereafter the execution of the interpreter would run at "VDMJ speeds" no matter how many expressions were evaluated, which is usually faster than Overture.

6 Drawbacks

One significant drawback of the tree creation method is that it is not a recognised design pattern. That is always the case for novel solutions, but it would be wise to carefully consider possible problems, and the advantages of visitors, before adopting this new pattern exclusively.

In come cases, the visitor pattern may make code simpler than direct recursion. For example, although the "main visitors" of the analyses are large and complex, there are also many "small visitors" that implement what used to be recursive processes on the tree. Note that one visitor can only do one job because it implements one method for each node type. So a process that (say) matches a type against a pattern to derive a set of bound variable definitions must be performed by a separate visitor to the main type check visitor. There are many of these "small visitors" in Overture (over 120). The nice thing about them is that they encapsulate all of the processing for the (few) node types that are relevant in one small class. The equivalent code in VDMJ is spread over many classes, which can be harder to work with. This point was made in [6]. In principle, such small visitors could be re-used by different analyses; this would not be possible with tree creation.

Note that tree creation does not mean that the trees concerned cannot also support visitors. A hybrid solution is perfectly possible, using tree creation for the main analyses (with all the advantages listed above), but using visitors for the small recursive processes that the main analyses use.

The current VDMJ tree structures do not include links to ancestors, only to children. This is is not a fundamental issue with the VDMJ approach – such links could be added – but it is an advantage of automatically generated AST systems, such as that in Overture. This point was also made in [6].

Tree duplication occupies more memory than a single AST. But for the most part, the extra memory (over the data that was present in the VDMJ version 3 design) is comprised of object reference linkages between the new tree nodes. Even for large specifications, this overhead only amounts to a few Megabytes. In principle, analysis trees can be deleted once the analysis has been performed (for example, the AST tree is deleted in VDMJ version 4 once the type checker tree has been created from it, since the type checker tree has all the information).

7 Related Work

The Overture project's use of the visitor pattern is covered in detail in [6]. The visitor and interpreter patterns are described in [5].

The author is unaware of any detailed work discussing the weaknesses of the visitor pattern when used at scale, as raised here. Similarly, the author is not aware of any use of tree copying as a design pattern that may replace the visitor pattern.

8 Future Work

The weaknesses in the visitor pattern at scale, highlighted in section 3, may be due to Overture's implementation choices rather than because of fundamental problems with

the visitor pattern. It may be that other tool implementations have solved visitor scaling problems in better ways, which should be investigated. It may also be the case that better organization of the Overture visitor code would mitigate many of the maintenance problems experienced by its developers.

The performance analysis in this work only used VDM-SL specifications. It is possible that other dialects perform differently, though this is unlikely since the majority of the AST is the same. However, measurements would give a more complete picture of the performance of the ClassMapper.

Although the analysis separation presented here would support a modular "plugin" architecture, this is not currently implemented by VDMJ. An analysis plugin would have to include its own mapping file, as well as an indication of the source analysis tree that it should be derived from. Plugins would then fit together in a dependency tree, not unlike the plugin architecture of the Apache Maven[8] build system.

The creation of new analysis classes is currently a manual process, as is the creation of new mapping files. This would benefit greatly from tool support, perhaps creating skeleton classes and a mapping file for a new analysis based on the class structure and mapping of the analysis from which it is derived.

Currently, the mapping process can only create a new analysis tree from one source tree. But it is conceivable that a new analysis would depend on information from two or more source analysis trees, as long as they were of the same shape. This would require more work in the ClassMapper.

Visitors are an elegant solution for small recursive processes that only involve a few node types. An overall improvement may be achieved by converting these processes into visitors, which would reduce the size of the analysis classes even more.

9 Conclusion

A method of analysis separation has been presented which does not use the traditional visitor pattern. The new method - separation by analysis tree creation - retains many of the advantages of tightly coupled code without the problems of the coupling. The performance penalty of tree creation has been shown to be small compared to the overall analysis cost.

The practicality of the solution has been demonstrated by its inclusion in VDMJ version 4.

References

1. International Organization for Standardization, Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language International Standard ISO/IEC 13817-1, December 1996.
2. P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative Integrating Tools for VDM". ACM Software Engineering Notes, vol. 35, no. 1, January 2010.
3. "VDMJ User Guide". Technical report, Fujitsu Services Ltd., UK, 2009. `https://github.com/nickbattle/vdmj`.

4. The Eclipse Integrated Development Environment, http://www.eclipse.org/.
5. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, “Design patterns, Elements of Object Oriented Software”, Addison Wesley, 1995.
6. “Migrating to an Extensible Architecture for Abstract Syntax Trees”, Luis Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman, and Kenneth Lausdahl
7. Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle, “Combinatorial Testing for VDM”, Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods, Pisa, September 2010.
8. Apache Maven, https://maven.apache.org/index.html.

A Discrete Event-first Approach to Collaborative Modelling of Cyber-Physical Systems

Mihai Neghina[1], Constantin-Bala Zamfirescu[1], Peter Gorm Larsen[2], Kenneth Lausdahl[2], and Ken Pierce[3]

[1] Lucian Blaga Univ. of Sibiu, Faculty of Engineering, Department of Computer Science and Automatic Control, Romania
{mihai.neghina,zbcnanu}@gmail.com
[2] Aarhus University, Department of Engineering, Denmark
{pgl,lausdahl}@eng.au.dk
[3] School of Computing Science, Newcastle University, UK
kenneth.pierce@newcastle.ac.uk

Abstract. In the modelling of Cyber-Physical Systems (CPSs), there are different possible routes that can be followed to gradually achieve a collection of constituent models that can be co-simulated with a high level of accuracy. This paper demonstrates a methodology which initially develops all constituent models at a high level of abstraction with discrete-event models expressed using VDM. Subsequently, a number of these are refined (without changing the interfaces) by more detailed models expressed using different formalisms, and using tools that are capable of exporting Functional Mock-up Units (FMUs) for co-simulation using the Functional Mock-up Interface (FMI). The development team of each of these more detailed models can then experiment with the interactions with all the other constituent models, using the high-level discrete-event versions until higher-fidelity alternatives are ready.

Keywords: Co-Simulation, Cyber-physical Production Systems, heterogeneous modelling

1 Introduction

In the development of Cyber-Physical Systems (CPSs), a model-based approach can be an efficient way to master system complexity through an iterative development. In this paper we illustrate how a co-simulation technology [6] can be used to gradually increase the detail in a collaborative model (co-model) following a "discrete event first" (DE-first) methodology. In this approach, initial abstract models are produced using a discrete event (DE) formalism (in this case VDM) to identify the proper communication interfaces and interaction protocols among different models. These are gradually replaced by more detailed models using appropriate technologies, for example continuous time (CT) models of physical phenomena.

The case study deals with the virtual design and validation of a CPS-based manufacturing system for assembling USB sticks. It is a representative example of part of a

distributed and heterogeneous system in which products, manufacturing resources, orders and infrastructure are all cyber-physical. In this setting, several features (such as asynchronous communication, messages flow, autonomy, self-adaptation, etc.) should be investigated at design time, for example using a collaborative modelling approach. Consequently, the case study offers a balance between being sufficiently simple to be easily followed as a production line example, including generating a tangible output, and at the same time being sufficiently general to allow the study of the co-simulation complexity. Furthermore, by choosing a USB stick, the example opens the possibility of extending the purpose of the study to interactions between generated hardware and generated software solutions in the production line.

The DESTECS project introduced a co-simulation approach enabling users to combine one DE and one CT model [5]. The Crescendo tool was developed here and it has a direct combination between the DE formalism VDM supported by the Overture tool [8] and the CT formalism Bond-graphs supported by the 20-sim tool [7]. Here a DE-first approach was presented as one of the possible methodologies for the development of CPSs. In the INTO-CPS project these results was generalised using a co-simulation technology based on the Functional Mock-up Interface (FMI) that enables any number of DE and CT constituent models to be co-simulated [1]. In this paper we demonstrate how it is valuable using the DE-first approach from DESTECS in a setting with many different constituent models.

The remaining part of this paper start by introducing the INTO-CPS technology in Section 2. This is followed by Section 3 describing the CPS developing approach for starting out with DE models for each constituent model and then gradually exchanging some of these with more realistic models. Section 4 introduces the industrial case study in the area of manufacturing from the company Continental used in this paper, including the results of using the INTO-CPS co-simulation technology both in a *homogeneous* as well as a *heterogeneous* setting on that case study. Finally, Section 5 provides a collection of concluding remarks and look into the future of using this approach.

2 The INTO-CPS Technology

In the INTO-CPS project we start from the view that disciplines such as software, mechatronics and control engineering have evolved notations and theories that are tailored to their engineering needs, and that it is undesirable to suppress this diversity by enforcing uniform general-purpose models [3,4,9,10,11]. Our goal is to achieve a practical integration of diverse formalisms at the semantic level, and to realise the benefits in integrated tool chains. The overall workflow and services from the tool chain used in this project is illustrated in Figure 1.

At the top level, the tool chain will allow requirements to be described using SysML with a new CPS profile made inside the Modelio tool. This SysML profile allows the architecture of a CPS to be described, including both software and physical elements and based on this it is possible to automatically generate FMI model descriptions for each constituent model [13]. It is also possible to automatically generate the overall connection between different Functional Mockup Units (FMUs) for a co-simulation. Note that although SysML also have diagrams to express behaviour the CPS profile in

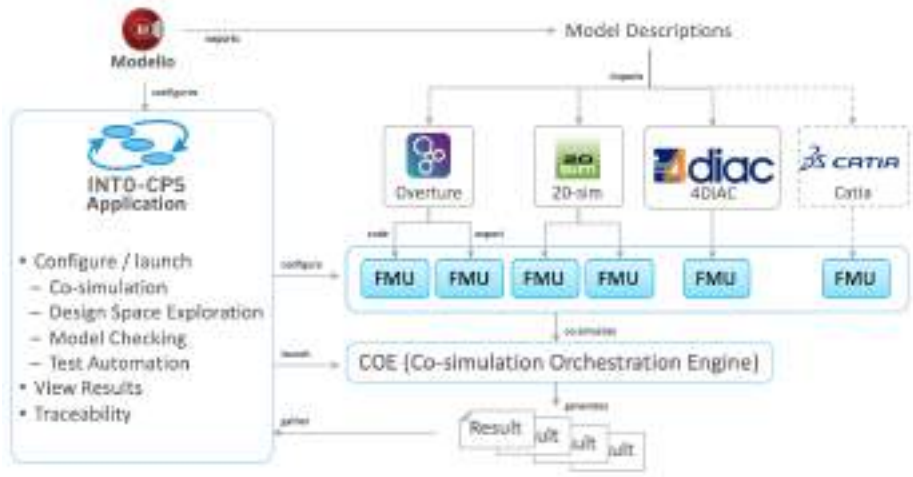


Fig. 1. The INTO-CPS tool chain used in this project

Modelio has not been extended to enable the generation of FMUs from such diagrams, so SysML in this work is primarily used at the high-level architecture describing how different constituent models are combined. It is also worth noting that the types that can be used in the interfaces for FMUs are quite basic so elaborate VDM values can only be exchanged using strings and in that case one need to have the same understanding of such structured values in the constituent models exchanging such values.

Such FMI model descriptions can subsequently be imported into all the baseline modelling and simulation tools included in the INTO-CPS project. All of these can produce detailed models that can be exported as FMUs that each act as independent simulation units that can be incorporated into an overall co-simulation. The constituent models can either be in the form of Discrete Event (DE) models or in the form of Continuous Time (CT) models combined in different ways. Thus, heterogeneous constituent models can then be built around this FMI interface, using the initial model descriptions as a starting point. A Co-simulation Orchestration Engine (COE) then allows these constituent models of a CPS to be evaluated through co-simulation. The COE also allows real software and physical elements to participate in co-simulation alongside models, enabling both Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation.

The different modelling and simulation tools used in this experiment included the technologies from the DESTECS project (Overture and 20-sim) and 4DIAC [14]. The original intention was to include Catia to model the robotic arm in the case study, but unfortunately a license was not available for the version of this tool capable of generating FMUs. The benefit of the co-simulation approach is that the Catia could be replaced by an equivalent 20-sim model.

In order to have a user friendly interface to managing this process a web-based INTO-CPS Application has been produced. This can be used to launch the COE enabling multiple co-simulations to be defined and executed, and the results collated and

presented automatically. The tool chain allows multiple co-simulations to be defined and executed using its Design Space Exploration (DSE) capabilities.

3 Discrete-Event First Modelling with INTO-CPS

3.1 Initial Models

Given a set of FMI model descriptions generated from a SysML model using Modelio, initial creation of the constituent models can begin. In general, this means importing the model descriptions into modelling tools (e.g. Overture, 20-sim), modelling behaviour in each, generating FMUs and then bringing them together in the INTO-CPS Application to run a co-simulation.

This direct approach can however be prone to failure. It requires that FMUs are available for all constituent models before co-simulations can be made. If each model is produced by a different team, then delays in one team may well delay all other teams. Similarly, if a single team produces constituent models in turn, then co-simulation can only begin at the end of modelling.

One way to overcome this is to produce quick, first version FMUs as early as possible to perform initial integration testing. These can then subsequently be replaced iteratively by increasingly detailed models, with integration testing performed each time a model is available, falling back to older versions as required. This however may be more difficult to do in some modelling paradigms.

An alternative is to take a "discrete-event first" approach. Here, a simple, abstract model of the entire system is created in the DE modelling environment, e.g. VDM and Overture, in order to sketch out the behaviour of all constituent models. This approach in a two-model setting is described in Fitzgerald et al. [5], including guidelines for simplifying continuous behaviour in discrete-event models. It is worth noting that this type of approach is conceptually similar to a traditional component-based development approach where one commonly initially use stub-modules initially.

3.2 Discrete-Event First with VDM/Overture

Given a SysML model and model descriptions for each constituent model, the suggested approach is to begin by building a single VDM-Real-Time (VDM-RT) [15] project in Overture with the following elements:

- A class for each constituent representing an FMU. Each class should define port-type instance variables (i.e. of type `IntPort`, `RealPort`, `BoolPort` or `StringPort`) corresponding to the model description and a constructor to take these ports as parameters. Each FMU class should also define a thread that calls a `Step` operation, which should implement some basic, abstract behaviour for the FMU.
- A `System` class that instantiates port and FMU objects based on the connections diagram. Ports should be passed to constructor of each FMU object. Each FMU object should be deployed on its own CPU.
- A `World` class that starts the thread of each FMU objects.

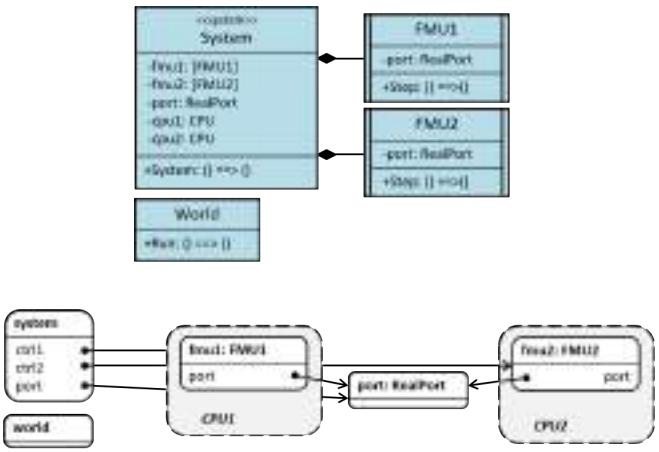


Fig. 2. Class diagram showing two simplified FMU classes created within a single VDM-RT project, and an object diagram showing them being instantiated as a test.

Class and object diagrams giving an example of the above is shown in Figure 2. In this example, there are two constituent models (called *FMU1* and *FMU2*) joined by a single connection of type real. Such a model can be simulated within Overture to test the behaviour of the FMUs. Once the behaviour of the FMU classes has been tested, FMUs can be produced for each and integrated into a first co-model. To generate FMUs, a project must be created for each constituent model, comprising:

- One of the FMU classes from the main project.
- A `HardwareInterface` class that defines the ports and annotations required by the Overture FMU export plug-in, reflecting those defined in the model description.
- A **`system`** class that instantiates the FMU class and passes the port objects from the `HardwareInterface` class to its constructor.
- A `World` class that starts the thread of the FMU class.

The above structure is shown in Figure 3. A skeleton project with a correctly annotated `HardwareInterface` class can be generated using the model description import feature of the Overture FMU plug-in. The FMU classes can be linked into the projects (rather than hard copies being made) from the main project, so that any changes made are reflected in both the main project and the individual FMU projects. Note that if the FMU classes need to share type definitions, these can be created in a class (e.g. called `Types`) in the main project, then this class can be linked into each of the FMU projects in the same way.

From these individual projects, FMUs can be exported and co-simulated within the INTO-CPS tool chain. These FMUs can then be replaced as higher-fidelity versions become available, however they can be retained and used for regression and integration testing by using different co-model configurations for each combination.

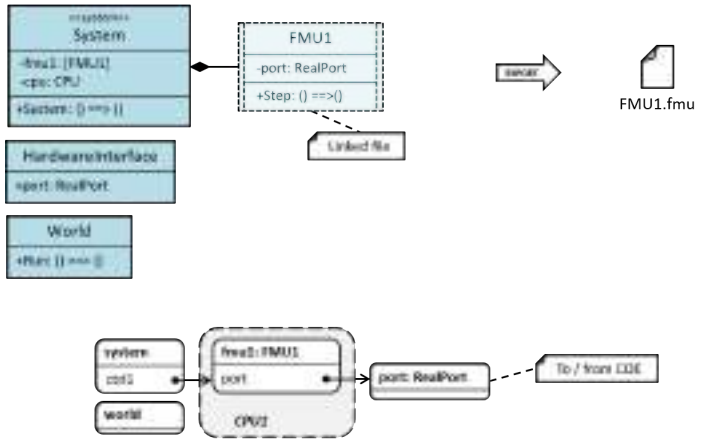


Fig. 3. Class and object diagrams showing a linked class within its own project for FMU creation.

4 Case Study: Manufacturing USB Sticks

The case study below concerns the manufacturing of USB sticks. The case study was chosen as a representative plant that might potentially be expanded into a full production line.

The USB sticks chosen consist of three parts (two lids and the main body). The production line should accept orders, change requests and cancellations from "virtual users", then assemble a stick with the required characteristics from parts available in a warehouse. The completed product should then move on a conveyor system to a test station, which validates the component before passing the completed product to the user. If the item is rejected (the test fails to confirm the requested colours on the stick), then the process is automatically re-started and a new request is made to the warehouse.

4.1 The Constituent Systems in the Case Study

Figure 4 shows the plant layout as intended at the beginning of the project, with emphasis on the physical entities where the control units will be embedded. These are:

W: Warehouse - buffer unit;
R: Robotic Arm - processing station;
C: Conveyor / Wagon (x3) - transportation units;
T: Test Station - processing station.

In addition, to capture the value adding processes in Industry 4.0 [12,15,2], the case study includes distinct units to reflect the users who place orders for assembling the USB sticks, and the required infrastructure that make possible for the others CPSs to exist. These are:

H: HMI (Human-Machine Interface) - reflecting the users; and

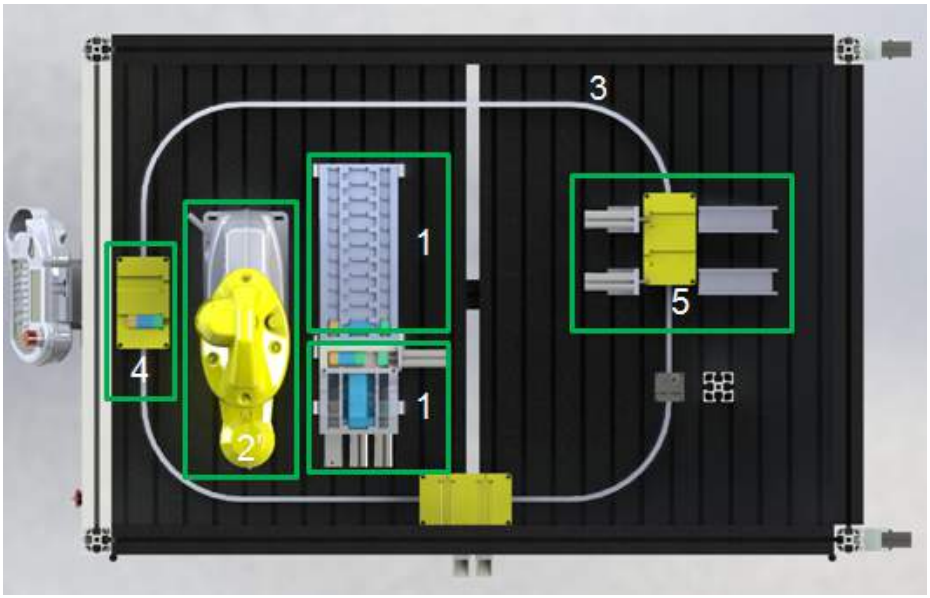


Fig. 4. Layout of the production line depicting (1) warehouse stacks and the assembly box, (2) the robotic arm and warehouse memory banks, (3) the circular track, (4) the loading station with one of the wagons, (5) the test station with another wagon.

P: Part Tracker - reflecting the infrastructure.

The HMI unit handles the user interface; it communicates only with the Part Tracker. The purpose of the real implementation of the HMI is to allow the user to place/change/-cancel orders at any time through an app running on smart devices. The Part Tracker is the central unit that handles orders; it communicates with the HMI, Warehouse, Wagons and the Test Station.

The Warehouse assembles the USB from component parts; it communicates only with the Part Tracker and the Robotic Arm, used for moving USB components from one location to another, either because the current available component does not correspond to the order or because the item has been assembled and needs to be moved to the waiting line.

The Robotic Arm moves parts or pieces from one location to another; it communicates only with the Warehouse. As with other models, the Overture/VDM-RT model of the Robotic Arm is on purpose incomplete, since the time required to move a part or piece from one location to another is implemented simply as a delay through a count-down timer.

The purpose of the Wagons is to transport the items from the waiting room (at the loading station) to the test station. The wagons communicate with the Part Tracker and to each other. Each wagon can be certain of its location only at the loading or test station, while in between the position is estimated periodically from the previous known position, time and own speed.

The Test Station reads the item characteristics and reports on their conformity to requirement. It communicates only with the Part Tracker, from which it received the requested set of colours for the item and to which it reports the test result.

4.2 Homogeneous Co-Simulations

The first step in the modelling of the case study was generating a functional, albeit not fully featured, model of the components (units) that would make up the USB stick production line. Each component was first modelled abstractly in VDM by using the Overture tool, following the approach described in Section 3. The goal of this homogeneous co-simulation was to identify the right interaction protocols (signals) among the various components (stations) of the prototype, and not on the model's accuracy. Therefore, the VDM simulation model includes distinct models for each component of the system (see Figure 5).

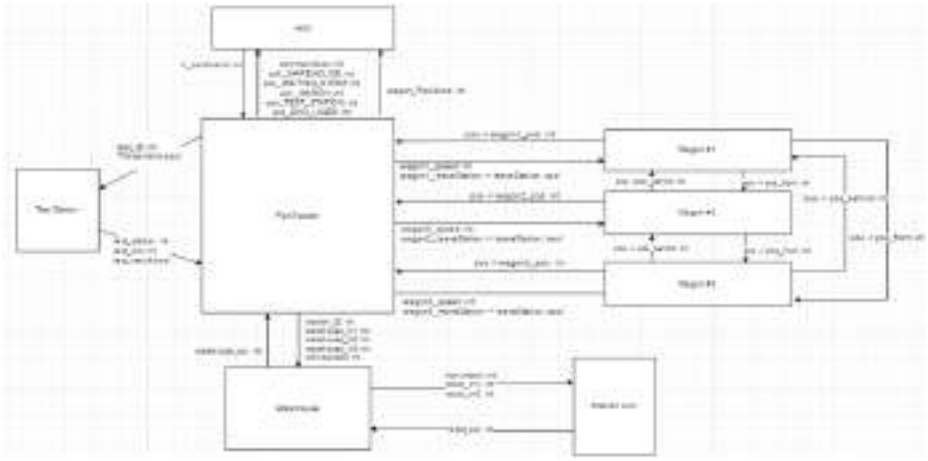


Fig. 5. Connections between VDM models

Having an early (working) co-simulation also provides other advantages, such as validating the interaction protocols, decomposing the project into units which could then be worked on separately, and the possibility of gradually increasing the complexity of the simulation and the possibility of replacing units of the model one by one with more accurate FMUs generated from dedicated platforms.

The VDM models do not need to have complete functionality for the homogeneous co-simulation, only the bare minimum from which the communication lines between units can be determined. The incompleteness of the VDM models is related to details of the inner workings of the components, not necessarily respecting all constraints of reality, such as randomly generating colours for USB parts and ignoring the physical capacity limit of the memory box in the warehouse or randomly choosing a duration for piece relocation for the robotic arm or randomly considering the test successful or

failed in the test station. Another example is breaking continuity: the warehouse model generates random colours for USB parts and can react to order cancellations. But if an order comes and the warehouse was able to select the correct colour for two of the three parts of the USB before the order was cancelled, when a new order arrives the warehouse just starts generating coloured parts anew and those leftover coloured parts are not considered (the reality of already being a part of a certain colour in a certain place is ignored and continuity is broken). Such aspects of the functionality are minor details with respect to the discrete event modelling used for the abstract validation. The internal states however are all well established, along with the communication patterns and lines between modules, such that the behaviour of the refined modules does not diverge substantially from the behaviour of the abstract models. Once established, the communication lines and the types of data they carry become hard constraints of the simulation that cannot be easily changed, but new lines of communication could be added if necessary.

The CSV (Comma-Separated Variables) library from the Overture tool is used in the HMI model to provide test data for orders for the production line. This approach is both flexible and powerful. The flexibility stems from the possibility of creating various scenarios with various amounts of orders and order/cancellation time delays for covering statistical scenarios, while the power comes from the repeatability of experiments. Having the same input CSV file, the co-simulation can be run with various parameters, but having exactly the same input orders at exactly the same time, thus generating a detailed picture of the behaviour of the system in controlled, repeatable experiments. The influence of certain system parameters may also be analysed.

The VDM model for the Warehouse is on purpose incomplete, since it randomly generates parts of various colours until the necessary (requested) colour is available. There is no stack of parts with pre-defined colours (therefore there is no possibility of ever running out of a colour) and there is no memory box where unused parts are deposited temporarily until a stick request with those characteristics comes around.

The abstract VDM model for the Wagons is idealised in the sense that the wagons can change speeds (accelerate or decelerate) immediately after receiving the command from the Part Tracker, (un)loading is instantaneous and that their estimated positions always correspond to the real positions on the track, which probably will not necessarily be true in the real implementation. Also, the wagons have no possibility of stopping unexpectedly, loosing the load or falling outside the track.

The abstract VDM model of the Test Station is also incomplete. It simply waits for a time and the randomly outputs true or false as the test result. The only parameter of controlling the output is the frequency of rejecting an item.

4.3 Communication between Units

The communication between units contains both simple (straightforward) messages, requesting the setting of a certain value or indicating the current value or state of a component, as well as composed messages that need to be decoded and the information extracted from them before that information can become useful. The purpose of the composed messages is twofold: to ensure that certain bits of information arrive simultaneously (as opposed to them coming on different message lines that may become

unsynchronised or for which further synchronisation logic might have been needed) and to account for the possibility of coded messages (that might be interesting in applications with significant noise, where error correcting codes might become useful).

For instance request from the Part Tracker to the Wagons to assume a certain speed or the feedback of the Wagon positions to the Part Tracker is done with straightforward messages (the value requested) on dedicated lines (that only carry these types of messages and nothing else). On the other hand, the order requests from the HMI to the Part Tracker or their acknowledgement (feedback) contain multiple pieces of information each.

4.4 Heterogeneous Co-Simulations

The heterogeneous co-simulation covers the successful attempts to replace the Overture-generated FMUs with more detailed and complete FMUs generated by specialized tools using appropriate formalisms. Table 1 shows the correspondence between the use case units and the program chosen (adequately suited) for implementing a complete simulation of the unit.

Types	Constituent system	Technology used
Orders	HMI	4DIAC with MQTT and Overture (VDM)
Infrastructure	Part Tracker	Overture (VDM)
Resource	Warehouse	20-sim
Resource	Robotic Arm	Catia v6
Resource	Wagons	4DIAC
Resource	Test Station	4DIAC

Table 1. Technologies used for different constituent systems

Having already established a homogeneous co-simulation (with all FMUs generated from Overture), the improvements to various parts of the project can be achieved independently from each other, because any newly generated FMU would simply replace the corresponding Overture/VDM-RT FMU as long as the interfaces between units remained unchanged. Thus, the order of integration for completely-functional FMUs is determined by the progress of individual teams rather than pre-defined dependencies. Teams were able at any time to rely on a working DE-first co-simulation and only replace their unit. Furthermore, the co-simulation could at any time be assembled from any combination of FMUs (generated by Overture or other tools).

While for most units the FMU generated by the specialised program is simply an improvement (a more realistic or more detailed simulation) over the Overture model, the test case captures two exceptions: the HMI and the Part Tracker. The HMI is different from the rest of the units in the sense that the two FMUs are meant to complement each other, although they cannot be used both at the same time. The Overture/VDM-RT FMU reads the orders from a CSV file, and can be used for performing benchmark-like tests with completely controlled and repeatable sequences of orders. The 4DIAC-MQTT FMU implements user heuristics, allowing for real-time placement of orders and

real user interaction with the simulation. Furthermore, because orders can be requested from a smartphone or tablet, the 4DIAC-MQTT FMU also implements a graphical interface. On the other hand, since the Part Tracker handles the flow of the process, it is a logical unit rather than a physical one. Therefore, the Part Tracker is the only unit whose FMU is generated in Overture throughout the heterogeneous co-simulation phase, also being the only complete VDM model.

For one of the other units, the flexibility of the co-simulation allowed for a back-up plan to be used. As described in Table 1, the generation of an FMU for the Robotic Arm was intended be produced in Catia v6, of which unfortunately only the academic version of 2013 was available to the team. This presented a major problem, because the libraries necessary for generating an FMU were theoretically available only since to Catia V6 2015 FD01 (Dymola) and the support received from the producer of the tool was rather limited. Because of the modularity of the co-simulation, it was possible to use 20-sim to produce an alternative FMU and replace the envisaged Catia FMU model.

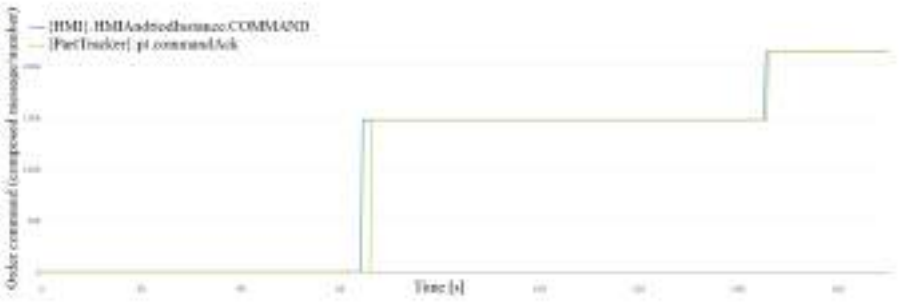


Fig. 6. Orders and acknowledgements exchanged between HMI and Part Tracker

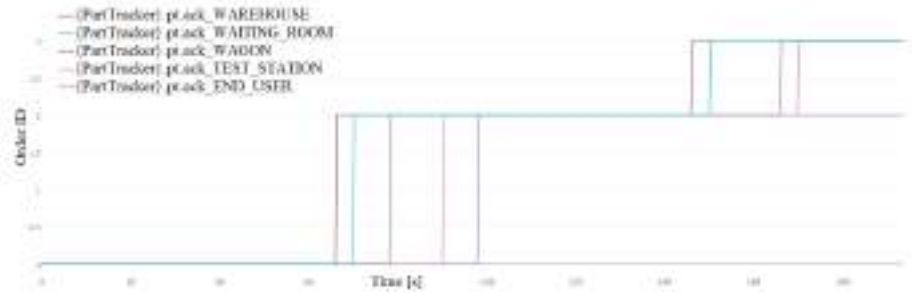


Fig. 7. Acknowledgement messages for stages of assembly sent by the Part Tracker

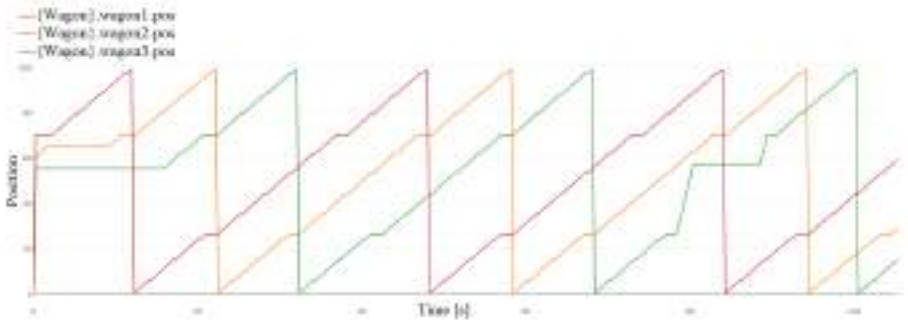


Fig. 8. Wagon positions on a circular track of length 100

The co-simulation in the heterogeneous phase allows for the analysis of interactions between the units that have been simulated using specialised programs. All messages exchanged between VDM FMUs (or their more rigorous versions) are available for display in the co-simulation engine and INTO-CPS Application. Figure 6 shows the communication between the HMI unit and the Part Tracker for two orders. Two orders are sent from the HMI and acknowledged by the Part Tracker. The progress of assembly on the orders is clearly noticeable in Figure 7 because acknowledgement messages are sent back to the HMI whenever the item reaches next stage. Furthermore, the acknowledgement messages contain the ID of the order, which would make the process easy to follow even in case of items being assembled in parallel, as if in a pipeline. Figure 8 shows the positions of the wagons on a circular track of length 100. The positions are communicated by the wagons themselves to the Part Tracker and the co-simulation engine eavesdrops on that communication line to render the positions.

5 Concluding Remarks and Future Work

This paper presented a methodology for modelling CPSs, as well as the steps for achieving a working heterogeneous co-simulation (with units modelled in various dedicated tools) for the development of an assembly line of USB sticks. The test case chosen was simple enough to be easily followed, but still sufficiently complex to allow the exploration of the capabilities and the limitations of both the methodology and the tools. The production line was first modelled completely in Overture/VDM-RT and the models were co-simulated using the INTO-CPS technology (a homogeneous co-simulation). The flexibility of the co-simulation engine allowed for the gradual integration of closer-to-reality and more detailed FMUs, generated in dedicated tools (4DIAC, 20-sim).

The test case also emphasised key possibilities of the methodology, such as:

– The initial development of the homogeneous co-simulation in VDM was particularly useful in driving cooperation and making clear the assumptions of the distributed teams involved in modelling the specific components; this phase proved to be the most difficult and time-consuming phase in building the co-simulation,

requiring a very intensive communication for a shared understating of the requirements;
- Once the VDM co-simulation is running, the independent development of units may be integrated and validated in the co-simulation in any order, and in any formalism, e.g. some units to remain modelled in Overture (in this case the Part Tracker and partially the HMI), while the others in their own formalisms in a co-simulation;
- An improved capability to handle some unpredictable requirements; the employment of co-simulations when designing an automated production system avoids the build-up inertia of subsequent design constraints, facilitating the low and late commitment for these decisions, i.e. the specific controllers or PLCs, the plant layout, the number of storage stacks from the warehouse, etc.

Further investigations in the co-simulation capabilities on the current test case include:

- The improvement of visualisation and debugging features for co-simulations;
- The inclusion of perturbations in the production line for a more realistic simulation;
- The optimisation of parameters for one or multiple units, depending on the perturbations, the amount and distribution of input orders and the physical constraints of the units (using the Design Space Exploration capabilities);
- The integration of FMUs generated by tools different from the ones mentioned in this paper (e.g. Catia for the robot arm); and
- The possibility of extending the purpose of the study to interactions between generated hardware and generated software solutions, in case the production line writing data onto the USB sticks and verifying it as part of the production process.

Acknowledgements

Our current work is partially supported by the European Commission as a small experiment selected for funding by the CPSE Labs innovation action (grant number 644400). The work presented here is also partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme (grant agreement number 664047).

References

1. Blochwitz, T.: Functional Mock-up Interface for Model Exchange and Co-Simulation. `https://www.fmi-standard.org/downloads` (July 2014)
2. Constantin B. Zamfirescu, Bogdan C-tin Pirvu, J.S.D.Z.: Preliminary Insides for an Anthropocentric Cyber-physical Reference Architecture of the Smart Factory. Studies in Informatics and Control 22(3), 269–278 (September 2013)
3. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains. In: FormaliSE: FME Workshop on Formal Methods in Software Engineering. ICSE 2015, Florence, Italy (May 2015)

4. Fitzgerald, J., Gamble, C., Payne, R., Larsen, P.G., Basagiannis, S., Mady, A.E.D.: Collaborative Model-based Systems Engineering for Cyber-Physical Systems – a Case Study in Building Automation. In: INCOSE 2016. Edinburgh, Scotland (July 2016)
5. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2014), http://link.springer.com/book/10.1007/978-3-642-54118-6
6. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: State of the art. Tech. rep. (feb 2017), http://arxiv.org/abs/1702.00686
7. Kleijn, C.: Modelling and Simulation of Fluid Power Systems with 20-sim. Intl. Journal of Fluid Power 7(3) (November 2006)
8. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
9. 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: CPS Data Workshop. Vienna, Austria (April 2016)
10. Larsen, P.G., Fitzgerald, J., Woodcock, J., Lecomte, T.: Trustworthy Cyber-Physical Systems Engineering, chap. Chapter 8: Collaborative Modelling and Simulation for Cyber-Physical Systems. Chapman and Hall/CRC (September 2016), iSBN 9781498742450
11. Larsen, P.G., Fitzgerald, J., Woodcock, J., Nilsson, R., Gamble, C., Foster, S.: Towards Semantically Integrated Models and Tools for Cyber-Physical Systems Design, pp. 171–186. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-47169-3_13
12. Mario Hermann, Tobias Pentek, B.O.: Design Principles for Industrie 4.0 Scenarios. In: 2016 49th Hawaii International Conference on System Sciences (HICSS). pp. 3928–3937 (2016)
13. Quadri, I., Bagnato, A., Brosse, E., Sadovykh, A.: Modeling Methodologies for Cyber-Physical Systems: Research Field Study on Inherent and Future Challenges. Ada User Journal 36(4), 246–253 (December 2015), http://www.ada-europe.org/archive/auj/auj-36-4.pdf
14. Strasser, T., Rooker, M., Ebenhofer, G., Zoitl, A., Sunder, C., Valentini, A., Martel, A.: Framework for distributed industrial automation and control (4diac). In: 2008 6th IEEE International Conference on Industrial Informatics. pp. 283–288 (July 2008)
15. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)

The Mars-Rover Case Study Modelled Using INTO-CPS

Sergio Feo-Arenis [1], Marcel Verhoef [1], and Peter Gorm Larsen [2]

[1] ESA/ESTEC, Keplerlaan 1, PO Box 299 NL-2200 AG Noordwijk, The Netherlands
{Sergio.Feo.Arenis,Marcel.Verhoef}@esa.int
[2] Aarhus University, Department of Engineering, Denmark
pgl@eng.au.dk

Abstract. The INTO-CPS project strives to preserve the diversity of disciplines such as software, mechatronic and control engineering, which have evolved notations and theories that are tailored to their engineering needs. We illustrate the utility of the proposed engineering approach to manage, track and monitor model artefacts used in collaborative heterogeneous modelling by applying it to ESA's Mars Rover case study, originating from earlier work performed in the DESTECS project. We describe the transition from the Crescendo technology to an INTO-CPS setting and report on the results and the challenges of our case study.

Keywords: Co-Simulation, Controller Design, Vehicle Dynamics

1 Introduction

In the development of Cyber-Physical Systems (CPSs) it is necessary to involve different disciplines with different kinds of formalisms and tools together in a heterogeneous setting [14]. One possible technique to see how such different kinds of models affect each other, is to use co-simulation [10]. This was for example enabled by the DESTECS project with the Crescendo tool [9]. However, this solution was fixed to always include only two models: one Discrete Event (DE) model expressed using the Vienna Development Method (VDM) [7] using the Overture tool [13] and one Continous-Time (CT) model expressed using bond graphs [11] and the 20-sim tool [12]. More recently, the INTO-CPS project has removed this limitation providing an open tool chain based on standards such as the Functional Mockup Interface (FMI) [3] and the Open Services for Lifecycle Collaboration (OSLC) [1].

At the European Space Agency (ESA) many systems fall into the CPS category. Here, the Mars Rover case study, which was proposed as in industrial challenge in the DESTECS project, was used as a trial example for the Crescendo technology. In DESTECS this worked very well, but unfortunately the CT model would have to be kept confidential. Thus, it was problematic to share this co-model with other parties. This could only be circumvented by running the co-simulation over the Internet, with the proprietary constituent CT model running at ESA. While technically feasible, this of course causes many other problems, such as allowing remote access through corporate firewalls, poor simulation performance, etc.

For this issue, FMI and the INTO-CPS technology offer a potential solution since the Functional Mockup Units (FMUs) produced for each constituent model do not necessarily need to contain the model itself. Thus, it is possible to protect the Intellectual Property (IP) in this manner. This paper reports on the attempt of migrating the Mars Rover co-model from Crescendo to the INTO-CPS technology. If this can be possible and the IP can be secured for the CT constituent model, this would enable openly sharing of the example with anyone.

The remaining part of this paper starts by introducing the co-simulation technologies in Section 2. This is followed by Section 3 describing the case study. Afterwards Section 4 presents the results of using the INTO-CPS co-simulation technology on that case study. Finally, Section 5 provides a collection of concluding remarks and looks into the future perspective for using this kind of technology in the space domain.

2 The Co-simulation Technologies

2.1 Crescendo

The Crescendo tool is the outcome of the European Framework 7 project Design Support and Tooling for Embedded Control Software (DESTECS), `www.destecs.org` which ran from early 2010 until 2013, which aim was to develop methods and tools that combine CT models of systems (in tools such as 20-Sim) with DE controller models developed using the Vienna Development Method (VDM) through co-simulation [19,17]. The approach was intended to encourage collaborative multidisciplinary modelling, including modelling of faults and fault tolerance mechanisms. The Crescendo tool in combination with Overture and 20-sim provides a platform for co-simulation. The platform only supports two simulators using a variable step size algorithm to progress simulation time. The intention in the DESTECS project was to have one controller which typically did not support roll-back and therefore have the plant model being in charge of providing future synchronization time intervals. In Crescendo the controller models are described in VDM using the Real-Time dialect which added timing information to the execution of each expression and statement [18]. The physical plant model was intended to be modelled using 20-sim in continuous time using bond-graphs, typically organised as block diagrams. The co-simulation is defined based on a contract which consists of *shared design parameters* used for sharing constant parameters between models, *monitored* variables used for providing inputs to the controller model, *controlled* variables which are controller outputs, and finally *events* which are used like interrupts in the controller [8,9].

2.2 The Functional Mock-up Interface

The tool independent standard Functional Mock-up Interface (FMI) was developed within the MODELISAR project[3]. The standard describes how simulation units are to be exchanged as ZIP archives called an *Functional Mock-up Unit (FMU)* and how the

[3] See `https://itea3.org/project/modelisar.html`.

model interface is described in an XML file named *modelDescription.xml*. The functional interface of the model is described as a number of C functions that must be exported by the library that implements the model inside the FMU. Since the FMU only contains a binary implementation of the model, it offers some level of IP protection. The standard lists a number of constraints on how the C functions in FMI can be called but no explicit algorithm.

2.3 The Integrated Tool Chain for Model-based Design of Cyber-Physical Systems (INTO-CPS) Technology

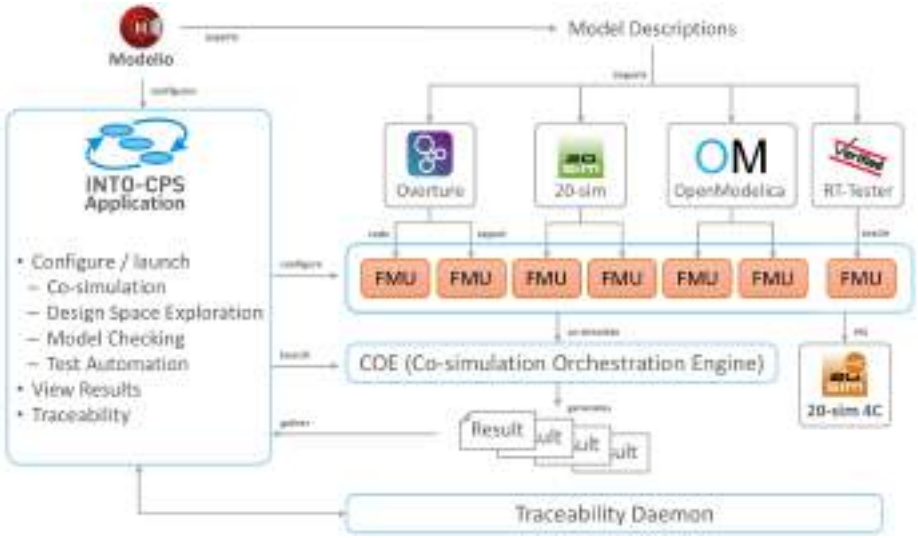


Fig. 1. Overview of the INTO-CPS toolsuite.

The vision of the INTO-CPS [4] consortium is that CPS engineers should be able to deploy a wide range of tools to support model-based design and analysis, rather than relying on a single "factotum". The goal of the project is to develop an integrated "tool chain" that supports multidisciplinary, collaborative modelling of Cyber-Physical Systemss (CPSs) from requirements, through design, to realisation in hardware and software, enabling traceability through the development. The project integrates existing industry-strength baseline tools in their application domains, based centrally around FMI-compatible co-simulation [2]. The project focuses on the pragmatic integration of these tools, making extensions in areas where a need has been recognised. The tool chain is underpinned by well-founded semantic foundations that ensures the results of analysis can be trusted [20,4].

[4] See `http://into-cps.au.dk/`.

The toolchain provides powerful analysis techniques for CPS models, including generation and static checking of FMI interfaces; model checking; Hardware-in-the-Loop (HiL) and Software-in-the-Loop (SiL) simulation, supported by code generation. These features enabled both Test Automation (TA) and Design Space Exploration (DSE) of CPSs.

The INTO-CPS project provides a lightweight application interface for managing the development of constituent system models for co-simulations and for configuring and running such co-simulations. The project offers the following FMI enabled modelling tools: Overture, 20-sim, OpenModelica and RT Tester. However, it is an open toolchain so in general any FMI 2.0 enabled tool can be used. An overview of the components of the toolchain is shown in Fig. 1.

The project also provides high-level system design using SysML through the Modelio tool [16]. A co-simulation configuration can be modelled in SysML using block diagrams and a custom INTO-CPS profile for FMI. The co-simulation in INTO-CPS supports FMI version 2.0 and the master algorithm supports both fixed and variable step sizes. The variable step size algorithm is extended with support for the `getMaxStepSize` method from [5], it also supports constraints to adjust the step size.

The Overture FMU Extension The Overture tool is extended with a FMU exporter that supports FMI 2.0. The exporter relies on a VDM FMI library and modelling guide line. The library consists of a `Port` class and the following specialisations: `BoolPort`, `IntPort`, `RealPort`, and `StringPort` each mapping to the corresponding scalar variable type in FMI. The tool requires the user to follow a design pattern where an instance of a `HardwareInterface` is placed in the `System` class, and a `World` class is responsible for blocking the initial thread to ensure infinite execution. This class must contain all ports and custom annotations to set the type and name of the exported port. In Listing 1.1 is a small example of three ports exported as a parameter, input and output.

```
class HardwareInterface
values
  -- @ interface: type = parameter;
  public v : RealPort = new RealPort(1.0);

instance variables
  -- @ interface: type = input;
  public distanceTravelled : RealPort := new RealPort(0.0);
  -- @ interface: type = output;
  public setAngle : RealPort := new RealPort(0.0);

end HardwareInterface
```

Listing 1.1. The `HardwareInterface` class

The FMI integration uses this special `HardwareInterface` class where parameters, inputs and outputs are modelled through a number of port classes: `BoolPort`, `IntPort`, `RealPort`, `StringPort`. These all have `getValue` and `setValue`

methods. Through the explicit modelling of FMI scalar variables as ports it is possible to generate an FMU without requiring additional user input. A simple mapping can then be made automatically (from *shared design parameters* `sdp` into parameters, `controlled` into input on the plant side and `monitored` output on the controller). The key difference with Crescendo is, however, the absence of support for `events`, as FMI does not support this concept. In FMI there is no ability to trigger data exchanges asynchronously; they must be emulated by a polling approach.

20-sim FMU Extension The 20-sim tool is extended with FMI support through a custom code generator template [5] that is able to export a module in a 20-sim model as a FMU. This solution does not support event detection and external library usage (Dynamic Link Library (DLL)) which can be used for e.g. event detection. It has support for the following integration methods: Discrete Time, Euler, Runge Kutta 2, Runge Kutta 4, CVODE [6], and MeBDFi [7].

This limitation can be overcome by generating a so-called FMI tool wrapper, which basically bundles the model with all resources needed to execute the FMU on a locally installed copy of 20-sim. The down side of this approach is that it does not provide the protection of the intellectual property, as the the source model is shared in its entirety.

3 The Mars Rover Case study

Our case study treats the model-based development of a planetary rover in the context of INTO-CPS, based on an existing co-simulation model developed by the second co-author, at that time part of the DESTECS project. Planetary rovers are mobile robots designed to carry and provide mobility to scientific instruments on the surface of planets. ESA's ExoMars mission, due to launch in 2020, includes a rover that will provide key mission capabilities: surface mobility, subsurface drilling and automatic sample collection, processing, and distribution to instruments (see Fig. 2).

Of particular interest and complexity is the rover's ability to navigate with obstacles. In the past, 20-sim models have been made for two potentially interesting suspension systems. A novel approach is to use additional tilt drives with steerable shoulders and beams to allow the rover to wheel-walk. Several types of gaits have been investigated, each with different influence on aspects like traversal velocity, efficiency, rover stability, etc. However, it was not possible to combine basic gaits into an actual trajectory to be followed. In DESTECS, a VDM model was developed of the supervisory controller for the rover, that allowed to create a co-model to study mission level usage scenarios.

The purpose of the experiment described in this paper is to assess whether the DESTECS results can be reproduced using INTO-CPS. The actual analysis of the simulation results obtained is out of scope for this paper, partly because that information is protected under an NDA allowing us to describe it only in general terms. Nevertheless,

[5] The FMU generator template for 20-sim is available at `https://github.com/controllab/fmi-export-20sim/`.

[6] `https://computation.llnl.gov/projects/sundials/cvode`

[7] `http://www.netlib.org/ode/mebdfi.f`)

Fig. 2. Artist's impression of the ExoMars 2020 rover. Source: ESA/ATG medialab

we will provide an overview of the model in order for the reader to gain an understanding of the level of detail and complexity of the models that can be dealt with by using this approach.

3.1 Description of the DESTECS Co-Model

In DESTECS, a co-simulation model was created that integrates the continuous vehicle dynamics, contained in a 20-sim model, and the discrete controller behavior modelled using VDM-RT. These are detailed in the following sections.

Continuous-Time Model The physical model includes the rover platform extended with a suspension system, actuators and sensors, in particular an Inertial Measurement Unit (IMU) that allows the rover to estimate its attitude.

In order to model the interaction of the rover's wheels with the ground and other obstacles that may be placed in the simulation, the model is equipped with 3D power ports. A power port in 20-sim is a connection point of a model, allowing energetic interaction between model parts. To allow a fully dynamic interaction, each power port consists of bi-directional conjugate signals indicating the force between parts and their differential velocity. This interface has been used to implement the contact model with the surface and the obstacles, using a dedicated solver.

An overview of the components of the vehicle dynamics model is shown in Fig. 3. The central component is a description of the geometry of the rover's mechanical components such as the payload platform, the suspension beams, the shoulder joints, the steering assemblies and the wheels. Contained in that part are also their interaction parameters, such as deflection limits for articulated connections and force characteristics of the drive motors. Attached to the geometric model are blocks that represent individual actuator controllers, typically PID loops, which receive input signals from the central controller and effect the drive motors to achieve the vehicle's motion. Finally, corresponding blocks to represent sensors are attached to allow observing the simulation's progress. The same information is used to feed the 3D visualisation, both during simulation as well as in post-simulation replay mode.

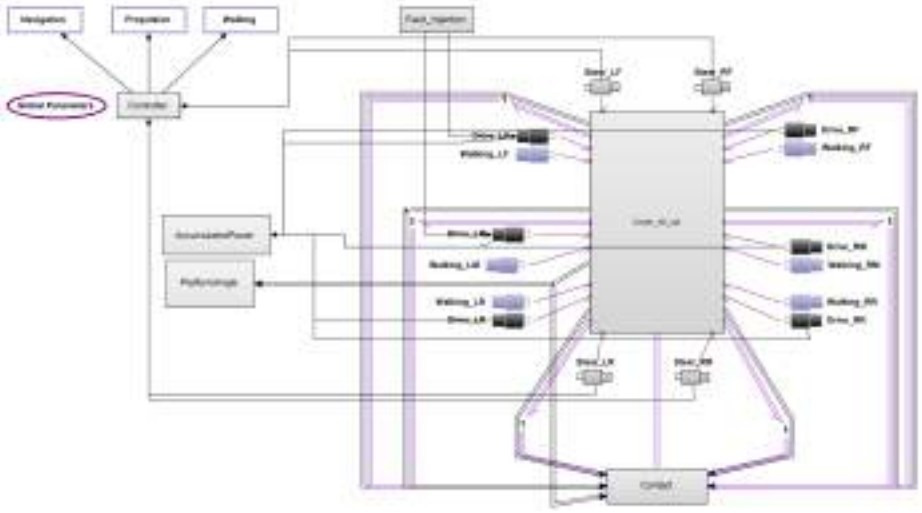


Fig. 3. Block diagram of the 20-sim rover co-simulation model. The central component is the dynamics model containing the rover geometry. Around it, simulated drives model the wheel drive and steering motors and the energy transfer to the ground by means of a contact model. Observable are the platform angles and the accumulated power usage. There is the possibility of injecting faults into the left front and middle wheels.

Discrete Event Model In VDM-RT, a simple model was created that reflects the structure of the physical components using the object-oriented features of the language. So, the Mars rover model in VDM-RT consists of six so-called DriveTrain objects, which are each composed of a Wheel, Beam and Steering object (see Fig. 5 for an overview). The Wheel object is used to control the setpoint of the wheel rotation, the Beam object is to control the setpoint of the shoulder movement and finally the Steering object is used to rotate the beam along its longitudinal axis.

The knowledge of how to set a particular gait mode is simplified to setting properties to the top-level DriveTrain objects, which will delegate this task to the underlying component objects. A trajectory can be composed by construction of a plan, which contains basic movements such as stop, wait, move forward, perform point turn, perform Ackermann turn, and so on. This simplifies the model enormously when compared to an implementation in 20-sim directly, which, among other benefits, provides modeling flexibility. Additionally, the DE model is extended with an independent supervisory control mechanism, a "safety monitor" that checks whether the roll angle of the vehicle remains within a specified range and initiates emergency countermeasures when the range is exceeded in order to guarantee the safety of the rover platform, preventing it from tumbling over. In case the safety monitor triggers, the current plan is aborted, the rover reverses the last two performed steps, rotates left 15 degrees and resumes the original plan.

Furthermore, the consistency mechanisms available in VDM-RT, such as pre- and post conditions, are used to ensure the integrity of the gait model during simulation. This

implies that the simulation is immediately stopped, such that the state of the model can be analysed in order to determine the root cause of the trapped inconsistency.

The main goal of this case study is the migration of the co-simulation developed by the DESTECS project to work with the INTO-CPS Framework in order to evaluate the benefits and challenges of the new toolchain.

3.2 Moving Towards the INTO-CPS Multi-Model

In order to adapt the existing multi-model to work with the INTO-CPS toolsuite, it was initially necessary to modify both the Continous-Time (CT) and Discrete Event (DE) models to enable the generation of a FMU. After generating FMUs, they were imported into the SysML architectural editor in order to create the specification of the data exchange between models. Finally, the results were integrated into a co-simulation configuration that can be executed by the Co-simulation Orchestration Engine (COE) to obtain simulation results. We provide additional details for those activities.

Modifications for FMU Generation In order to accommodate the particularities of the FMI specification, and of the specific tool extensions to support it (see Sect. 2.3), additional modelling of an interface (the "hardware interface") that encapsulates the data items exchanged between the co-simulation components, together with the direction of said exchange (input or output).

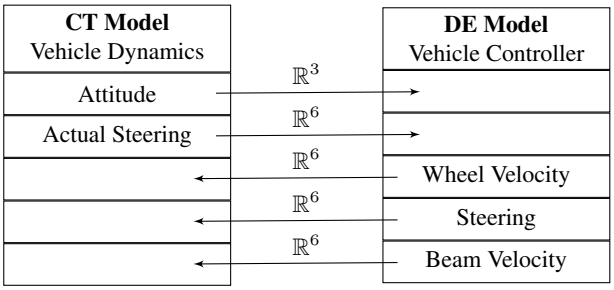


Fig. 4. Overview of the information exchange in the co-simulation model.

In Fig. 4, we show the data ownership and the direction of information exchange between the models. The CT model *owns* (i.e., controls and outputs) the data items related to the vehicle's attitude (3 real numbers representing the roll, pitch and yaw angles) and the actual steering positions of the individual wheel assemblies (6 real numbers representing the rotation angles for each wheel). The CT model also receives as input all the setpoint values output by the controller, represented by the DE model. Conversely, the DE model receives as input the values owned by the CT model, and owns the data pertaining the setpoints of the rover hardware: individual wheel velocity values (6 reals representing the commanded speed for each wheel), steering values (6 reals representing the required wheel steering angles), and beam velocity values (6 reals representing the commanded angular velocity of the shoulder joints).

Listing 1.2. Excerpt of the interface declaration of the CT model in 20-sim

```
externals
  // External variables
  real global export Actual_Steering_1;
  real global export Actual_Steering_2;
  real global export Actual_Steering_3;
  ...
  real global import steering_1;
  real global import steering_2;
  real global import steering_3;
  ...
equations
  ...
```

The custom FMU generator integrated into 20-sim (see Sect. 2.3) relies on the declaration of *external* variables, belonging to the exchange interface with other models. External variables are labeled with either the *import* or *export* keywords to indicate the direction of the exchange. A snippet of the corresponding declarations is shown in Listing 1.2.

Note that although the models effectively exchange vectors, all vector components must be declared explicitly, with individual variable names. This is due to a fundamental limitation of the FMI specification: only basic types (integer, boolean, floating point and string) are allowed [3].

We fitted the model with the necessary external variable declarations. Additionally, we took the opportunity to perform an upgrade of the collision detection library used for simulation. The model was refactored to target a newer, improved version of the Bullet Physics Engine[8]. Finally, we obtained a packaged tool wrapper FMU generated automatically by the corresponding 20-sim extension.

On the discrete event (DE) side, some modifications to the VDM-RT model were also necessary. The provided model already contained a model of the rover abstracting the sensors and actuators of the vehicle, together with a controller capable of executing a maneuver plan and equipped with a safety monitor (see Sect. 3.1).

The extension for the Overture tool offers an option to integrate an FMU library into any model. The library contains class definitions that represent *ports* for each of the allowable exchange data types. Based on those definitions, we introduced a new `HardwareInterface` class (similar to the one shown in Listing 1.1) containing the 27 variables shared by the components of our case study. We then followed the pattern outlined in Sect. 2.3 by adding an instance of the hardware interface model to the `System` class and creating a specialization of the provided `RobotController` class to include a mapping of the hardware interface ports to the internal variables of the controller. Finally, every controller step was augmented with additional operations to read from the interface ports before computing and to write the controller outputs to the corresponding interface ports. An excerpt of the overall model is shown in Fig. 5 as a class diagram. The classes added during the migration to the INTO-CPS toolchain are highlighted by a blue rectangle.

[8] See `http://bulletphysics.org/wordpress/`.

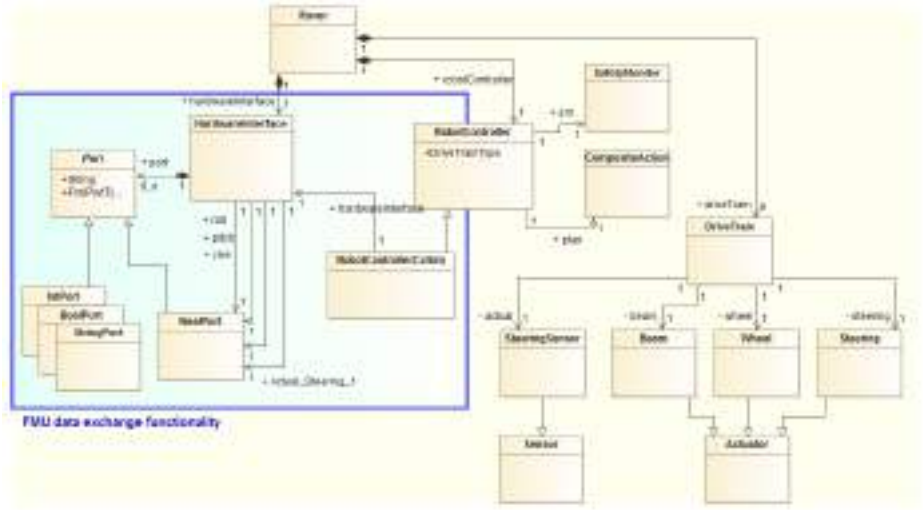


Fig. 5. Class diagram of the resulting DE model. Some of the classes and attributes added for compatibility with the FMI approach are highlighted.

Having generated an FMU from the resulting DE model by using the Overture extension, we proceeded to integrate it with the FMU of the CT model. In order to aid this process, we created a SysML specification of the multi-model architecture in Modelio. There, the generated FMUs can be imported automatically into a block diagram. The connections between the interface ports of the models can be then specified graphically by drawing arrows from an output port of a model to the corresponding input port of the other model.

From the architectural model, a multi-model configuration file can be generated automatically via the custom INTO-CPS profile for FMI. All artifacts together (the models, their generated FMUs and the co-simulation configuration) constitute a project in the INTO-CPS toolsuite.

From the multi-model, a co-simulation configuration can be created. Here, simulation parameters such as the step size, simulation time, step algorithm can be specified. Also, a set of exchange variables can be chosen for graphical visualization. The COE can then be launched and a simulation started from the user interface.

4 Comparison of Co-simulation Results

To verify the correctness of the migrated multi-model with respect to the initial models and to assess the performance of the toolchain, various tests were performed.

Prominently, we ran co-simulations of the Mars Rover negotiating a series of simulated obstacles in order to compare the behavior of the migrated models with the initial models. In general, we observed similar results. The simulation outcomes are however, not identical on the continuous-time side due to the model upgrades and tuning

performed. A visual inspection allowed us to nevertheless determine the correct functioning of the multi-model and to attest its utility for simulation and testing purposes. Also, we were able to verify the correct functioning of the safety monitor feature of the simulated controller while operating as an FMU.

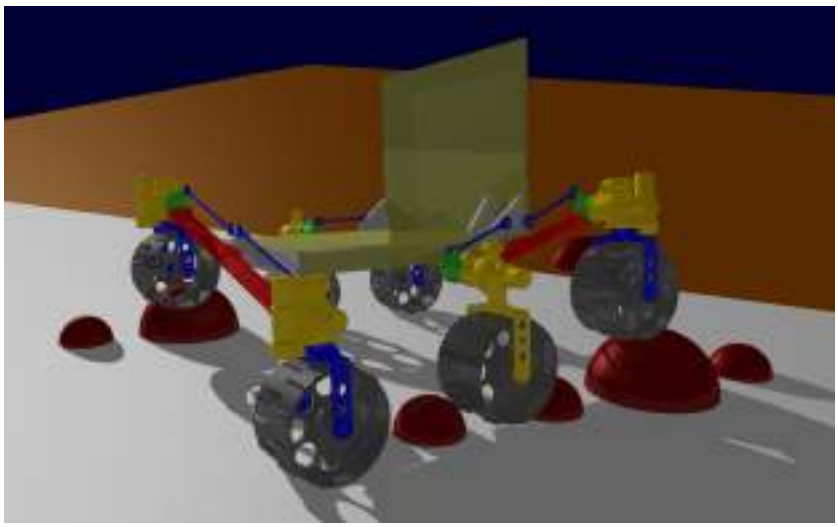


Fig. 6. 3D visualization of the Mars Rover simulation showing the effects of moving over simulated obstacles including the orientation of the rover platform.

A snapshot of the simulated Mars Rover driving over simulated obstacles during co-simulation is shown in Fig. 6. This visualization is displayed as the simulation is executed in near real-time, providing feedback to the modelling engineer regarding the simulation outcome. An excerpt of the physical variables logged during co-simulation, represented as line graphs is shown in Fig. 7. This data can be exported for archival and further analysis.

Overall, we estimate the effort required for model migration to not exceed 20 man-hours, not including the the time spent by the tool developers to fix the issues encountered during the activity.

4.1 Challenges Encountered

Adapting the existing models to work with the INTO-CPS toolchain posed several challenges. First, the temporal difference between the co-simulation undertakings made it difficult to reproduce the results of the provided models using current versions of the tools. Due to tool and language evolution cycles, the models, left without maintenance for some time, were no longer fully compatible with the current versions of the VDM and 20-sim modelling environments. An additional effort was necessary to update the models and ensure their functioning with current versions of the tools.

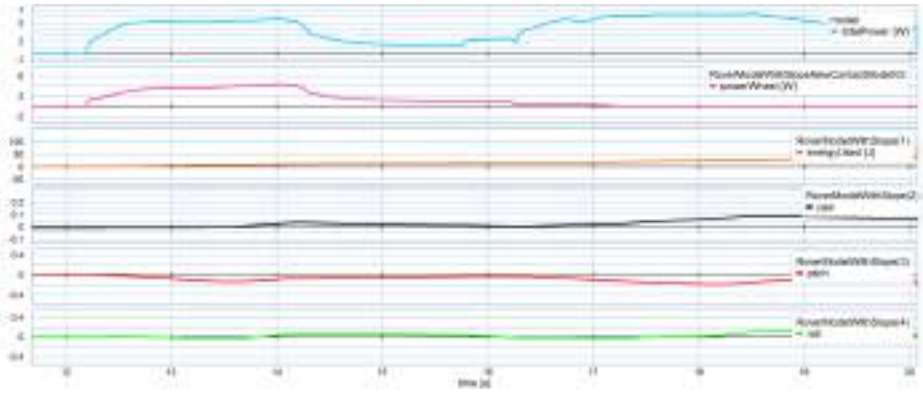


Fig. 7. Variable plots of the Mars Rover simulation. They show the evolution of the values of simulated variables over time. In this example, accumulated power consumption in Watts of the simulated rover, instantaneous poewr applied to the wheels, total energy used in Joules and finally the yaw, pitch and roll of the rover body.

Concretely, language features of VDM-RT were modified in recent updates. Identifiers used in the original model such as **`stop`** became reserved keywords; also, the default visibility of member variables became more restrictive, prompting a refactoring of the DE model.

Conversely, due to the upgrades performed on the CT model, specifically in the collision detection interface, additional tuning of the model parameters was necessary in coordination with the tool vendor. Additional support was requested and provided which resulted in a longer than expected time frame for the migration activity.

As a positive side effect of the activity, issues in the simulator (collision detection interface and FMU generation scripts) and the Modelio tool (FMU import function) were discovered, enabling us to provide feedback to the corresponding developers. Updated versions of the tools including the requested fixes will be made publicly available.

5 Concluding Remarks and Future Works

In summary, we have successfully migrated the DESTECS Mars Rover case study to the INTO-CPS tool chain, without loss of fidelity. However, we have encountered a few issues that would potential hamper the successful usage of INTO-CPS into our context:

1. In INTO-CPS, or rather in the FMI standard, the notion of *events* is not supported, as compared to Crescendo. Although this feature is currently not used in the Mars Rover case study, and hence not blocking us in this particular case, it still can be seen as a step back in expressive power.

2. The restriction to scalar types of the FMI specification is a challenge moving forward towards the more complex multi-models supported by the INTO-CPS toolchain. Although the workaround of explicitly declaring vector components for this

specific case study did not represent a significant effort, it may complicate the inclusion of larger, additional models as the number of variables of the exchange interfaces increases. We thus proposed two possible solutions for overcoming that limitation. First, adopting a convention for *name mangling* that would allow FMU generation extensions to create and update interfaces for vector values. Second, integrating *encoding* and *decoding* routines in the generated FMUs that would allow encapsulating vector types in string-typed scalar variables.

3. Specifically for the use of 20-sim, it is currently not possible to create self-standing FMUs in case the CT model relies on external DLLs. Even though the tool wrapper FMI is available and is demonstrated to work in this case study, it does not resolve the issue of protecting the Intellectual Property (IP) of the model. However, from a technical perspective, we believe that it must be possible to also create FMUs that call external DLLs. Hence this limitation might be resolved in the future.

The ability to run the MarsRover case in the context of INTO-CPS provides us with the following advantages:

1. **Management of IP and ease of model construction**.
   The ability to contain the IP of a model inside an FMU is an important enabler for industrial system development. In our case study, the aim was to separate the plant model (owned by ESA) from the controller model (developed by an external supplier). It should be noted that in particular in the European space market, typically a multitude of suppliers are involved in any mission, with the systems engineering (integration and validation) responsibility assigned to a prime contractor, leaving ESA in a supervisory and oversight role. In this case we sought to provide an executable reference model without disclosing the internal design of the plant model in order to objectively compare proposed controller designs.
   It is easy to understand that communication between all these stakeholders is important. In particular, scientific space missions are typically performed on the edge of what is technically possible; we are flying prototypes essentially, as most spacecraft are built as "one-off" products. However, these prototypes must of course have a high reliability. The only means to demonstrate that reliability prior to launch is to build end-to-end mission simulators and to gradually build up our confidence by increasing the fidelity of those models. This is typically done by replacing models by actual (sub)systems to demonstrate their fitness for use. Managing this process is complex, error prone and therefore very costly. Downstream suppliers understand the need to build such simulators, but they also want to protect their specific intellectual property in order to maintain their competitive edge. Usually, the higher the model fidelity, the more important IP protection becomes. Therefore, the ability to couple models of (sub)systems into a distributed simulation setting easily allows us to adapt to changes (and therefore quickly repeat validation) without having to disclose the proprietary nature of the co-model, which is enabled by the FMI.

2. **System of systems mission analysis**.
   The second advantage of INTO-CPS over the DESTECS approach is that it enables

us to perform system-of-systems type simulations, by combining several models into the same co-simulation environment. In the case of the Mars Rover for example, the rover cannot operate without support from a satellite orbiting the planet, which acts as a relay station for all telecommanding and telemetry between the ground and the rover. Moreover, the satellite can use its on-board cameras to make detailed images of the terrain surrounding the rover, such that it can be used as part of the waypoint planning (target of interest selection and obstacle avoidance). Similarly, several new missions are currently on the drawing board that consist of multiple spacecraft working closely together to achieve their mission goals. For example, Proba-3 will observe the corona of the Sun by having the telescope being blocked from direct sunlight by another satellite moving in the line of sight. This requires guidance and navigation control accuracy in the millimeter range, while the two satellites are flying roughly 150 meter away from each other. These kind of system-of-systems missions require competent co-models for both satellites at a fairly detailed level of fidelity in order to study the behavior of the constellation as a whole. It is also clear from this example that the behavior of the constellation is considered an entity in its own right, so just combining two separate (DESTECS style) co-simulations would not be sufficient.

3. **Validating on-board software**.
   The third advantage of the FMU approach is that it becomes easier to validate on-board software that is under development. Currently, Numerical Software Validation Facilities (NSVF) and Avionics Test Benches (ATB) are used to demonstrate the fitness for use of the software. But these facilities come into play rather late in the development process, while the FMU approach allows to integrate on-board software artifacts much earlier. In the context of this case study, we are working on synthesizing software directly from the DE models, using the vdm2c code generator, and deploying these artifacts using the TASTE software development environment and runtimes [15,6]. The result is a binary application which can be included as an FMU under the INTO-CPS co-simulation execution engine. We hope to be able to demonstrate this capability during the workshop.

Acknowledgments

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. We would in particular like to thank Frank Groen of Controllab Products for his support with the 20-sim part of our co-model.

References

1. Open Services for Lifecycle Collaboration (OSLC). `http://open-services.net/`
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 (September 2012)

3. Blochwitz, T.: Functional Mock-up Interface for Model Exchange and Co-Simulation. https://www.fmi-standard.org/downloads (July 2014)
4. Cavalcanti, A., Woodcock, J., Amálio, N.: Behavioural models for fmi co-simulations. In: Sampaio, A.C.A., Wang, F. (eds.) International Colloquium on Theoretical Aspects of Computing. Lecture Notes in Computer Science, Springer (2016)
5. Cremona, F., Lohstroh, M., Broman, D., Natale, M.D., Lee, E.A., Tripakis, S.: Step revision in hybrid co-simulation with FMI. In: MEMOCODE. pp. 173–183. IEEE (2016)
6. Fabbri, T., Verhoef, M., Bandur, V., Perrotin, M., Tsiodras, T., Larsen, P.G.: Towards integration of Overture into TASTE. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 94–107. Aarhus University, Department of Engineering, Cyprus, Greece (November 2016), ECE-TR-28
7. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. Wiley Encyclopedia of Computer Science and Engineering (2008), edited by Benjamin Wah, John Wiley & Sons, Inc.
8. Fitzgerald, J., Larsen, P.G., Pierce, K., Verhoef, M.: A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems. Mathematical Structures in Computer Science 23(4), 726–750 (2013)
9. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2014), http://link.springer.com/book/10.1007/978-3-642-54118-6
10. Gomes, C., Thule, C., Broman, D., Larsen, P.G., Vangheluwe, H.: Co-simulation: State of the art. Tech. rep. (feb 2017), http://arxiv.org/abs/1702.00686
11. Karnopp, D., Rosenberg, R.: Analysis and Simulation of Multiport Systems: the bond graph approach to physical system dynamic. MIT Press, Cambridge, MA, USA (1968)
12. Kleijn, C.: Modelling and Simulation of Fluid Power Systems with 20-sim. Intl. Journal of Fluid Power 7(3) (November 2006)
13. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
14. Lee, E.A.: Cyber Physical Systems: Design Challenges. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley (Jan 2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html
15. Perrotin, M., Grochowski, K., Verhoef, M., Galano, D., Mosdorf, M., Kurowski, M., Denis, F., Graas, E.: TASTE In Action. In: 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016). Toulouse, France (January 2016)
16. Quadri, I., Bagnato, A., Brosse, E., Sadovykh, A.: Modeling Methodologies for Cyber-Physical Systems: Research Field Study on Inherent and Future Challenges. Ada User Journal 36(4), 246–253 (December 2015), http://www.ada-europe.org/archive/auj/auj-36-4.pdf
17. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2009)
18. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. pp. 147–162. Lecture Notes in Computer Science 4085, Springer-Verlag (2006)
19. Verhoef, M., Visser, P., Hooman, J., Broenink, J.: Co-simulation of Distributed Embedded Real-time Control Systems. In: Davies, J., Gibbons, J. (eds.) Integrated Formal Methods: Proc. 6th. Intl. Conference. pp. 639–658. Lecture Notes in Computer Science 4591, Springer-Verlag (July 2007)
20. Woodcock, J., Foster, S., Butterfield, A.: Heterogeneous Semantics and Unifying Theories, pp. 374–394. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-47166-2_26