View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Tidsskrift.dk (Det Kongelige Bibliotek)

PROCEEDINGS OF THE 11TH OVERTURE WORKSHOP

Electrical and Computer Engineering
Technical Report ECE-TR-17

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA SHEET

Title: Proceedings of the 11th Overture Workshop
Subtitle: Electrical and Computer Engineering
Series title and no.: Technical report ECE-TR-17

Editors: Ken Pierce and Stefan Hallerstede
Department of Engineering – Electrical and Computer Engineering,
Aarhus University

Internet version: The report is available in electronic format (pdf) at the Department of Engineering website http://www.eng.au.dk.

Publisher: Aarhus University©
URL: http://www.eng.au.dk

Year of publication: 2013 Pages: 68
Editing completed: November 2013

Abstract: The 11th Overture Workshop was held in Aarhus, Denmark on Wed/Thu 28–29th Au- gust 2013. It was the 11th workshop in the current series focusing on the Vienna De- velopment Method (VDM) and particularly its community-based tools development project, Overture (http://www.overturetool.org/), and related projects such as COMPASS (http://www.compass-research.eu/) and DESTECS (http: //www.destecs.org). Invited talks were given by Yves Ledru and Joe Kiniry. The workshop attracted 25 participants representing 10 nationalities.
The goal of the workshop was to provide a forum to present new ideas, to identify and encourage new collaborative research, and to foster current strands of work towards publication in the mainstream conferences and journals. The Overture initiative held its first workshop at FM'05. Workshops were held subsequently at FM'06, FM'08 and FM'09, FM'11, FM'12 and in between.

Keywords: software engineering and systems

Please cite as: Ken Pierce and Stefan Hallerstede (Editors), 2013. Proceedings of the 11th Overture Workshop. Department of Engineering, Aarhus University. Denmark. 68 pp. - Technical report ECE-TR-17

Cover image: Logo, Overture Open Source Community

ISSN: 2245-2087

Reproduction permitted provided the source is explicitly acknowledged

PROCEEDINGS OF THE 11TH OVERTURE WORKSHOP

Ken Pierce and Stefan Hallerstede
Aarhus University, Department of Engineering

Abstract

The 11th Overture Workshop was held in Aarhus, Denmark on Wed/Thu 28–29th August 2013. It was the 11th workshop in the current series focusing on the Vienna Development Method (VDM) and particularly its community-based tools development project, Overture (www.overturetool.org/), and related projects such as COMPASS (www.compass-research.eu/) and DESTECS (www.destecs.org). Invited talks were given by Yves Ledru and Joe Kiniry. The workshop attracted 25 participants representing 10 nationalities.

The goal of the workshop was to provide a forum to present new ideas, to identify and encourage new collaborative research, and to foster current strands of work towards publication in the mainstream conferences and journals. The Overture initiative held its first workshop at FM'05. Workshops were held subsequently at FM'06, FM'08 and FM'09, FM'11, FM'12 and in between.

Introduction

The 11th Overture Workshop was held in Aarhus, Denmark on Wed/Thu 28–29th August 2013. It was the 11th workshop in the current series focusing on the Vienna Development Method (VDM) and particularly its community-based tools development project, Overture (`http://www.overturetool.org/`), and related projects such as COMPASS (`http://www.compass-research.eu/`) and DESTECS (`http://www.destecs.org`). Invited talks were given by Yves Ledru and Joe Kiniry. The workshop attracted 25 participants representing 10 nationalities.

The goal of the workshop was to provide a forum to present new ideas, to identify and encourage new collaborative research, and to foster current strands of work towards publication in the mainstream conferences and journals. The Overture initiative held its first workshop at FM'05. Workshops were held subsequently at FM'06, FM'08 and FM'09, FM'11, FM'12 and in between.

The workshop organisers and editors of these proceedings were:

- Ken Pierce (Newcastle University, UK)
- Stefan Hallerstede (Aarhus University, Denmark)

List of Participants

Ken Pierce Newcastle University, UK.
Martin Mansfield Newcastle University, UK.
John Fitzgerald Newcastle University, UK.
Peter Gorm Larsen Aarhus University, Denmark.
Stefan Hallerstede Aarhus University, Denmark.
Peter Würtz Vinther Jørgensen Aarhus University, Denmark.
Sune Wolff Aarhus University, Denmark.
Kenneth Lausdahl Aarhus University, Denmark.
José Antonio Esparza Isasa Aarhus University, Denmark.
Luis Diogo Couto Aarhus University, Denmark.
Joey Coleman Aarhus University, Denmark.
Martin Peter Christiansen Aarhus University, Denmark.
Jesper Gaarsdahl Aarhus University, Denmark.
Sergi Rotger Griful Aarhus University, Denmark.
Sren Mikkelsen Aarhus University, Denmark.
George Kanakis Aarhus University, Denmark.
Morten Larsen Conpleks Innovation, Denmark.
Nico Plat West Consulting BV, The Netherlands.
Hiroshi Ishikawa Niigata University of International and Information Studies (NUIS), Japan.
Mads von Qualen Terma A/S, Denmark.
Klaus Kristensen Bang & Olufsen, Denmark.
Uwe Schulze University of Bremen, Germany.
Lasse Lorenzen CLC bio, Aarhus, Denmark.
Yves Ledru Laboratoire d'Informatique de Grenoble (LIG), France.
Joe Kiniry Technical University of Denmark (DTU), Denmark.

Table of Contents

Introduction 3

List of Participants 5

Table of Contents 7

The Overture Approach to VDM Language Evolution **8**
Nick Battle, Anne Haxthausen, Sako Hiroshi, Peter W. V. Jørgensen, Nico Plat, Shin Sahara, and Marcel Verhoef

An Architectural Evolution of the Overture Tool **16**
Peter W. V. Jørgensen, Kenneth Lausdahl, and Peter Gorm Larsen

Towards an Overture Code Generator **22**
Peter W. V. Jørgensen and Peter Gorm Larsen

The COMPASS Proof Obligation Generator **28**
Luis Diogo Couto and Richard Payne

Model Based Testing of VDM Models **34**
Uwe Schulze

Co-modelling of a Robot Swarm with DESTECS **42**
Ken Pierce

Modelling Systems of Cyber-Physical Systems **48**
Martin Mansfield and John Fitzgerald

Modelling Different CPU Power States in VDM-RT **56**
José Antonio Esparza Isasa and Peter Gorm Larsen

Modelling a Smart Grid System-of-Systems using VDM **63**
Stefan Hallerstede and Peter Gorm Larsen

The Overture Approach to VDM Language Evolution

Nick Battle, Anne Haxthausen, Sako Hiroshi, Peter Jørgensen,
Nico Plat, Shin Sahara, and Marcel Verhoef

The Overture Language Board

Abstract. The Overture Language Board (LB) has a strategic role in the development of the VDM-10 Languages, VDM-SL, VDM++ and VDM-RT, and deals in particular with Requests for Modifications (RMs) to the language. Such requests come usually from participants in the Overture project. This paper describes how the LB uses a well-defined process with several phases to deal with the RMs, from when they are requested until they are either rejected or accepted and implemented. The paper also gives an overview of language changes that have been accepted and implemented in the period April 2009 – June 2013.

Keywords: VDM, formal specification languages, language design, language modification process, The Overture Language Board

1 Introduction

Languages evolve, whether these be natural languages, programming languages or formal specification languages (or any other languages for that matter). Obviously the same holds for what is now known as the VDM-10 family of formal specification languages. New needs arise as a result of new insights, and application of the language in new areas also requires the definition of new language constructs or the revision of previous ones.

VDM "as a language" dates back to 1970, and was called VDL (Vienna Definition Language) at that time. VDL later evolved into languages or dialects such as "Meta-IV" [1] and the "VDM notation" [7] in the late eighties.

The next milestone in the development of VDM was the production of the ISO VDM-SL standard [6,10]. An official ISO standard was put forward, after a rather lengthy effort, coordinated by the BSI (British Standardization Institute), including all aspects of language definition, such as lexis, syntax, and semantics. This was done for the so-called "flat language": there was no concept of modularization (although this was considered at the time), let alone real-time aspects and so forth. Nevertheless the definition of the language was set in stone at that point.

Much of the language evolution was the result of joint European research projects. For example, the Afrodite ESPRIT project, which was undertaken in the early nineties, proposed extensions for object-orientation [8]. The VICE project (VDM In Constrained Environments) introduced a real-time extension [9] and this strand of work continued in the context of the BODERC project [5]. Asynchronous operations and explicit computation and communication architectures were introduced which enable reasoning about

deployment and performance in a distributed setting. This dialect is now commonly referred to as VDM-RT and this work has matured in the DESTECS project [2], where focus was on developing tool support based on a unified structured operational semantics for co-simulation of VDM-RT (discrete event) models with continuous time models specified using bond graphs [4,3].

In 2004, the language was "adopted" by the Overture project, and in the years following this meant that the language was more or less defined by the tools that supported it. In 2009 the Language Board (LB) was introduced consisting of a selection of members from the Overture Community, with the specific task of coordinating changes to the language, and/or clarifying any issues with the language. In 2010, it was adapted to use the term "VDM-10" for the family of languages based on the original VDM-SL.

In its five years of existence the LB has taken a fairly reactive approach in dealing with language changes, meaning that it has not initiated any language changes by itself, but has awaited proposals to be put forward by the community at large. This is due to the fact that the Overture project is an open source project run by volunteers. More fundamental tasks, such as a solid (re)definition of the semantics of the object oriented nature of the language require a major research effort, something which cannot reasonably be expected from the LB considering the current resources.

2 Language Board process

At its start, the LB proposed a community process to deal with "Request for Modifications" (RM) for the language. In principle, an RM may be put forward by anyone (this person is referred to as the "originator"), even individuals outside the Overture Community. The RM has to describe aspects such as motivation for the request (rationale), a precise description, if possible including semantics, and so on. As a first step an "owner" is assigned to an RM. This is an LB member and therefore not necessarily the same person as the one who put the RM forward. The owner is responsible for moving the RM forward through the process of any possible language changes to support the RM. A special issue tracker in the SourceForge environment [1] is used to coordinate and manage all RMs and this environment also provides an audit trail for each RM. Then the following phases and milestones are acknowledged:

1. **Initial consideration:** The RM is evaluated by the LB. The LB may then request expert opinions from named members, subject to the agreement of the originator. After this the RM may be (1) "rejected" (2) "approved unmodified", or (3) "approved subject to revision". Next steps for (1) and (3) are obvious, in case (2) the next phase is entered, the Discussion phase.
2. **Discussion:** During the discussion phase, the wider Overture Community is given the opportunity to participate in discussing the RM. The discussion takes places via e-mail, but also at the monthly Overture meetings on Skype. The results of the discussion are taken back to the LB and is input for the next phase: Deliberation.
3. **Deliberation:** The LB considers the outcome of the discussion by the overall Overture Community in the Deliberation phase. As a result, the RM may be rejected, and

[1] `http://sourceforge.net/projects/overture`

in that case the process terminates. It can also be the case that the RM is accepted without modification. In that case the next phase, the Execution phase, is entered. Finally it may be the case that the RM is accepted but modifications are required. In that case the RM is returned to the originator asking him to revise it.

4. Execution: Once a language change has been accepted, it must be reflected in the VDM-10 Language Reference Manual (LRM). This is the responsibility of the owner of the RM. The release manager of the Overture toolset is responsible for planning and organizing an update of the toolset reflecting the language change that has been proposed. This activity is coordinated with the LRM update. Once the LRM has been updated and the RM has been implemented in the tools, the Execution phase (and the process) ends. It may be the case that the RM cannot be implemented in the tools. In that case the RM is returned to the LB to make a decision on how to proceed (so far this has never happened).

3 Examples of language change

Over the years several RMs have been dealt with by the LB. In the following subsections we discuss the highlights for only a few of those that are considered of most importance to the language.

3.1 RM #23, Map Patterns

This RM was received in November 2011. VDM includes patterns for matching various aggregate types, like sets, sequences and tuples, but the language did not include support for map patterns. This RM was a proposal to add support for map patterns, following some initial work in VDMTools. [2]. The example below demonstrates the use of the map pattern and returns the value 3:

```
let {1 |-> a, a |-> b} = {1 |-> 2, 2 |-> 3} in b
```

The changes to the grammar are as follows:

```
pattern = ... map enumeration pattern
        | map munion pattern
        | ... ;

map enumeration pattern = '{', maplet pattern list, '}'
                        | '{', '|->', '}' ;

maplet pattern list = maplet pattern, { ',', maplet pattern } ;

maplet pattern = pattern, '|->', pattern ;

map munion pattern = pattern, 'munion', pattern ;
```

[2] http://www.vdmtools.jp/en/

There was some initial discussion regarding the detail of how a match would work, and under what circumstances a match would fail. The proposal for the map union pattern was changed from using the ++ operator to the munion operator, as this was closer to the intuitive semantics, and the map enumeration pattern was modified to include the empty map as a legal value. The result is a natural extension to the language that significantly simplifies the process of working with map data.

3.2 RM #12, Non-deterministic statements in traces

In June 2010, an RM was submitted to the LB requesting that the language be extended to include a non-deterministic trace statement. The purpose of this is to allow traces to explore the effect of the unpredictable ordering of operation calls in a concurrent environment. The new trace statement is similar to the normal non-deterministic statement, which calls the list of statements defined in a non-deterministic order [3]. Within a trace definition, the new `trace concurrent expression` now expands to *every possible order* of the trace statements contained within it.

trace concurrent expression = ‘||’, ‘(’, trace definition,
‘,’, trace definition,
{ ‘,’, trace definition }, ‘)’ ;

It was agreed that the syntax should be like the normal non-deterministic statement. As an example, the two trace statements below are equivalent:

```
|| (Op1(); Op2(); Op3())

(Op1(); Op2(); Op3()) |
(Op1(); Op3(); Op2()) |
(Op2(); Op1(); Op3()) |
(Op2(); Op3(); Op1()) |
(Op3(); Op2(); Op1()) |
(Op3(); Op1(); Op2())
```

3.3 RM #2, The introduction of the “reverse” sequence operator

In May 2009, an RM was received which requested a change to the sequence reverse processing defined in the language. Before the change, the `reverse` keyword was available only as part of `for` loops that iterate over sequences, causing them to iterate in reverse. The RM proposed to introduce a new unary `reverse` operator, taking a sequence argument and returning the sequence in reverse order. At the same time, the change also proposed removing the `reverse` keyword from the `for` loop, since this would be redundant. The affected syntax is as follows:

[3] The interpreter actually executes statements in a consistent under-determined order, rather than a non-deterministic order.

sequence for loop = ‘**for**’, pattern bind, ‘**in**’, expression,
‘**do**’, statement ;

unary operator = ‘+’ | ‘−’ | ‘**abs**’ | ‘**floor**’ | ‘**not**’ | ‘**reverse**’
| ‘**card**’ | ‘**power**’ | ‘**dunion**’ | ‘**dinter**’
| ‘**hd**’ | ‘**tl**’ | ‘**len**’ | ‘**elems**’ | ‘**inds**’ | ‘**conc**’
| ‘**dom**’ | ‘**rng**’ | ‘**merge**’ ;

The new unary `reverse` operator is the same precedence and grouping as other unary sequence operators – i.e. precedence 6 in the Evaluator family, with left grouping.

The change is a generalization of the existing semantics, with the advantage that legal specifications under the old grammar are still legal in the new grammar. There is one subtle change in the semantics which could occur, if a specification uses a combination of sequence operators in a `for reverse` loop, for example `for elem in reverse S1^S2`. This used to mean the reverse of the entire concatenated list, but under the new grammar this means the concatenation of (reverse S1) and S2. It was considered sufficient that this case should cause a warning to be generated in the tools, “Warning 5013: New reverse syntax affects this statement”.

3.4 RM #3, Generalising let-be expressions

This RM was received in May 2009. Before the change, the `let be st` expression and statement used a simple `pattern bind`, as did the equivalent `trace let be st binding`. The change proposed to extend this to include a `multiple bind`, allowing multiple separate patterns to bind to the same set or type. This avoids having to write nested let expressions, and it is particularly useful in traces, where it is common to want to bind over multiple variables. The new syntax is as follows:

let be expression = ‘**let**’, multiple bind, [‘**be**’, ‘**st**’, expression], ‘**in**’,
expression ;

The proposal was accepted with little discussion as it is a natural extension, and avoids the need to write nested expressions to achieve the same effect. The change is also backward compatible with existing specifications. The proof obligations generated for the binding are extended in a natural way.

3.5 RM #4, Functions should be implicitly static in VDM++

This RM was received in May 2009. Before the change, the existing implementations of VDM++ would allow functions to be declared as static or non-static, and it was also possible for functions to call operations. A function cannot access instance variables, and therefore does not need a `self` reference – in effect functions are always static. Similarly, a function should produce no side effects in the state, and so it should not be possible to call an operation from a function. This proposal created considerable debate, much of which overlapped with the issues raised in RMs #13, #14 and #15 regarding the object oriented semantics of VDM++.

At the LB meeting on the 20th September 2009, the following points were agreed: (1) every function is implicitly static; (2) reference to `self` is not allowed in a function; (3) `obj.fn()` binding is determined by actual type of `obj` (polymorphism); (4) `fn()` binding is determined by the enclosing class and super classes statically; (5) `C`fn()` is still possible, as is `obj.B`fn()`, to select a function in a hierarchy; (6) we disallow all operation calls in a function definition.

Points 1, 2 and 6 were new at the time, while rest were current functionality. For existing specifications, occurrences of 1, 2 and 6 should generate deprecated warnings. Going forward, violations of these will become type checking errors. Issues remain regarding the object-oriented semantics of VDM++, but enforcing the purity of functions is a step in the right direction.

4 Synchronization with tool development

Although it is the business of the LB to consider language changes, these would be of little value if the tools did not implement them. On the other hand, it is not the business of the LB to control the development and release schedule of the tools – this is a matter for the Overture Core group. Therefore there needs to be clear communication between the LB and the development group, partly to implement the changes when approved, but also to coordinate updates to the LRM (which is owned by the LB) with releases of the tools. In this way the language features described are actually supported at the time of release. Currently, this is coordinated through the Overture core group, who convene for an on-line meeting ten times per year. A release manager is appointed and currently a tool group is established to coordinate tool development at a higher level.

The LB follows the *Overture Community Process* [4]. At any given time, several RMs may be progressing through the process. Towards the end of the process, the Execution phase allows the LB to track the implementation of the change in the tools. This is done informally, between the RM owner and the development group. The RM owner is then responsible for updating the LRM to describe the changes implemented. When the LRM changes have been made, and the tool implementation is complete (or complete enough for beta testing in a release), the LB informs the Release Manager of the availability of the change and the corresponding updates to the LRM. If all goes well with beta testing, the new feature will then be made available in the next release of the tool.

5 Future and conclusions

In this paper we presented the approach that the Overture community has taken in dealing with requests for changes to the VDM-10 family of specification languages. This approach has been a structured one and very much aimed towards making decisions as a community. Central in the approach has been the installation of the "Overture Language Board" (LB), which is responsible for moving language change requests (Requests for modification: RM) forward and providing expert knowledge on the details of the language and advising on the RMs. The LB actively consults the Overture community as

[4] `http://wiki.overturetool.org/index.php/Community_Process`

part of its decision making. Since the beginning of the existence of the LB it has received a total of 23 RMs, 8 of which are in progress. The paper has described some of them in more detail.

The attitude the LB has taken so far been rather reactive, i.e. waiting for RMs to come in and then deal with them according to the established process. Recently, the language board proposed to extend their scope also to the standard libraries that the Overture tool currently supports. Main motivation for this widened scope is that in the context of language evolution the consistency of (legacy) models do not only rely on the definition of the language itself, but also on the definition and behavior of any support libraries. More fundamental changes, or at least enhanced language definitions need to be made as well. For example, the formal semantics of the object-oriented version of the language needs to be defined, and this at the very least requires further research. This is a certainly an area that the LB wishes to coordinate, however significant further resources are needed to do this which are not currently available to the LB or the Overture community.

Acknowledgments. The authors would like to thank all people who have contributed to the evolution of the VDM languages, e.g. by making requests for language modifications (RMs), by discussing RMs, and by implementing RMs.

References

1. Bjørner, D., Jones, C. (eds.): The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science, vol. 61. Springer-Verlag (1978)
2. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., F., W.: 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)
3. Coleman, J.W., Lausdahl, K., Larsen, P.G.: Semantics for generic co-simulation of heterogenous models (April 2013), Submitted for publication to the Formal Aspects of Computing journal
4. Coleman, J.W., Lausdahl, K.G., Larsen, P.G.: D3.4b — Co-simulation Semantics. Tech. rep., The DESTECS Project (CNECT-ICT-248134) (December 2012)
5. Heemels, M., Muller, G.: Boderc: Model-Based Design of High-tech Systems. Embedded Systems Institute, Den Dolech 2, Eindhoven, The Netherlands, second edn. (March 2007)
6. ISO: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996)
7. Jones, C.B.: Systematic Software Development Using VDM. Prentice-Hall International, Englewood Cliffs, New Jersey, second edn. (1990), iSBN 0-13-880733-7
8. Lano, K., Haughton, H. (eds.): Object-oriented Specification Case Studies, chap. 6: Object-oriented Specification in VDM++. Object-oriented Series, Prentice-Hall International (1993)
9. Mukherjee, P., Bousquet, F., Delabre, J., Paynter, S., Larsen, P.G.: Exploring Timing Properties Using VDM++ on an Industrial Application. In: Bicarregui, J., Fitzgerald, J. (eds.) Proceedings of the Second VDM Workshop (September 2000), Available at www.vdmportal.org
10. Plat, N., Larsen, P.G.: An Overview of the ISO/VDM-SL Standard. Sigplan Notices 27(8), 76–82 (August 1992)

Appendix: Request for Modifications – Overview and Status

ID	Summary	Milestone	Status	Tool support
1	International character support for Overture identifiers.	Completed	Closed	yes
2	The introduction of the“reverse” sequence operator.	Completed	Closed	yes
3	Generalising let-be expressions.	Completed	Closed	yes
4	Functions should be implicitly static.	Completed	Closed	yes
5	Scope rules for assignments.	Completed	Closed	yes
6	Inheritance of constructors.	Withdrawn	Closed	no
7	Adding explicit object reference expressions to VDM++.	Withdrawn	Closed	no
8	Need definition of VDM++ operation pre/post functions.	Withdrawn	Closed	no
9	VDM++ object oriented issues.	Rejected	Closed	no
10	Invariant functions for record types	Rejected	Closed	no
11	Exception handling in interpreter.	Rejected	Closed	no
12	Include the non-deterministic statement inside traces.	Completed	Closed	yes (Overture only).
13	Static Initialization.	Withdrawn	Closed	no
14	Object Construction.	Withdrawn	Closed	no
15	Inheritance, Overloading, Overriding and Binding.	Withdrawn	Closed	no
16	Expressions in periodic thread definitions.	Completed	Closed	yes (Overture only).
17	Values in duration / cycles statements.	Completed	Closed	yes (Overture only).
18	Sporadic thread definitions.	Execution	Open	n.a.
19	Extend duration and cycles (allow intervals + probabilities).	Discussion	Open	no
20	Antagonist STOP operation for periodic threads is missing.	Execution	Open	n.a.
21	More descriptive time expressions.	Rejected	Closed	n.a.
22	Append narrow expression.	Completed	Closed	yes
23	Append map pattern.	Completed	Closed	yes
24	Additional print statements in IO library in VDM-RT.	Rejected	Closed	no

Table 1. Overview and status of all community Requests for Modifications

An Architectural Evolution of the Overture Tool

Peter W. V. Jørgensen, Kenneth Lausdahl, Peter Gorm Larsen

Department of Engineering, Aarhus University, Finlandsgade 22, 8200 Aarhus N, Denmark

Abstract. The Overture project governs an open source platform providing tools for formal modeling, and thus the success of the project depends on contributions made by members of the community. Source code contributions are often provided as plugins that rely on core elements of the platform such as the VDM Abstract Syntax Tree (AST), the type checker and the interpreter. To support the efficient development of plugins, the most recent changes made to the platform modify the structure of the AST to promote the extendability of the platform. The intent is to make core functionality more easily accessible for outside developers and create an attractive platform to build on. This paper covers some of the important changes recently made to the AST. Using real examples, demonstrations will be given for how these changes can be exploited in order to benefit from and extend the existing platform.

Keywords: Overture tool, VDM, abstract syntax tree, tool development, software architecture, plugin development, Eclipse IDE

1 Introduction

The development of the Overture tool started back in 2003 and was primarily carried out by Master's students [13]. At that time the tool was Eclipse [4] based with support for partial checking of the syntax and its static semantics. The syntax tree was stored in XML and did not preserve all the information of the parse tree. This design choice was made to avoid the inefficiency of storing the potentially large tree and traversing the entire structure. Later it was changed to an AST isomorphic to the concrete syntax, composed of hand written nodes [11]. This design was, however, prone to errors due to the manual work of maintaining and extending the tree. As a response to this, Verhoef developed a tool enabling the generation of AST nodes in both Java and VDM. The generated Java nodes were used in Eclipse for developing the tool, while the VDM nodes supported the development of tool extensions using the VDM related validation techniques such as invariant checks. These VDM models would then be transformed into Java code as enabled by VDMTools [3, 6] and then adopted by the Overture platform or a new code generator can be added to Overture directly [8].

While these efforts were ongoing the command-line based VDM interpreter, VDMJ, was being developed [1] and later integrated with the Overture tool in order to establish a common front-end [10]. This resulted in two different internal representations of the AST inside Overture, which is costly to maintain and would have complicated reuse across platform components. Although it was possible to convert the generated AST

into a structure compatible with VDMJ, this architecture has several drawbacks, which opposes the goal of having an extendable platform. For that reason, and to solve these design conflicts, a redesign of the AST architecture was needed.

This paper is structured such that the introduction is followed by Section 2, which provides an overview of the VDMJ based AST architecture used in version one releases of the Overture tool. Then Section 3 covers the new AST architecture used in version two releases, motivated by the above mentioned design conflicts. Afterwards Section 4 demonstrates the new architecture by example using recent development projects. Finally, Section 5 provides suggestions for future platform extensions with the aim of making it a more attractive and sound foundation for developers to build on.

2 The VDMJ based AST architecture

Initially VDMJ was developed independently of the Overture tool. Therefore it has not been a primary goal to extend or integrate VDMJ with the Overture tool in the first place. This is reflected in the implementation, which is characterized by 1) performance being a key quality architectural design driver and 2) a close coupling between core components (e.g. the type checker and the interpreter). Design choices motivated by performance often require architectural decisions to be made, which impact extendability in a negative manner. Thus it is appropriate to say that the performance of VDMJ comes at the cost of extendability.

The AST nodes of VDMJ are hand written and functionality like type checking and interpretation is integrated directly into the tree structure. For example, nodes that can be type checked and evaluated must implement the `typeCheck` and `eval` methods, and the same applies to other functionality such as generation of proof obligations. Following this design, tool extensions are likely to require direct modifications to the AST nodes, which may affect other components using the tree. Therefore this design opposes the goal of the Overture tool being a platform for developers to build on.

The feedback from Overture development has motivated a new design that aims for an extendable AST with the possibility of adding custom nodes, and where functionality can be added without affecting other components using the tree. In addition, modifications or extensions to the AST should require minimum effort to avoid tedious and erroneous work. This is desirable for large grammars like that of VDM, as it results in many AST nodes being produced.

3 The new AST architecture

In order to deal with the design conflicts of the VDMJ based AST architecture, the new design introduces three major changes. First, it moves all non-trivial functionality (e.g. type checking and evaluation) out of the nodes. Secondly, nodes are being generated using a tool that takes as input an AST description and produces Java based AST nodes. This tool is inspired by the SableCC [12] parser generator and produces fixed nodes in the sense that they should only be changed via the AST description and re-generated based on that. Finally, the AST structure uses bidirectional node relations so information requiring navigating up or down the tree can be obtained. This is intended

to support the implementation of commonly used Eclipse features such as refactoring, auto-completion and going to a definition. For example, it is possible to find the type definition of a given type in the following manner:

```
type.getAncestor(ATypeDefinition.class)
```

Functionality such as type checking and evaluation resides in visitor classes, which specify the appropriate actions to be taken when nodes of different type are being visited. Thus the type check of each node is specified in a method of appropriate name in the type checker visitor class. As an example, the type check of a "greater than" expression (>) requires that the left and right hand sides are of numeric type. The interpreter is developed using the exact same approach as that of the type checker, and visitors can be implemented by all platform extensions that need access to the AST.

3.1 How to extend the AST

The output of the AST generator reduces the effort needed to experiment with the language. For example, in order to add a new expression to the AST one would first have to update the AST description and process it using the generator as shown in Figure 1. Next the parser, type checker and interpreter must be extended to handle the new expression. For type checking and evaluation this means implementing the appropriate methods in the visitor subclasses.

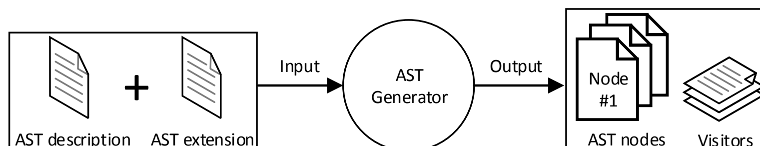


Fig. 1. The AST generator takes an AST description and outputs the AST and visitor base classes

The AST description supports inheritance, and thus a newly declared (say) numeric expression becomes a child of that node (in the object-oriented sense). To demonstrate this, the AST description snippet below shows the declaration of the two numeric binary expressions "greater than" and "less than". From this the AST generator will produce Java nodes for each of the two expression declarations.

```
#Numeric {-> package='org.overture.ast.expressions'}
        =   {greater} //e.g. '3 > 2'
        ... // Other numeric expressions omitted
        |   {less}  //e.g. '2 < 3'
```

3.2 AST analysis using visitor classes

Like for AST nodes, visitor base classes are generated from the AST description. The most general visitor is designed according to a question-answer principle and serves as a template for various kinds of analyses. It is implemented as a parametrized class

(or generic in Java terms) and takes as input two types representing a question and an answer. The question is passed along to the nodes as they are visited and holds information that is needed in order to answer the question. The answer, on the other hand, specifies the information to be contained in a reply each time nodes are being visited.

The type checker is a visitor structured according to this principle with the parametrized visitor `QuestionAnswerAdaptor` as base class. For this example, the question holds the information needed to do the type checking, whereas the answer requires each node visited to return the resolved type. This leads to the type checker implementation shown in the code snippet below. Note that it includes the type check of the "greater than" expression mentioned in Section 3.1.

```
public class TypeCheckerExpVisitor
extends QuestionAnswerAdaptor<TypeCheckInfo, PType> {
   ... // Fields and visitor cases omitted
   @Override
   public PType caseAGreaterNumericBinaryExp(
      AGreaterNumericBinaryExp node,
      TypeCheckInfo question)
   throws AnalysisException {
      ... // Type check omitted
   }
}
```

The visitor design supports termination of an analysis even if some nodes are left to be visited. This may be appropriate if the visitor has found what it was looking for, e.g. an operation of a certain signature used as entry point in a VDM model. In that case the most convenient way of terminating the visitor is simply to throw an `AnalysisException` to avoid further tree traversal. In addition, the new AST architecture introduces the notion of an assistant which provides a placeholder for node functionality that does not belong to the visitor itself. As an example, the visibility of a node representing an access specifier can be found using the associated type checker assistant.

4 Applications of the visitor based architecture

The new AST architecture introduced above is already adopted by the Overture platform and used in multiple projects. This means that practical experience has given feedback and influenced the design. However, some projects and plugins developed for the platform make heavy use of the new architectural constructs and deserve to be mentioned.

The COMPASS project: In the ongoing EU-FP-7-Frame Programme Project COMPASS research is made that extends the VDM language [5, 2]. COMPASS builds tools for formal modeling based on the Overture/Eclipse platform to support the COMPASS modeling language (CML), which includes subsets of VDM-SL and VDM++. The COMPASS tool developers at Aarhus University are active contributors to the Overture platform, and thus the project provides continuous feedback for the architecture of

the Overture platform. This experience has given rise to a number of suggestions for architectural changes to promote the extendability of the Overture platform, some of which will be addressed in Section 5.

VDM-UML mapping: Available in the Overture tool is a plugin [9] that translates between VDM-RT and the Unified Modeling Language (UML) [7]. Aside from accessing the parse tree through the model representation, the plugin performs different kinds of AST analyses in order to enable translation from VDM to UML. This plugin is interesting with respect to the new AST architecture as it makes heavy use of the functionality in the assistant classes to analyse the AST nodes. For example, in UML classes representing threads and processes are "active" and drawn differently from those that are passive. Therefore, the translation checks whether the class contains a thread definition.

5 Future plans

The new AST architecture brings the Overture platform closer to its goal of being an attractive platform to use for developing tools for formal modeling. The changes from the VDMJ based design to the new visitor based architecture have made it easier to benefit from the existing functionality of the platform. This is enabled by the automatic generation of the AST from a description and the visitor based design that moves all non-trivial functionality out of the nodes and places it into visitors and assistants. However, this change in architectural design is only the first step towards the goal and more things remain to be done. In this section some suggestions for future improvements to the design are discussed.

Integrated Development Environment (IDE) modeling support: Modern development environments offer auto-completion of e.g. identifiers and operations as these are being typed, and refactoring for making behavior-preserving model transformations that reduce manual intervention. The new AST design is the first step towards implementing such features, which require the possibility to search and manipulate the tree structure. By providing a convenient way of accessing the AST, the intent is to alleviate the effort needed for developing new tool features.

Generation of Overture components: Generally speaking, writing parsers is a time consuming task, prone to errors, and it only gets more difficult the larger a language is. In the visitor based design currently adopted, the parser almost remains the same with respect to the VDMJ based architecture, i.e. it is written manually. VDM is a large language, and reducing the effort needed for maintaining and extending on the parser functionality would be a good place to improve on the platform architecture in upcoming releases of the Overture tool. This would potentially lead to a shorter development cycle with respect to extensions and maintenance of core functionality.

Constructing a core interpreter: The current interpreter (including the parser and type checker) is structured in a way so it handles all three dialects of the VDM language: Prior to invoking the interpreter the VDM dialect is set and then taken into account

during evaluation. Perhaps a better design would be to identify a core set of language elements that are evaluated in the same way for both the VDM-SL and VDM++ interpreters. This core could then have its own abstract interpreter base class extended by the VDM-SL and VDM++ interpreters, the latter being a superclass of the VDM-RT interpreter. This would allow the VDM-SL and VDM++ interpreters to share the core VDM language subset without depending on each other. This design is illustrated in Figure 2. The challenge is to design this structure so that subtle differences across dialects do not cause similar code to be maintained in the interpreters. For example, the code for instantiating a class during evaluation is almost the same for VDM++ and VDM-RT, except in the latter dialect an object can be deployed to a CPU.

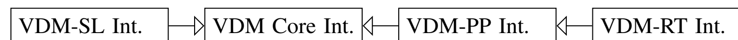


Fig. 2. Illustration of the architectural interpreter design based on a common language core

Acknowledgments. The authors would like to thank the reviewers for their valuable feedback on the work presented in this paper. Special thanks to Nick Battle, Augusto Ribeiro, Joey Coleman and Marcel Verhoef for their vital contributions to the development of the Overture platform.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Coleman, J.W., Malmos, A.K., Nielsen, C.B., Larsen, P.G.: Evolution of the Overture Tool Platform. In: Proceedings of the 10th Overture Workshop 2012. School of Computing Science, Newcastle University (2012)
3. CSK: VDMTools homepage. *http://www.vdmtools.jp/en/* (2007)
4. Eclipse website (2013), http://www.eclipse.org/
5. The COMPASS project website (2013), http://www.compass-research.com/
6. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. ACM Sigplan Notices 43(2), 3–11 (February 2008)
7. Fowler, M., Scott, K.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley (2003)
8. Jørgensen, P.W., Larsen, P.G.: Towards an Overture Code Generator. In: Submitted to the Overture 2013 workshop (August 2013)
9. Lausdahl, K., Lintrup, H.K.A., Larsen, P.G.: Connecting UML and VDM++ with Open Tool Support. In: Cavalcanti, A., Dams, D.R. (eds.) Proceedings of the 2nd World Congress on Formal Methods. Lecture Notes in Computer Science, vol. 5850, pp. 563–578. Springer-Verlag, Berlin, Heidelberg (November 2009), http://dx.doi.org/10.1007/978-3-642-05089-3_36, ISBN 978-3-642-05088-6
10. Møller, D.H., Thillermann, C.R.P.: Using Eclipse for Exploring an Integration Architecture for VDM. Master's thesis, Aarhus University/Engineering College of Aarhus (June 2009)
11. Nielsen, J.P., Hansen, J.K.: Development of an Overture/VDM++ Tool Set for Eclipse. Master's thesis, Technical University of Denmark, Informatics and Mathematical Modelling (August 2005), iMM-THESIS-2005-58
12. SableCC website (2013), http://www.sablecc.org/
13. van der Spek, P.: The Overture Project: Designing an Open Source Tool Set. Master's thesis, Delf University of Technology (August 2004)

Towards an Overture Code Generator

Peter W. V. Jørgensen and Peter Gorm Larsen

Department of Engineering, Aarhus University, Finlandsgade 22, 8200 Aarhus N, Denmark

Abstract. When one spends time on producing a formal model using a notation such as VDM, the insight one gains should make it worth the time spent on producing this model. One possible way to improve the value of the model is to use it for automatic generation of the implementation in a programming language. In this paper we describe work in progress targeting such a code generation feature for the Overture platform.

Keywords: VDM, Java, code generation

1 Introduction

The intent of spending a significant amount of time on producing, validating and verifying a high quality formal model is to gain an improved understanding of the system under construction and its solution. Having invested that time it is desirable to automatically generate the implementation from this model in order to reduce the effort needed for transitioning to the implementation phase. Automation tools such as code generators may be helpful for this task.

In this paper we demonstrate our initial attempt to produce a VDM to Java code generator for the Overture platform inspired by the earlier work of VDMTools [4]. The intent of this paper is to enable the reader to get a first impression of the general principles used in this new attempt to produce code generation support for VDM models.

When considering the use of code generators for production code one also needs to be aware of what this means with respect to the possible abstraction levels that are appropriate to apply to a formal model. For example, a type checker for a computer language could be written as a model that simply yields true or false depending upon the static correctness of the input it takes. However, from a practical perspective the model will be of low value since the user of the computer language would like error messages indicating where problems occur so they can be fixed. Thus, code generation can be a cost-effective approach but one needs to be aware of the consequences of applying the different abstraction mechanisms.

This paper starts with an overview of related contributions that have inspired our work in Section 2. Afterwards Section 3 describes the architecture of the code generator. This is followed by a small case study we use for generating code in Section 4. Finally, Section 5 ends the paper with an indication of the future work planned so far.

2 Related work

For VDMTools a code generation feature was produced already in the nineties for both C++ and Java [2]. Towards the end of the nineties support was also produced for the

concurrency parts of VDM++ [6]. However, the target for most of this work has been prototype code generation, rather than targeting final production code.

Much later a first attempt to produce a code generator inside the Overture project was made [5]. However, with this solution it was not possible to extract type information from the Abstract Syntax Tree (AST). As a consequence the code generator produced at that time never got to a stage where it was working for anything but trivial examples.

3 Code generator architecture

All version one releases of the Overture tool rely on the performance efficient architecture of the VDMJ interpreter [1]. However, from version two releases onwards this has changed into a *visitor* based architecture, which intends to provide a more convenient way of extending the Overture platform through simple access to the information in the (decorated) AST [3]. This change has been imposed to make it easier for community users to contribute with additional platform extensions.

The code generator for the Overture platform presented in this paper uses visitors for traversing the AST describing the VDM model the code generator takes as input. Several visitors deal with *VDM nodes* of certain types (expressions, definitions etc.). Together these visitors construct a new intermediate AST based on nodes reflecting concepts present in most Object-Oriented (OO) programming languages (construction of objects, class definitions etc.). Subsequently these trees are referred to as *OO ASTs* consisting of *OO nodes*.

The OO AST serves two primary purposes: First, it provides a way to gradually deal with the complexity of generating code from a VDM model as the OO AST does not include details specific to a programming language. Secondly, it intends to make code generation for multiple OO languages easier through configuration of the *backend* that generates the code based on the OO AST. The backend can be configured with *templates* and *library code* all specific to a programming language and which completely determine the code generated from the VDM model. The templates[1] consist of scripts describing how OO nodes are mapped into a programming language based on the information stored in the OO AST. In addition, the backend makes use of library code that implement equivalent VDM functionality that is not easily expressed in a programming language (e.g. Java code handling concatenation of sequences). A complete overview of the code generator architecture is shown in Fig. 1.

4 Case study

To illustrate the current state of the code generator this section generates code from a small VDM model representing a system for handling company salary expenses. We use this particular model as it includes many of the VDM constructs currently supported by the code generator. The VDM model is shown in the UML class diagram in Fig. 2 which only shows public operations to keep the diagram simple.

[1] The templates are targeting the Apache Velocity template engine: http://velocity.apache.org/

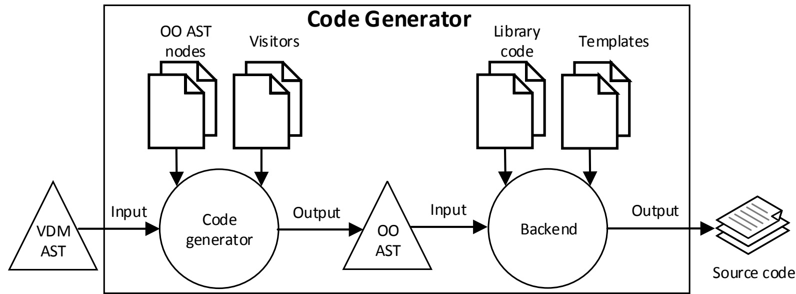


Fig. 1. An architectural overview of the code generator

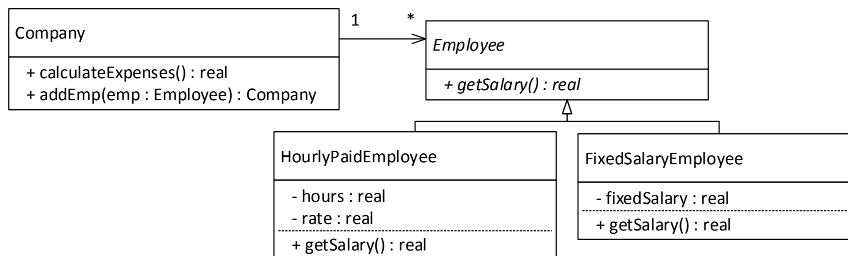


Fig. 2. The Company Salary Expenses System illustrated using a class diagram

In Fig. 2 the abstract `Employee` class is parent of `FixedSalaryEmployee` and `HourlyPaidEmployee`, which both provide the appropriate implementations of the `getSalary` operation declared in `Employee`. The code generator generates the inheritance hierarchy from the VDM model one would expect, since inheritance exists both in VDM and Java. However, one should be aware that the current version of the code generator does not allow code to be generated from a VDM model that uses multiple inheritance. Instead the modeller must refine the model to avoid use of such VDM constructs before applying the code generator to it.

A `Company` is associated with a number of employees that together constitute the company salary expenses. The `calculateExpenses` operation in the `Company` class performs this calculation by iterating through the collection of company employees while summing the salaries. In the VDM model a sequence is used for associating employees with a company. The calculation of the company expenses is done in VDM by traversing this sequence recursively in the `start_calc` function as shown in Listing 1.1.

Listing 1.1. The VDM specfication of the `Company` class

```
class Company
instance variables
  private employees : seq of Employee;
operations
```

```
  -- Constructor omitted...
  public calculateExpenses: () ==> real
  calculateExpenses() ==
    return start_calc(employees);

  public addEmp : Employee ==> Company
  addEmp (emp) ==
  (
    employees := employees ^ [emp];
    return self;
  );
functions
  private start_calc: seq of Employee -> real
  start_calc(emps) ==
    if emps = [] then 0
    else (hd emps).getSalary() + start_calc(tl emps);
end Company
```

Due to space limitation we focus on the code generated from the `Company` class. For mapping of sequences we use library code based on standard functionality of subclasses realizing the `java.util.List` interface and Java generics, the last being similar to C++ templates. The code generated from the `Company` class is shown in listing 1.2 from which we see that the **`hd`** `emps` and **`tl`** `emps` expressions map into the Java calls `emps.get(0)` and `emps.subList(1, emps.size())`, respectively. In addition, we see that concatenation of sequences in Java is handled in the `Util` class using the `seqConc` method.

Listing 1.2. The Java code generated from the VDM `Company` class

```
import java.util.List;
public class Company {
  private List<Employee> employees;
  // Constructor omitted...
  public double calculateExpenses() {
    return start_calc(employees);
  }
  public Company addEmp(Employee emp) {
    employees = Utils.seqConc(employees, Utils.seq(emp));
    return this;
  }
  private double start_calc(List<Employee> emps) {
    if (emps.isEmpty()) {return 0;}
    else {
      return emps.get(0).getSalary()
              + start_calc(emps.subList(1, emps.size()));
    }
  }
}
```

5 Future plans

Although the code generator present in this paper is early work, it has raised several questions that would be interesting to address as part of the future work. We discuss some of these in the sub-sections below.

5.1 Code generation for a distributed hardware architecture

The initial intent of the code generator was to address the research challenge of generating code for a distributed hardware architecture. Modelling of a distributed system executing on CPUs communicating through buses is already supported by the VDM-RT extension. However, extensions to the Overture tool will be needed in order to allow the model to be annotated with additional information before generating code for a distributed hardware architecture. Such an annotation could be the specification of the communication protocol used between different CPUs (TCP/IP, UDP etc.).

5.2 Mapping of union types

Mapping of union types into a programming language that does not support this construct is considered one of the difficult challenges of generating code from a VDM model. One possibility is to include no support for union types, but this is impractical as they easily appear in a VDM model without the modeller being aware of this. Expressions similar to the two examples shown in Listing 1.3 will be likely to appear in a VDM model. Here the if expression and the sequence expression have types **`seq1 of char`** `|` **`nat1`** and **`seq1 of`** `(FixedSalaryEmployee | HourlyPaidEmployee)`, respectively.

Listing 1.3. Two examples of VDM expressions that involve union types

```
-- Has type seq1 of (char) | nat1
if true then "one" else 2
-- Has type seq1 of (FixedSalaryEmployee | HourlyPaidEmployee)
[new FixedSalaryEmployee(), new HourlyPaidEmployee()]
```

If the sequence expression in Listing 1.3 is generated to Java and assigned to a variable what should the type of that variable be? This mapping is not trivial because no construct similar to union types is supported by the language. One approach is to find a common denominator such as the `Object` class which acts as a parent of every class in Java. The advantage of this approach is its simplicity, but it is likely to lead to a lot of cast operations in the produced code since the type of an expression must be narrowed down before members can be accessed. This makes the code harder to read and maintain. Therefore, this future work item suggests investigating and comparing different approaches to mapping of union types into a OO language that does not support this construct.

5.3 Investigating the extensibility of the code generator

The architecture of the code generator intends to keep a clear separation between OO concepts and the actual programming language that the code generator generates code for. The initial work has been focusing on Java, but it would be interesting to use the OO AST with templates based on other programming languages. Since different OO programming languages have subtle differences in (for example) constructor semantics challenges are envisaged here. Ideally, extending the code generator with another programming language should be done by changing the templates that specify how the different constructs of the OO AST are mapped to the concrete programming language. However, it may be difficult to perform a mapping if the information needed for a particular programming language is not easily accessible from the OO AST, i.e. it requires extensive analysis of the tree. The OO AST could provide additional information that would make it easier for languages that require (say) declaration before use to easily get hold of the forward declarations needed for the generated code to compile.

Acknowledgments The authors would like to thank the reviewers for their valuable feedback on the work presented in this paper. Special thanks to Nick Battle and Kenneth Lausdahl for vital input and interesting discussions on the code generator architecture.

References

1. Battle, N.: VDMJ User Guide. Tech. rep., Fujitsu Services Ltd., UK (2009)
2. Group, T.V.T.: The VDM++ to Java Code Generator. Tech. rep., CSK Systems (January 2008), http://www.vdmtools.jp/en/
3. Jørgensen, P.W., Lausdahl, K., Larsen, P.G.: An Architectural Evolution of the Overture Tool. In: Submitted to the Overture 2013 workshop (August 2013)
4. 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
5. Maimaiti, M.: Towards Development of Overture/VDM++ to Java Code Generator. Master's thesis, Aarhus University, Department of Computer Science (May 2011)
6. Oppitz, O.: Concurrency Extensions for the VDM++ to Java Code Generator of the IFAD VDM++ Toolbox. Master's thesis, TU Graz, Austria (April 1999)

The COMPASS Proof Obligation Generator: A test case of Overture Extensibility

Luis Diogo Couto[1] and Richard Payne[2]

[1] Aarhus University
lcouto@iha.dk
[2] Newcastle University
richard.payne@ncl.ac.uk

Abstract. Proof obligation generation is used as a compliment to type checking for the verification of consistency of VDM specifications. The Overture toolset includes a Proof Obligation Generator (POG). Overture is designed to be a highly extensible platform. CML, a new language designed for modelling systems of systems is based in part on VDM. The CML tools are themselves built on Overture. We evaluate the extensibility and potential for reuse of Overture by reporting our experiences in developing a POG for CML as an extension of the Overture POG. During this process, we alter the existing Overture POG visitors in order to make them more extensible and reusable.

1 Introduction

Type checking is statically undecidable in VDM [1]. VDM specifications can be generally divided into 3 sets: on the one end we have correct or "good" specifications; on the other end we have incorrect or "bad" specifications; and between these two ends, we have undecidable specifications.

The VDM type checker can handle the first 2 sets on its own (it accepts correct specifications and rejects incorrect ones). Specifications from these 2 sets will not have any associated proof obligations. But for the third set, the undecidable specifications, we need the assistance of a Proof Obligation Generator (POG).

The POG therefore picks up where the type checker leaves off and generates a series of proof obligations related to the elements that make the specification undecidable. Discharging these obligations helps prove the internal consistency and correctness of the specification.

The Overture platform, an open source tool for VDM, has a POG for VDM as part of its toolset, although there is no support yet for discharging proof obligations [9].

The COMPASS project seeks to develop tools and practices for modelling Systems of Systems (SoS) [4], including the COMPASS Modelling Language (CML) and a supporting toolset built on top of Overture [3]. Part of the COMPASS toolset will include a POG for CML, developed as an extension of the Overture one.

In this paper, we consider the extensibility of the Overture POG and discuss the issues in the reuse of the Overture toolset. In Section 2, we provide a brief introduction to CML, Section 3 describes the CML POG, we discuss the extensibility of the Overture POG and issues for future development effort in Section 4. Counclusions are drawn in Section 5.

2 The COMPASS Modelling Language

The CML is the first language to be designed specifically for the modelling and analysis of SoS [10]. It is based on the languages VDM [6], CSP [7] and Circus [11]. A CML model comprises a collection of types, functions, channels and processes. Each process encapsulates a state and operations written in VDM and interacts with the environment via synchronous communications in CSP. A semantic model for CML using UTP [8] is in development as part of the COMPASS project [2].

As CML and the COMPASS tool platform are based upon VDM and Overture, the Abstract Syntax Tree (AST) generated by the COMPASS parser is extended from the Overture AST. The ASTCreator tool, a part of the Overture platform, is used to automatically generate ASTs for VDM dialects, which is extended to support CML. This reuse allows us to directly reuse elements of the Overture platform, including the type checker, interpreter and POG.

Being partly based upon VDM, the CML POG will generate those VDM Proof Obligation (PO)s generated by the Overture platform. As such, we aim to reuse and extend the Overture POG.

3 The COMPASS Proof Obligation Generator

3.1 Structure

The COMPASS POG is built on two sets of classes: visitors [5] and proof obligations. This structure was inherited from the existing Overture POG.

The ProofObligation class and its various subclasses are responsible for holding proof obligation data. Each different type of proof obligation has its own subclass (for example NonZeroObligation is a class for representing proof obligations that an expression must evaluate to something other than zero). There are also a related set of classes for storing data related to the proof obligation context. For example, the POFunctionContextDefintion stores the various syntactic elements of a function required for function-related proof obligations.

The other set of classes are the visitors. They are responsible for traversing the CML AST and generating the various proof obligations. Whereas the proof obligation classes can be thought of as holding the data, the visitor classes implement the behavior of the POG. Unlike the proof obligation classes, whose type hierarchy is dictated by the proof obligations we want to generate, the visitor hierarchy reflects the CML ast. We have 4 kinds of visitors, each responsible for a subset of AST nodes (POGProcessVisitor is responsible for traversing processes, etc.). At runtime we need an instance of each visitor type and we also need to move between them and so every visitor has a pointer to its parent visitor.

3.2 Behavior

The COMPASS POG is built as a series of visitors. The overall behaviour is relatively simple. The main visitor (ProofObligationGenerator) initializes the various

sub-visitors and applies them to the AST. Whenever one of the sub-visitors encounters a node it cannot handle (e.g. the process visitor encounters an expression) it will pass the node up to the main Visitor who will then re-apply the correct sub-visitor.

This behavior is shown in the SysML sequence diagram in Figure 1.

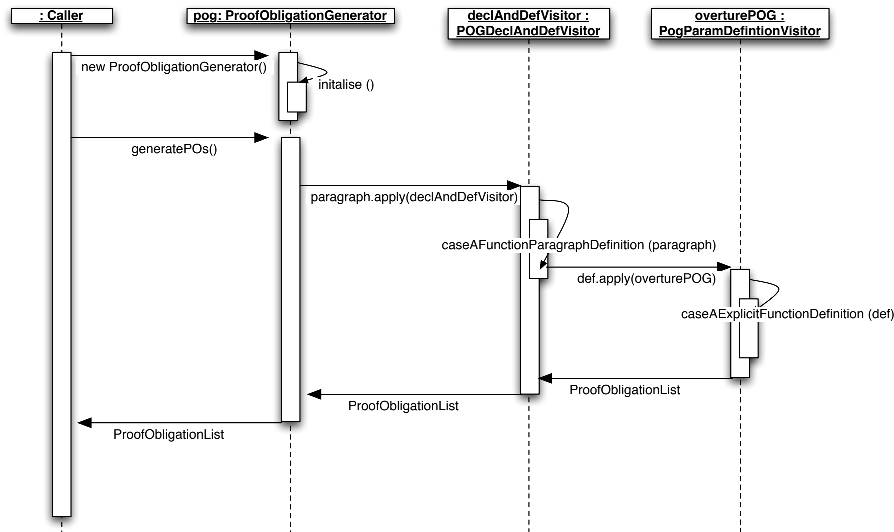


Fig. 1: Sequence diagram representing COMPASS POG visit

3.3 Reuse

Our main goal for reuse was to be able to directly utilise the Overture POG to generate all the Proof Obligations from VDM constructs directly. Because of this, the overall structure and behavior of the COMPASS POG are heavily influenced by the Overture POG. The entire visitor style of passing AST nodes between the various is lifted from Overture.

However, rather then simply passing a node up to their root, CML visitors must pass the node up to the Overture visitors. For example, the CML expression visitor must handle new CML expressions and then call the Overture expression visitor to handle the VDM expressions. There are two main issues with this approach.

The first issue is that there is no way to immediately identify a node as being from Overture or CML without using `instanceof` checks in a manually implemented decision method. One must use the default cases of visitors to work around this limitation. We can set up a for default case for CML nodes and another default case for all nodes (including the extended ones) . This of course limits our ability to handle default cases.

It would be good if we had 3 default cases available: extended, non-extended and all nodes. This limitation seems to be in the AST itself and not the Overture POG

The second issue we encountered was with the Overture POG visitors. When we pass a node to the Overture visitors, we are no longer able to control what happens. The AST goes under control of the Overture visitors and that is never relinquished. Their default cases are to call the root Overture visitor and its default case is to simply return `null`. The issue of course comes when you have both VDM and non-VDM nodes in a branch of the AST, which happens quite often. When the AST is passed to Overture, its visitors will not know how to handle the VDM nodes. Of course, this means that at best our POG will be unable to produce the proof obligations for these hybrid trees and at worst, it will die (this will be the most frequent outcome).

To handle this second issue, we had to alter the existing Overture POG to enable its visitors to release the AST back to COMPASS. We introduce the notion of a main visitor. The main visitor is the one that is called on most (any non-parent) calls of the `apply()` method. Previously these calls were of form `node.apply(this)`. Now they become `node.apply(mainVisitor)`. This main visitor becomes a parameter in the Overture visitors. To preserve compatibility with existing Overture plugins, we rename the altered visitors to ParamVisitor and create new subclasses of these parametrized visitors with the old visitor names. In these cases, the visitor receives a reference to itself as the main visitor parameter.

When the Overture visitors are used by COMPASS the COMPASS visitor is set as the main visitor parameter. This means that every `apply()` method will return the AST to COMPASS. Now, all decisions belong to COMPASS. The Overture visitor will simply unpack the node, generate any relevant proof obligations and apply the COMPASS visitor to any sub-nodes. In effect, the Overture visitor is called for the use of only one method at a time.

4 Discussion

The current version of the COMPASS POG generates the majority of VDM POs as generated in Overture. This is due to the reuse of the Overture Expression visitor, the ability to reuse the majority of the Overture declaration and definition visitors (apart from the Operation syntactic elements which differ in CML), and the reuse of ProofObligation and POContext classes. As mentioned above, this to reuse these elements required some effort. Whilst this reuse has been useful and reduced the amount of effort to generate VDM-related POs, there are two main dimensions in which the reuse is insufficient for a CML POG.

- We shall need to address the CSP syntactic elements of CML and the resultant POs not covered in VDM. The CML visitors currently have placeholders for most of the process and action CML language elements, influenced by the Overture visitors. Further language development effort is required to define the POs resulting from CML, not present in VDM.
- The current format of storing POs is adequate when their use is limited to printing to the screen. However, as the POs will be used by other analysis tools, storing POs

as strings is not appropriate. This is due to the fact that storing POs in this way allows only one form of PO representation, limiting the use the toolset can make from the generated POs. To address this issue, the PO representation format will be reimplemented in the form of its own AST, which will be an extended subset of the existing CML expression nodes. This new PO format will be composed of one PO expression (the assertion to be proved) and a set of PO expressions holding the context information. Work on this new format is underway, beginning with its implementation in Overture.

When tackling these issues, we should consider how much effort should be made in making changes in the Overture POG (which can be reused in the COMPASS tool platform) and how much is COMPASS-specific. Effort placed in the former case may slow down development of the COMPASS POG, however this will aid in future Overture reuse. However, we must be careful not to add complexity to Overture where it is not necessary for VDM. Our initial thought would be to make COMPASS-specific POG changes for the first issue above, and make changes in the Overture for the second issue.

The COMPASS toolset proposes the incorporation of several analysis tools as plugins to reason over properties of a CML model. The POG, therefore, is a clear source of such properties and thus the proof obligations generated must be made available to the analysis plugins and the analysis results must be related to the PO in the COMPASS toolset. Different plugins will need the proof obligation in different syntaxes and the new AST format will help with that. We can simply develop new visitors that traverse the PO AST and generate the relevant syntax. A clear example of this need for extensibility is the use of the proof assistant Isabelle[3]. To be of use, the proof obligations must be made available in Isabelle compatible syntax, refer to the relevant part of the CML model, and be associated with the result of any proof generated in Isabelle. The connection between proof obligations and their respective Isabelle proofs, particularly across multiple versions of a model is a problem currently under study.

5 Conclusion

We have presented a POG for CML, developed as an extension of the Overture POG. In developing, we have gained insight into the current extensibility and potential for reuse of Overture.

Overall, reuse is definitely possible and is quite powerful. However, it is not a particularly easy task. There were several issues with extending the Overture POG and were it not for existing familiarity with Overture, the task would have been extremely complicated.

We also benefited greatly from being able to alter existing Overture code. The visitor context swaps (particularly, return going from Overture back to COMPASS) were very challenging and without changes to the existing code, it would have been impossible to implement the COMPASS POG with proper reuse. It is clear to us that more work must be done to improve the extensibility of Overture.

[3] http://isabelle.in.tum.de

It is also worth mentioning that the development of these extended versions of Overture plugins can be quite challenging. It will be interesting to see how the combination of all Overture and COMPASS plugins turns out.

Acknowledgements

The authors wish to thank Peter Gorm Larsen and Joey Coleman for reviews on the manuscript. Nick Battle implemented the original Overture POG and is currently working on the AST version. His work is greatly appreciated. Simon Foster is developing the Isabelle plugin for COMPASS and his ideas on the format for proof obligations have been a great help.

The work presented here is supported by the EU Framework 7 Integrated Project "Comprehensive Modelling for Advanced Systems of Systems" (COMPASS, Grant Agreement 287829). For more information see http://www.compass-research.eu.

References

1. Hans Bruun, Flemming Damm, and Bo Stig Hansen. An Approach to the Static Semantics of VDM-SL. In *VDM '91: Formal Software Development Methods*, pages 220–253. VDM Europe, Springer-Verlag, October 1991.
2. Jeremy Bryans, Andy Galloway, and Jim Woodcock. CML definition 1. Technical report, COMPASS Deliverable, D23.2, September 2012.
3. Joey W. Coleman, Anders Kaels Malmos, Peter Gorm Larsen, Jan Peleska, Ralph Hains, Zoe Andrews, Richard Payne, Simon Foster, Alvaro Miyazawa, Cristiano Bertolini, and André Didier. COMPASS Tool Vision for a System of Systems Collaborative Development Environment. In *Proceedings of the 7th International Conference on System of System Engineering, IEEE SoSE 2012*, volume 6 of *IEEE Systems Journal*, pages 451–456, July 2012.
4. Comprehensive Modelling for Advanced Systems of Systems, 2011. http://www.compass-research.eu/.
5. R.Johnson E.Gamma, R.Helm and J.Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995.
6. John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.
7. Tony Hoare. *Communication Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey 07632, 1985.
8. Tony Hoare and Hi Jifeng. *Unifying Theories of Programming*. Prentice Hall, April 1998.
9. Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1), January 2010.
10. J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: a Formal Modelling Language for Systems of Systems. In *Proceedings of the 7th International Conference on System of System Engineering*, volume 6 of *IEEE Systems Journal*. IEEE, July 2012.
11. Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 184–203, London, UK, UK, 2002. Springer-Verlag.

Model Based Testing of VDM Models

Uwe Schulze*

Department of Mathematics and Computer Science,
University of Bremen, Bibliotheksstr. 1, 28334 Bremen, Germany
uschulze@informatik.uni-bremen.de

Abstract. This paper describes a framework to generate test procedures from a UML/SysML test model and execute these tests against an interpretation of a VDM design model. A generic algorithm for a test driver is defined as well as an architecture that allows powerful interface mappings. The generated test procedures can be used with different levels of abstraction of the design model and with different test integration levels. This approach allows test procedure definition and evaluation as well as requirements tracing in early development stages. With that, failures in the design model as well as in the test model or the generated test procedures can be detected early in the development.

1 Introduction

A common problem in software development is the fact that testing activities are scheduled after the design of a system has been developed. Even if tests are developed during the design phase, they cannot be evaluated without being able to execute these test procedures. This way, failures that are already in the design can only be found in a later stage of the development process. The increase in duration and cost of the development will be significantly higher than it would have been if the same failures would have been found earlier. The execution of the design model in a test environment providing inputs and recording outputs controlled by a test procedure is denoted as *Model-in-the-Loop Testing*. Model-in-the-loop testing would allow to execute tests against a design model so that these failures can be detected earlier. Also failures in the test procedure or problems in the test strength of a test suite could be detected.

1.1 Overture Remote Control Interface

Overture is an open source industry-strength tool that supports modeling and analysis in the design of computer-based systems. Models are expressed using the specification languages VDM-SL, VDM++ or VDM-RT which are based on the Vienna Development Method. Overtures interpreter component[4] allows the interpretation of an executable subset of VDM. The Remote Control Interface (*RCI*) is an extension of the interpreter that allows to control the execution of a

* The author's research is funded by the EU FP7 COMPASS project under grant agreement no.287829

VDM model from an external Java application. [1] This remote control functionality of Overture is used for the test execution described in section 4.

1.2 Model Based Testing with Rtt-Mbt

Model-based testing (MBT) is considered as leading-edge technology and state of practice in parts of industry. In this paper the term model-based testing is used in the following sense: [2]. The behaviour of the system under test (SUT) is specified by a model, in this case a UML or SysML model. Optionally, the SUT model can be paired with an environment model restricting the possible interactions of the environment with the SUT. A symbolic test case generator analyses the model and specifies symbolic test cases as logical formulas identifying model computations suitable for a certain test purpose. Constrained by the transition relations of SUT and environment model, a solver computes concrete model computations which are witnesses of the symbolic test cases. The inputs to the SUT obtained from these computations are used in the test execution to stimulate the SUT. The SUT behaviour observed during the test execution is compared against the expected SUT behaviour specified in the original model. The stimulation sequences are automatically transformed into test procedures executing the concrete test cases in a model-in-the-loop, software-in-the-loop, or hardware-in-the-loop configuration. A test model designed for model based testing with Rtt-Mbt always contains directed interfaces which are either of type TE2SUT (stimulations) or SUT2TE (observable SUT behaviour). A detailed introduction to model based testing with RT-Tester is given in [6][7].

2 Model-in-the-Loop Test Architecture

Model based testing with RT-Tester uses a UML or SysML test model to specify the expected behaviour of the system under test (SUT) or parts of the SUT that is to be tested. The design model also specifies the desired behaviour of the SUT, but with the intention of formal verification of the design, further refinement or to support the implementation e.g. through explicit interface specification or code generation. Both models can describe the SUT on different levels of abstraction but it must be possible to map the inputs and outputs of the SUT in the design model to inputs and outputs of the SUT in the test model and vice versa. The Rtt-Mbt test generator uses the test model (usually an XMI representation) to generate test procedures covering parts of the test model and reaching given test goals. Test procedures generated by Rtt-Mbt consist of a stimulator component and a test oracle evaluating the outputs of the SUT according to the given stimulations. In this approach we only need the stimulations and then perform replay to check SUT reactions against the test model for a given test execution. The stimulations are a timed traces of input vectors to the SUT. An interface

[1] For more information about the RCI interface see [1].
[2] adopted from [6]

module (IFM) is attached to the VDM design model to perform a mapping of SUT input signals from the test model to the design model and of SUT output signals from the design model to the test model[3]. During the test execution, each stimulation of the generated test procedure is mapped to a stimulation of the design model and each output of the SUT in the design model is mapped to signals in the test model. The result of a test execution is a timed trace of stimulations to the SUT and SUT outputs. The signals used in this log file are the ones from the test model.

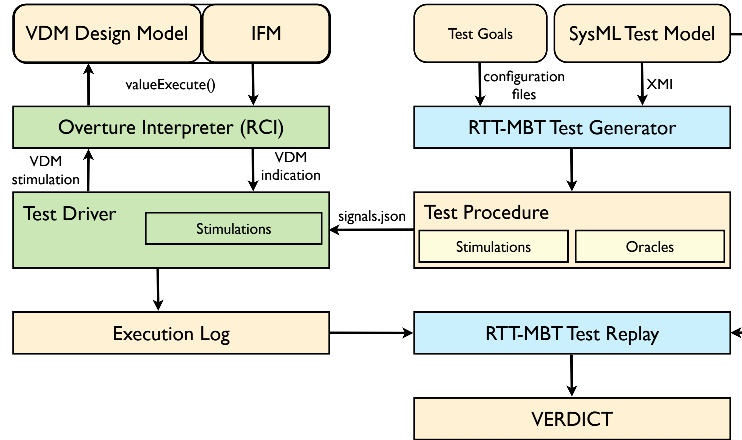


Fig. 1. Model-in-the-loop testing with Rtt-Mbt

The replay functionality of the Rtt-Mbt tool suite is used to calculate the test verdict of a test execution by checking the inputs and outputs of the test execution log against the expected behaviour specified in the test model. Figure 1 illustrates how the components of this architecture work together to generate and execute a test procedure against the design model.

3 Interface Mapping

Mapping the inputs and outputs of the SUT in the design model to SUT input (TE2SUT) and output (SUT2TE) interfaces of the test model is a vital part of the test configuration. The test procedures are generated automatically by Rtt-Mbt, but the mapping must be provided as part of the VDM test environment and may have to be adjusted each time the design model changes. Interfaces in a test model are treated as state variables during the test procedures generation. Although VDM models can provide functionality based on state variables only, it is more common to model the SUT behaviour using functions and operations. In this case, a SUT stimulation from the test model have to be mapped to a function call or operation call where it can also be the case that multiple test model stimulations take part as arguments in the same function or operation call. The

[3] see section 3 for details

approach described here uses a so called interface module (IFM) that performs the mapping from test model interfaces to the design model interfaces. The IFM will be defined as a VDM specification that is added to the design model only when testing the SUT. For each interface variable of the test model, a respective VDM instance variable is provided with the same name within the interface module so that the mapping from test model signals to design model signals of the IFM is the identity. The mapping from TE2SUT variable changes to concrete SUT stimulations is specified as an operation of the IFM. The mapping from changes in state variables or return values of function or operation calls to SUT2TE variables is also defined as an operation of the IFM. During a test execution, the interface module will be used to perform the actual VDM stimulation according to the test model interface changes and translate the relevant VDM design model changes into test model changes. The IFM can be adjusted to reuse a test model and the respective test suite to different test integration levels. This way, the same model and test suite can be used for model-in-the-loop tests of design models of different levels of abstraction as well as for software integration testing and hardware-in-the-loop tests.

4 Test Execution

A test is performed by a test driver that uses the remote control interface (RCI) of the Overture VDM interpreter to stimulate the SUT and to capture the SUT state and reaction. At the beginning of a test execution, an instance `interpreter` of class `RemoteInterpreter` is used to create the SUT and setup the interface module for the mapping. The interface module is referred to as `ifm` in algorithm 1.1 below. The test driver starts with the first test step at time stamp 0. It uses the trace of stimulations of a given test procedure to decide which stimulations (TE2SUT signals) are to be sent to the SUT in each test step (`getStimulations(timestamp)`). The stimulation file provides a sequence of stimulations that are either triggered by the current time stamp or as a reaction to SUT outputs (SUT2TE signals). For each stimulation, the current signal value is assigned to a variable with the same name in the interface module. The interface module provides variables for all TE2SUT and SUT2TE signals. After all signal values are assigned, the `ifm` is used to perform SUT stimulations for these signal changes (`ifm.performStimulation()`). The stimulations are a result of the mapping from the TE2SUT signals to design model stimulations defined in the `ifm`. The SUT reactions to the stimulations are mapped to SUT2TE signals by the interface module. All test-relevant changes in the state of the SUT are mapped to SUT outputs (SUT2TE signals) and are retrieved from the `ifm` by the test driver (`ifm.getChangedOutputs()`). All stimulations and SUT outputs are added to the test execution log using test model signal names and values (`logSutInputs(stimulations)`, `logSutOutputs(v)`) At the end of a test step, the time stamp for the next test step is calculated `getNextTimestamp()`. The output changes of the SUT are captured with a predefined cycle time. If the difference between the current time stamp and the time stamp of the next

Algorithm 1.1 A generic test driver

```
void testExecution() {
    long timestamp = 0;
    while (timestamp > 0) {

        // assign stimulations to TE2SUT signals in the IFM
        List<String> stimulations = getStimulations(timestamp);
        for (int idx = 0; idx < stimulations.size(); idx++) {
            // stimulations are of the form "setTE2SUTsignalName(value)"
            interpreter.execute("ifm." + stimulations[idx]);
        }
        logSutInputs(stimulations);

        // perform stimulation with new input signals
        interpreter.execute("ifm.performStimulation()");

        // retrieve and log changed states of the SUT
        Value v = interpreter.valueExecute("ifm.getChangedOutputs()");
        logSutOutputs(v);

        // calculate next timestamp (and sleep until then)
        timestamp = getNextTimestamp();
    }
}
```

stimulation is greater than this cycle time, the next test step will start after the cycle time exceeded [4]. Otherwise the next time stamp will be the one of the next stimulation. For stimulations that are triggered by SUT outputs, the respective SUT outputs are evaluated within the `getNextTimestamp()` function. If the conditions for a SUT triggered stimulation are satisfied, the next time stamp returned will be the current time stamp and the next test step starts immediately[5].

5 Example

In the following example, the stimulation of SUT inputs and capture of SUT outputs is explained for a single test step (one stimulation and the respective SUT indication). For the example, the SUT defines an interface operation `setSignalState`, that can be used to set the internal state of the SUT. Possible arguments for valid states are `{dark, stop, warning, drive}`. The

[4] In this case the next time stamp will be the current time stamp increased by the cycle time

[5] In this case, the time triggered stimulations for this time stamp have already been performed in the previous test step and are not performed again.

SUT uses three lamps `{<L1>, <L2>, <L3>}` to indicate its state. The operation `getActiveSignals` can be used to retrieve the illuminated lamps.

The test model for this SUT contains an input (TE2SUT) signal `State` that can be used with enumeration values `{Dark, Stop, Warning, Drive}` and output (SUT2TE) signals `L1`, `L2` and `L3` of type bool that indicate whether the different lamps are active or not.

In the test step described in figure 2, the function `getNextTimestamp` did return 100, which is the next time stamp for that stimulations are defined. The function `getStimulations(100)` calculates the stimulations for time stamp 100 from the stimulations file (`stimulations.json`). In this example, the only stimulation is that the TE2SUT signal `State` should become `Stop`. The `remoteInterpreter` is used to execute `ifm.setState(Dark)` inside the VDM model which copies the stimulation into the TE2SUT signal attribute `State` of the IFM. The function `logSutInputs` adds the stimulation to the test log (replay.log). The IFM operation `performStimulations` is used to calculate the SUT stimulations from the changed TE2SUT signal attributes of the IFM. In this case, the SUT operation call `setSignalState(stop)` is performed by the IFM inside the VDM model. As a reaction to the stimulation, the SUT outputs have changed. The operation `getChangedOutputs` retrieves the set of active lamps from the SUT using `getActiveSignals` and compares the active lamps with the values of the SUT2TE attributes `L1`, `L2` and `L3` of the IFM. The values for the lamps `L1` and `L2` have changed and `getChangedOutputs` returns `{L1, L2}` to the test driver. The test driver retrieves the values for the changed signals from the IFM and writes them to the test log (`logSutOutputs`).

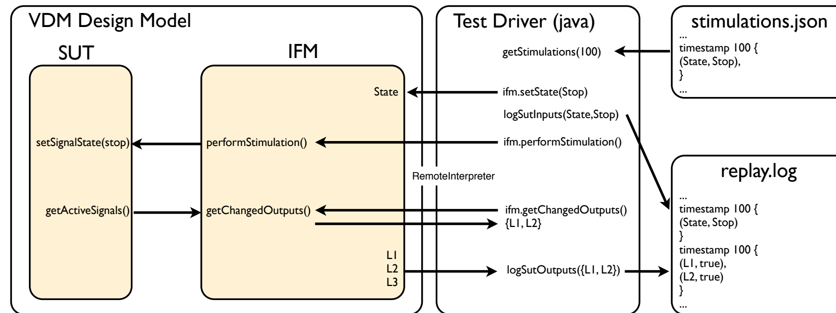


Fig. 2. A single test step at time stamp 100

After a complete test execution, the test log contains all stimulations to the SUT as well as all SUT outputs together with their time stamps. This information is used by the RTT-MBT replay tool to replay the result against the test model and to calculate a test verdict.

6 Test Verdict Calculation Using Replay

A replay of the test execution log against the test model is used to calculate the test verdict of a test execution as well as the test case and requirements

coverage. The test model defines the correct behaviour of the system under test. During replay the inputs and outputs of the SUT in the test execution log are checked against the expected behaviour specified in the test model. Test models used with RTT-MBT are deterministic so that each trace of inputs and outputs identifies a unique trace through the model. Each deviation of the outputs in the execution log to the expected outputs according to the test model are interpreted as a failure of the test case related to this behaviour. Tolerances are used for acceptable deviation of the SUT in timing or values of expected SUT outputs. The result of a replay is a log file listing all test cases covered during the test execution together with the PASS/FAIL information obtained for this test case during replay. More Information about replay of test results and how it is used with RTT-MBT can be found in [2]

7 Related Work

The VDM tool Overture provides tool support for automated combinatorial testing of VDM models described in [5]. This test automation is based on VDM traces which are conceptual similar to UML sequence diagrams. VDM traces are expanded into a collection of test cases, which can be executed and evaluated. The strength of this approach is detecting internal inconsistencies, in particular missing pre-conditions, but also inconsistencies in invariants or post conditions with the focus on unit-level tests rather than integration testing.

8 Concluding Discussion

The framework described in this paper uses Overtures VDM interpreter to allow model-in-the-loop testing of VDM design models using test procedures generated from UML/SysML test models. A generic algorithm for the test driver and an architecture for powerful interface mappings are defined. The approach described provides test generation and execution on a different test integration level than the existing testing capabilities of Overture. The test model and the test procedures generated from it can be used with different abstraction levels of the design model and with different test integration levels as long as the interface mapping is adjusted to the respective SUT. Mapping interfaces from the test model to complex interfaces of the system under test that can include protocols on the interface can be a challenging task and is a significant part of developing a test environment. The concept of interface modules supports this task because it allows to separate the complexity of the interface from the behaviour of the SUT itself. Test models for RTT-MBT test generation allow to define requirements tracing information in the test model. This way the tracing from requirements to test procedures can already be defined during the system design. Difficulties in defining a mapping between the design model and the test model interfaces can point to problems in the test-ability of the design.

8.1 Future Work

A prove of concept implementation for this test framework will be developed and tested with a small design models and test models. In addition, this implementation will be extend for use with the COMPASS project where the additional capabilities of CML[3] compared to VDM will have to be taken into account and will be integrated in the RTT-MBT plug-in of the COMPASS tool suite.

8.2 Acknowledgment

I would like to thank Kenneth Guldbrandt Lausdahl, Claus Ballegaard Nielsen and Jan Peleska for valuable input on the work presented here.

References

1. K. Ballegaard Nielsen, C. Lausdahl and P. Larsen. Combining vdm with executable code. In *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316, pages 266–279. Springer, 2012.
2. J. Brauer, J. Peleska and U. Schulze. Efficient and trustworthy tool qualification for model-based testing tools. In *Testing Software and Systems*, volume 7641 of *LNCS*, pages 8–23. Springer, 2012.
3. J. Bryans and J. Woodcock. CML definition 2. Technical report, COMPASS Deliverable, D23.3, March 2013.
4. P. G. Larsen, K. Lausdahl, A. Ribeiro, S. Wolff, and N. Battle. Overture VDM-10 Tool Support: User Guide. Technical Report TR-2010-02, The Overture Initiative, www.overturetool.org, May 2010.
5. P. G. Larsen, S. Wolff, N. Battle, J. Fitzgerald, and K. Pierce. Development process of distributed embedded systems using vdm. Technical Report TR-2010-02, The Overture Open Source Initiative, April 2010.
6. J. Peleska. Industrial-strength model-based testing - state of the art and current challenges. In *EPTCS*, volume 111, pages 8–28, 2013.
7. J. P. Wen-ling Huang and U. Schulze. D34.1 test automation support. Public document, available under http://www.compass-research.eu/deliverables.html, COMPASS, January 2013.

Co-modelling of a Robot Swarm with DESTECS

Ken Pierce

School of Computing Science, Newcastle University,
Newcastle upon Tyne, NE1 7RU, United Kingdom.
kenneth.pierce@ncl.ac.uk

Abstract. In this paper a DESTECS co-model of 10 robots (Kilobots) is described. Interesting aspects of the co-model are highlighted and the experience is used to evaluate the current DESTECS technology for modelling robotic swarms.

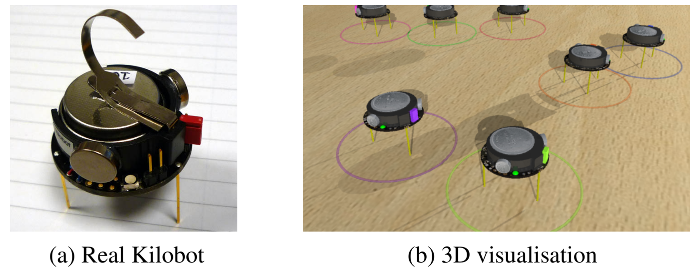
1 Introduction and Background

A robot swarm is a group of robots that act together autonomously towards some common goal. Suggested uses for such swarms include exploring hazardous environments (e.g. [1]). Swarms are assumed to be inherently dependable due to the redundancy introduced by the use of multiple robots, however establishing trust in such systems is complicated by their challenging nature. A swarm represents a cyber-physical system: multiple physical elements are controlled by multiple corresponding cyber elements. In addition, swarms may be formed from heterogenous robots with specialised functions, communicating wirelessly in a hazardous environment. Therefore development of such systems is challenging, however (formal) modelling and simulation can help in the design of such systems, permitting the design space of such swarms to be explored, with designs being tested and assessed early-on in the development process, increasing confidence in dependability before prototypes are produced.

This paper describes a model of a robotic swarm using the DESTECS tool[1], which enables the definition and simulation of *co-models*. A co-model comprises a discrete-event (DE) model that shares data with a continuous-time (CT) model. The nature of the communication between these models is defined in a "contract" that specifies the names and types of the data. Shared data includes scalar values, arrays and matrices, but is currently limited to Boolean, integer, and real types. The DESTECS approach has previously shown its utility in analysing systems with a single controller described in the DE formalism, interacting with a single plant described in the CT formalism (e.g. [2]). This paper describes work where a co-model of a robot swarm was created, which naturally requires multiple controllers and plant elements. From this experience, an assessment of the DESTECS tool for modelling such swarms is made.

The DESTECS tool currently supports the Vienna Development Method (VDM) [3] for DE modelling. The Overture[2] tool [4] provides the DE simulation engine. VDM supports object-orientation and concurrency [5], and provides features to describe real-time embedded systems, including the deployment of control processes onto networked

[1] http://www.destecs.org
[2] http://www.overturetool.org/



(a) Real Kilobot (b) 3D visualisation

Fig. 1: Photograph of a Kilobot showing two of the three legs, the two vibrating motors on the side, and the battery on top (a); and a 3D visualisation from 20-sim showing 7 Kilobots. The circles indicate the communication range of the Kilobots (b).

abstract CPUs [6]. The DESTECS tool supports the 20-sim[3] tool [7] for CT modelling, which allows CT models to be built, simulated and visualised. The 20-sim tool allows the dynamics of the plant to be modelled in several ways, including the powerful bond graph [8] notation, which is a domain-independent description of a physical system's dynamics, realised as a directed graph.

In the remainder of this paper, Section 2 describes the robots that were modelled and interesting details of the co-model. Section 3 looks at the strengths and weaknesses of the current DESTECS in relation to modelling robot swarms. Section 4 draws some conclusions and describes the next steps in this work.

2 Case Study: Kilobot Swarm

The Kilobot [9][4] is a small, cheap robot designed for swarm robotic experiments. They are currently commercially available from the K-Team[5] corporation. A Kilobot (see Fig. 1a) is 32mm in diameter and 20mm high. It has three fixed legs and uses two vibrating motors for movement, resulting in an energetic if imprecise form of locomotion. The on-board CR2032 rechargeable lithium battery can be charged by connecting the metal "crest" and one leg to a voltage source. Kilobots have an 8MHz processor and can run a single-threaded C program, with a simple API providing functions to set the motors, read incoming messages and so on.

Kilobots can communicate with each other by sending short messages (3 bytes long) up to a distance of 7cm using infrared (IR) light-emitting diodes (LEDs). Messages are communicated by bouncing IR signals of the surface underneath the Kilobots. Upon receiving a message, the strength of the signal is used to estimate the distance to the sending robot. Communications and distance estimates are dependant surface reflectivity and ambient light levels. The potential for corrupt and dropped messages makes for an interesting environment in which to build dependable swarms.

[3] `http://www.20sim.com/`
[4] A longer TR version of this paper is available as [10].
[5] See `http://www.k-team.com/`

A co-model of 10 robots was built to test the capabilities of DESTECS in modelling robot swarms (output visualised in Fig. 1b). Two scenarios described by Rubenstein et al. [9] were replicated in the co-model. In the dispersion scenario, the Kilobots move around until they no longer receive messages from other Kilobots, which will tend to make the swarm disperse within an area. In the orbit scenario, a stationary Kilobot regularly broadcasts a message and a second Kilobot uses the distance information to "orbit" around the stationary bot by maintaining a fixed distance. Without the information from the stationary Kilobot, the vibration-based motion is not precise enough to allow a Kilobot to follow a path accurately.

2.1 DE Model

As described previously, the Kilobot has a single-threaded controller and a basic API to access the robot's functions. This is mirrored in the VDM model and the object-orientated features of VDM are used for structuring. Each controller is represented as an object of an `KilobotController` class. This class defines a periodic thread and the *jitter* parameter is used to ensure that the order in which the controllers execute their loops is not fixed. Controller objects are each deployed to a CPU. The `KilobotController` class provides operations equivalent to the API provided in the real Kilobot, such as setting the motor speeds or reading incoming messages. Controllers must extend this class implement the `user_program()` operation. This approach means that apart from the need to have a constructor, the definition of a Kilobot controller in VDM is very similar to the corresponding C implementation.

Each `KilobotController` object has a unique identifier so that the correct shared variables can be accessed for each robot. Access is provided through a single `SharedVariables` class, which provides getter and setter operations with the robot identifier as a parameter. Following the *IO Synchronisation Pattern* described in the DESTECS methodology [11], the `SharedVariables` class defines a periodic thread and updates all shared variables regularly in a single atomic step. This significantly increases the speed of simulation at the expense of fidelity, however it is justifiable due to the simplicity of the robots being modelled. For example, no CT-DE events are required, which would be affected by this approach. Synchronization constraints are defined to protect concurrent access to shared variables.

Communications between robots are modelled on the DE side, following the *Ether Pattern* described in the DESTECS methodology [11]. An `Ether` class is used to represent a communication medium between the Kilobots. Each `KilobotController` class has access to this 'ether' and can broadcast messages to it. Upon receiving a message, the `Ether` class calculates which Kilobots are close enough to receive the message and passes it to the relevant controller objects. This means that the location of each robot must become a monitored variable that the `Ether` class can access. Currently communications are perfect, however the affect of messages clashing and of reduced reflectivity or increased ambient light could easily be added.

2.2 CT Model

The CT model is based on a two-wheel robot described by Pierce et al. [2]. While this provides a good initial approximation of the movement of the real Kilobots, it does not replicate the 'bouncy' and somewhat erratic movement produced by the vibrating motors. While this is sufficient for general algorithms such as dispersal, results from more precise behaviours will not currently map to the real Kilobots (e.g. tuning parameters for the orbit controller will not be reliable). In addition, there is currently no collision detection / response in the CT model, as this is an open area of research for bond graph models. So again at this stage only general conclusions can be drawn from simulation results, though as described above the Kilobots are able to interact through message passing modelled on the DE side.

Each Kilobot is represented as a submodel in 20-sim (i.e. as a single block in the visual model), with inputs representing values from the controller and outputs reporting the robots position, orientation and sensor values. Following the DESTECS methodology [11], all of the submodels are connected to a 'controller' block that acts as a communication point to the DE model. As with the DE model, each robot has an identifier so that is can access data that is specific to that Kilobot (such as starting location and orientation). Unlike VDM however, the 20-sim tool does not have a notion of object-orientation. Each submodel block must be replicated statically before simulation (i.e. 10 robots requires 10 blocks). This is a particular problem if changes needed to be made, but is easier to do if each implementation is identical. To ensure this, the identifer is provided as a 'signal' to each Kilobot submodel. The CT model currently contains 10 submodels, giving a 10 robot swarm. This replication must also be made in the 3D visualisation, which is a laborious process. Since 20-sim files are actually XML files, it is possible to open them with a text editor and replicate the relevant portion of the scene graph, using the search/replace function to connect this new (e.g. `kilobot1\position\x` becomes `kilobot2\position\x`). Such functionality would be relatively straightforward to script.

2.3 Co-model

Two matrices were defined in the contract for communication between the DE and CT sides (one for monitored variables and one for controlled variables). This was done as there are seven controlled variables and five monitored variables per robot, which results in 120 individual values for 10 Kilobots (which would make for a large and cumbersome contract). As mentioned in the previous section, the need to statically replicate robots in the CT model essentially makes the maximum number of robots static (currently 10). A co-simulation using fewer robots is possible however, though it requires some model-level tweaks. The DE side simply needs to know the desired number of robots (e.g. two in the orbit scenario), it then only instantiates this number of controllers and the `Ether` class only considers these objects for message passing. Since there is currently no interaction between Kilobots on the CT side, they can simply be 'hidden' in the 3D visualisation (e.g. by placing them 100 metres away) and essentially take no part in the co-simulation.

3 Evaluation of DESTECS for Swarm Robotics

As described in previous sections, a co-model of a swarm of 10 Kilobots was successfully built and simulated using the DESTECS tool. Based on the experience, the following two lists summarise the key strength and weaknesses in the current DESTECS technology with respect to modelling a swarm of robots. Many of these observations will be applicable more generally to co-models with multiple controllers and controlled plant.

Strengths

- Co-modelling allows for high-fidelity plant models to be coupled with sophisticated controller models.
- The DESTECS approach is sufficiently general to permit modelling and simulation of co-models with multiple controllers and plant.
- Composite shared variables (matrices) allow for straightforward definition of sets of monitored and controlled variables.
- The 3D visualisation of 20-sim is useful for software designers in assessing the affects of different controller schemes.
- The object-orientation of VDM allow for easy replication of controllers on the DE side.
- The generality of VDM allows for communications to be modelled between Kilobots.

Weaknesses

- The lack of object-oriented type features in 20-sim limits the ability to run co-simulations with a dynamic number of robots.
- It is currently difficult to model collision detection and response in CT models in 20-sim.
- Co-simulation speed increases significantly with each additional Kilobot if the synchronisation is not controlled DE side.
- Instantiating alternative controllers for different scenarios on the DE side must be controlled at a model-level with a conditional statement.

4 Conclusions and Future Work

This paper described a model of swarm of 10 robots using the DESTECS approach. Interesting aspects of the co-model were highlighted and an evaluation of the strengths and weaknesses of DESTECS for modelling such systems was made. This work described demonstrates that is is possible to use the current DESTECS technology to model a swarm of robots. Though the tool was designed primarily for single-controller, single-plant models, it does not restrict co-models to one-on-one simulation. A number of model-level "tricks" are needed to achieve certain results however.

Some relatively small changes to the tool would help improve its ability to cope with such "multiple-model" systems. For example, the ability to replicate submodels / 3D representations in 20-sim, and to switch between `System` classes in VDM for different scenarios (permitting different controllers to be instantiated).

Further work on the co-model itself should focus initially on modelling of realistic behaviours. On the DE side, this mainly involves realistic IR communications including lost and corrupted messages. On the CT side, modelling “jumpy” movement caused by the vibrating motors should be done in the first instance, followed by some form of collision detection and response. All of these improvements should then be tested against empirical evidence from experiments on the real Kilobots to establish if the fidelity of the co-model is sufficient to predict the behaviour of real swarms.

Acknowledgments

The work described here was funded under the European Community’s 7th Framework Programme (Grant agreement 248134, DESTECS) and the UK EPSRC Platform Grant on Trustworthy Ambient Systems. The author is grateful to Martin Mansfield for experiments performed on a real set of Kilobots.

References

1. W. Burgard, M. Moors, and F. E. Schneider, “Collaborative exploration of unknown environments with teams of mobile robots,” in *Advances in Plan-Based Control of Robotic Agents*, pp. 52–70, 2001.
2. K. G. Pierce, C. J. Gamble, Y. Ni, and J. F. Broenink, “Collaborative modelling and co-simulation with destecs: A pilot study,” in *3rd IEEE track on Collaborative Modelling and Simulation, in WETICE 2012*, pp. 280 – 285, IEEE-CS, June 2012.
3. P. G. Larsen, B. S. Hansen, H. Brunn, N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, *et al.*, “Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language,” December 1996.
4. 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, January 2010.
5. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.
6. M. Verhoef, P. G. Larsen, and J. Hooman, “Modeling and Validating Distributed Embedded Real-Time Systems with VDM++,” in *FM 2006: Formal Methods* (J. Misra, T. Nipkow, and E. Sekerinski, eds.), pp. 147–162, Lecture Notes in Computer Science 4085, 2006.
7. J. F. Broenink, “Modelling, Simulation and Analysis with 20-Sim,” *Journal A Special Issue CACSD*, vol. 38, no. 3, pp. 22–25, 1997.
8. V. Duindam, A. Macchelli, S. Stramigioli, and H. Bruyninckx, *Modeling and Control of Complex Physical Systems*. Springer, 2009.
9. M. Rubenstein, C. Ahler, and R. Nagpal, “Kilobot: A low cost scalable robot system for collective behaviors,” in *ICRA*, pp. 3293–3298, 2012.
10. M. Rubenstein, C. Ahler, and R. Nagpal, “Kilobot: A low cost scalable robot system for collective behaviors,” tech. rep., Computer Science Group, Harvard University, 2011.
11. J. F. Broenink, J. Fitzgerald, C. Gamble, C. Ingram, A. Mader, J. Marincic, Y. Ni, K. Pierce, and X. Zhang, “D2.3 — Methodological Guidelines 3,” tech. rep., The DESTECS Project (CNECT-ICT-248134), available from `http://www.destecs.org/`, December 2012.

Towards Modelling Systems of Cyber-Physical Systems

Martin Mansfield and John Fitzgerald

Newcastle University, Newcastle upon Tyne, NE1 7RU, UK
firstname.lastname@ncl.ac.uk

Abstract. Complex cyber-physical systems (CPSs) can be thought of as systems of systems (SoSs) in which constituent systems involve close interaction between software and the physical environment. The exploration of design spaces, the verification of conformance and emergence, and the analysis of performance for such systems requires the analysis of both cyber and physical properties. Current technology facilitates the creation of heterogeneous co-models of single CPSs, but does not make it convenient to model interaction between multiple CPSs. SoS modelling provides a framework for representation of the relationship between complementary systems. We conjecture that a combination of modelling technologies for CPSs and SoSs might enable the demanding forms of analysis required by these products.

1 Introduction

In a Cyber-Physical System (CPS) a computing and communication core monitors, controls and integrates the operation of multiple physical and engineered systems. Examples include cyber-controlled networks of devices in, for example, manufacturing plant, sensor networks, infrastructure such as smart grid or water supplies, and robot swarms. CPSs combine characteristics of embedded systems and systems of systems. As in embedded systems, they involve the close interaction between a digital world typically described in discrete event terms with a physical world that is usually analysed using mathematical models based on continuous time. In contrast to a stand-alone embedded system, a CPS is considered as a *network* of interacting elements that themselves integrate ICT with the physical domain [1]. The CPS concept thus represents a shift of focus to the integrity of the *coupling* of computational and physical components. In many applications, the constituents of the CPS may themselves be independently owned or managed, with the result that they may evolve over time outside the control of the CPS as a whole. In spite of this, reliance may have to be placed on the emergent behaviour of a CPS. All these characteristics are shared with systems of systems [2–4].

Given the need for dependability in CPSs, methods and tools are required to support the verification of conformance and emergence, as well as a means of discovering hitherto unidentified emergent behaviours. One approach to gaining confidence is through the production, static analysis and simulation of descriptive models of CPS behavior. However, challenges arise in producing such

models due to the heterogeneity of the constituent elements. Whilst software engineers intrinsically use rich DE models to describe the supervisory control of such systems, it is preferable to engineers of diverse disciplines implement more appropriate CT techniques to describe the controlled components in the physical world.

There is a body of research on heterogeneous collaborative modelling for embedded systems, and a largely separate body of (ongoing) work on SoS modelling, but we are not aware of much work that seeks to combine the two areas. DESTECS[1] introduces modelling primitives, guidelines and tools for defining co-models allowing discrete event (DE) and continuous time (CT) models to be co-simulated through a common harness, based on a reconciled operational semantics. DE and CT models are linked through a common interface specification (called a *contract*) that identifies shared (monitored/controlled) variables, design parameters and events [5]. The DESTECS framework supports co-simulation, but work has not been done on supporting the verification of conformance or of emergent behaviours. The framework has been instantiated using VDM [6] (extended with Real Time modelling features [7] and supported by the Overture tool[2]) and 20-sim [3] [8] as the DE and CT modelling formalisms respectively, although the authors have reported initial experience coupling VDM with MatLab [9] through the same co-simulation engine.

There is a large and growing body of work on model-based approaches SoS Engineering [10]. SoS typically describe a collection of task-oriented or dedicated constituent systems, where the combination of constituents collectively offer a system with greater functionality and performance than simply the sum of that of the constituent systems, by combining resources and individual capabilities in a collaborative fashion [11].

Definitions of SoS vary from application to application, with common themes throughout, e.g. [12, 13]. DeLaurentis [14] states that SoS problems are a collection of trans-domain networks of heterogeneous systems that are likely to exhibit emergent and evolutionary behaviors that would not be apparent if the systems and their interactions are modeled separately. Inherent to SoS problems are several combinations of traits, not all of which are exhibited by every such problem [3, 4].

Maier [2], formalised five common traits for identifying SoS challenges, from managerial and operational independence of components, to geographical distribution and evolutionary development of components, as well as emergent behaviours. DeLaurentis [14] has proposed additional traits to be considered from the study of mathematical implications of modeling and analysing SoS challenges, including inter-disciplinary study, heterogeneity of systems and networks of systems.

Depending on the combination of characteristics exhibited, SoS can be categorised according to the degree of managerial control [2, 15]. Distinctions of SoS

[1] http://www.destecs.org/
[2] http://www.overturetool.org/
[3] http://www.20sim.com/

range from *fully directed*, where a SoS is designed to fulfil a specific purpose by way of integration of constituent systems), to *virtual*, where a SoS lacks both central management and any agreed purpose.

Woodcock et al. [16] a formal foundation for SoS modelling, supporting both simulation and static analysis. Complementary technologies and formalisms are used to describe both functional and behavioural aspects of constituent systems and verify global properties of the SoS. Technologies include Systems Modelling Language (SysML) for describing architectural descriptions, and CSP [17] for representing concurrency and communication and VDM for data and functionality. An extension of SysML [18] provides the functionality to express rigorous interface contracts [19], enabling comparison of architectural design approaches. Interfaces of constituent systems can be defined for use of system developers, or to be used for system assessment in an effort to gain confidence that constituents adhere to the expected interface specifications.

Utilising this multi-paradigm approach is advantageous over previous approaches to SoS modelling, such as study by Bryans et al. [20] working entirely in Event-B, in that it more clearly shows interfaces between the SoS constituents, highlighting potential structural approach alternatives. Utilising multiple formalisms also emphasises interaction behaviour between the SoS constituents, made explicit in CSP rather than obscured in event guards. It enables verification of properties of the entire SoS that intersect multiple considerations. Whilst [20] provides a less transparent representation of SoS architecture and interaction behaviour, it does facilitate automated verification tools that can be developed for a single formalism. The multiple formalism approach lacks a consistent semantic base, so automated analysis of a similar type is limited.

The COMPASS project provides tools and techniques to support a formally grounded model-based approach to developing SoS by introducing the COMPASS Modelling Language (CML). Extending widely spread industrial notations such as SysML by the addition of formal CML notation, augments SoS modelling by way of additional tools and techniques to enable informal SoS development to be undertaken under the guidance of CML analysis techniques to later allow for the convenient introduction of formal SoS development.

2 Example of a Complex CPS

An example of a complex CPS can be demonstrated by intelligent energy saving through smart grid services: A cloud service infrastructure integrating and interacting with a continually growing set of third-party systems. A grid manager can take several kinds of data from various independent sources, and determine interaction with the physical domain by way of manipulation of energy provisions between nodes, based on computation of the sensory input. Data sources, energy suppliers and consumers may each be independently operated and managed.

Data transfer forms the foundation of the energy services, such as metering data. Metering data can be collected from a range of contrasting smart meters, and transferred between metering solutions offered by partners. Each metering

system forms an independent, autonomous system. Other data includes spot prices of energy from different sources, specific requirements of a consumer, as well as carbon emission data or meteorological data.

Based on the computation of the various data sources, devices directly manipulate energy flow between various sources over various mediums. From switching devices on or off to changing operating modes of devices based on a given computation, such a system exhibits definite traits of a CPS, and so it is evidently advantageous to utilise a co-modelling approach to its representation.

Such a system also strongly satisfies several characteristics that make it a SoS. Third party constituent systems are independently owned and managed, and are autonomous. The overall SoS delivers a service that cannot be offered by any of its constituents, which can be seen as an advantageous emergent behaviour. The constituent systems may also evolve in both their size and the features they might offer. A central enabling system compliments external constiuents by the addition of central management and distribution services to coordinate constituents.

Constituents that fall within the boundary of the enabling system include cyber elements such as data storage and management. This might range from energy prices and abundance from a given source, to real-time requirements and any specific constraints of a given consumer. Other internal constituents may fall within the physical domain. From communication mediums to energy storage, the SoS is likely to control its own physical components. The structure of such constituents may be centrally managed, but depend on systems outside of the SoS boundary.

Constituents that fall outside the enabling system boundary include all third-party providers and consumers. These systems are unlikely to be directly modified by the SoS, but instead can provide an interface outlining how they can be expected to interact with other constituents, and any services provided by other constituents that they may depend upon.

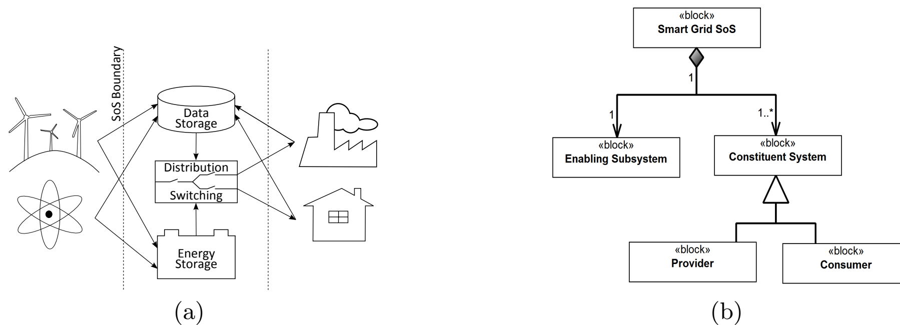


Fig. 1. (a) - Some key consitituents of a smart grid, and (b) - a SysML representation.

Fig. 1(a) shows some of the key constituents of the smart grid. The figure demonstrates that constituents are not only linked by the transfer of data, but also through exchanges of energy.

There is strong suggestion that representing each constituent of such a system as a co-model is beneficial, in order to enable focus of the tight coupling of discrete computation and continuous physics. There is also good reason to represent the system as a whole as a SoS to explore any emergent behaviour. Current technology limits SoS design to discrete elements only, abstracting away from consequences on the physical domain. Such consequences might include data transfer that initiates a large number of components to draw energy from a single source (perhaps at a time when spot prices make a source more preferable). A single co-model of the relevant components makes it possible to model such a trend, and satisfy safety properties in both cyber and physical domains, but is intrusive to the managerial and operational independence of the components involved. A SoS approach maintains the independence of the constituents but is forced to make discrete approximations of the behaviour of any physical elements.

To extend SoS modelling to facilitate the specification of heterogeneous co-models of cyber-physical constituents would utilise complementary formalisms to enable identification and analysis of emergent behaviours occurring in both the cyber and physical domain.

3 Systems of Cyber-Physical Systems

A number of CPSs existing in a common environment can co-operate through combination of computational power and physical ability in order to demonstrate a collaborative functionality which far surpasses the capabilities of any constituent CPS.

Such a system can be modelled using a single co-model approach through the production of a single DESTECS model. A collection of CPSs can be expressed by creating representations of physical components of all constituent CPSs in a single 20-sim model, and instantiating an independent VDM controller to be associated with each one. This requires explicit named pairing between each DE/CT coupling. Interaction may be modelled at both DE- and/or CT-side, by either the use of functions to facilitate data transfer between controller objects, or by use of a dynamic-link library to maintain the state of the environment in which the collection of physical components co-exist, and calculate any interaction physics such as collision.

Whilst CPSs are best represented through a heterogeneous co-model, this approach does not facilitate modification of an entire collection of CPSs without direct access and modification of each constituent CPS. By approaching a set of complementary CPSs in a shared environment as a SoS, emergent behaviour can be explored without prior knowledge of the internals of each constituent system.

Modelling groups of CPSs as a SoS is beneficial where components exhibit traits of managerial and/or operational independence. With some groups of

cyber-physical components, a subset of components may be useful outside the context of the entire collection. Similarly, a subset of components may be able to satisfy some useful goal outside of the entire collection. In addition, evolutionary development and emergent behaviour are both common traits of cyber-physical components exhibiting behaviours of self-organisation.

In order to model such characteristics using a SoS approach, subsets of components can be modelled as constituents of an overall system. This approach is beneficial in that it can be used to give indication of structural approach alternatives, and emphasises possible interaction behaviour between constituents. Current SoS architectures are currently limited to the definition of DE constituents. In order to model cyber-physical components as a SoS, discrete approximations would have to be assumed, limiting continuous behaviour to coarse approximations and removing focus from the coupling of cyber and physical elements of constituent CPSs.

By combining co-modelling technologies and SoS architectures, complementary cyber-physical components co-existing in a common environment can be modelled as a System of Cyber-Physical Systems (SoCPS) to preserve the rich but continuous advantages of heterogeneous co-model abstractions, whilst maintaining clear representation of overall system structure and subsystem interaction. Building such models would enable design space exploration of multiple cyber-physical entities co-existing in a common environment through simulation of possible emergent behaviours.

3.1 Implementation

In order to exploit such an approach, several aspects must be considered. Firstly, CML must be extended to provide support for the specification of cyber-physical constituents represented as heterogeneous co-models. Using an extension to CML ensures overall system definition with an underlying formality to enable a range of analysis techniques. CML extensions can be added to a variety of languages to create models with sufficient syntactic detail and semantic strength while relating to understandable, industry standard models.

With CML extended, the current SoS modelling framework must be extended to enable design space exploration of SoCPS. Current SoS modelling frameworks are focused around the relationships between DE constituents. In order to model SoCPS, considerations of the physical domain must be added. The notion of physical state must be defined, and be accessible by constituent CPSs. Not only must SoCPS models maintain state, they must facilitate the definition and calculation of any interaction physics between constituents.

Finally, In order to complete design space exploration of SoCPSs, there must be the provision of a library of useful CPS models as well as a methodology for the creation of new heterogeneous cyber-physical models and their inclusion of a SoCPS.

3.2 Utilisation

In order to evaluate design space exploration of SoCPS, a series of examples of implementation alternatives of a range of SoCPS must be completed and analysed. Case studies of SoCPS should cover a series of classes, from centrally controlled collectives of identical CPSs, to self-organising collectives of identical CPSs, and from centrally controlled complementary CPSs to self-organising complementary CPSs.

SoCPS model semantics can be defined in such a way to facilitate model simulation, allowing investigation into verification of emergent behaviour, leading to the identification of important relationships within SoCPS. Not only can this be used to improve the design of collaborative and self-organising distributed algorithms and facilitate optimisation of emergent behaviour, but more importantly can identify a level of guarantee that an eventual emergent behaviour can be depended upon.

References

1. E. A. Lee, "Cyber physical systems: Design challenges," Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan 2008.
2. M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.
3. J. Boardman, M. DiMario, B. Sauser, D. Verma, L. Martin-MS, and N. Moorestown, "System of systems characteristics and interoperability in joint command and control," in *2nd Annual System of Systems Engineering Conference, Defense Acquisition University, Ft. Belvoir, Virginia*, 2006.
4. A. Sousa-Poza, S. Kovacic, and C. Keating, "System of systems engineering: an emerging multidiscipline," *International Journal of System of Systems Engineering*, vol. 1, no. 1, pp. 1–17, 2008.
5. J. Fitzgerald, P. G. Larsen, K. Pierce, M. Verhoef, and S. Wolff, "Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems," in *IFM 2010, Integrated Formal Methods* (D. Méry and S. Merz, eds.), vol. 6396 of *Lecture Notes in Computer Science*, pp. 12–26, Springer-Verlag, October 2010.
6. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object–oriented Systems.* Springer, New York, 2005.
7. M. Verhoef, P. G. Larsen, and J. Hooman, "Modeling and Validating Distributed Embedded Real-Time Systems with VDM++," in *FM 2006: Formal Methods* (J. Misra, T. Nipkow, and E. Sekerinski, eds.), pp. 147–162, Lecture Notes in Computer Science 4085, 2006.
8. J. F. Broenink, "Modelling, Simulation and Analysis with 20-Sim," *Journal A Special Issue CACSD*, vol. 38, no. 3, pp. 22–25, 1997.
9. C. Kleijn, P. Visser, and F. Groen, "D3.5 — Extension to Matlab/Simulink," tech. rep., The DESTECS Project (CNECT-ICT-248134), December 2012.
10. C. B. Nielsen, P. G. Larsen, J. Fitzgerald, J. Woodcock, and J. Peleska, "Model-based engineering of system of systems." Submitted to ACM Computing Surveys, September 2013.
11. M. Jamshidi, "System-of-systems engineering-a definition. ieee smc, oct., pp. 10-12," 2005.

12. J. Boardman and B. Sauser, "System of systems-the meaning of of," in *System of Systems Engineering, 2006 IEEE/SMC International Conference on*, pp. 6–pp, IEEE, 2006.
13. V. Kotov, *Systems of systems as communicating structures*. Hewlett Packard Laboratories, 1997.
14. D. DeLaurentis, "Understanding transportation as a system-of-systems design problem," in *43rd AIAA Aerospace Sciences Meeting and Exhibit*, vol. 1, 2005.
15. J. S. Dahmann and G. Rebovich Jr, "Crosstalk: The journal of defense software engineering. volume 21, number 11," tech. rep., DTIC Document, 2008.
16. J. Woodcock, A. Cavalcanti, J. Fitzgerald, P. Larsen, A. Miyazawa, and S. Perry, "Features of CML: a Formal Modelling Language for Systems of Systems," in *Proceedings of the 7th International Conference on System of System Engineering*, IEEE, July 2012.
17. C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.
18. Object Management Group, "OMG Systems Modeling Language (OMG SysML) v1.2," 2010. OMG Document Reference: formal/2010-06-02.
19. R. J. Payne and J. S. Fitzgerald, "Interface contracts for architectural specification and assessment: a sysml extension," in *Proc. Workshop on Dependable Systems of Systems, WDSoS*, 2011.
20. J. W. Bryans, J. S. Fitzgerald, and T. McCutcheon, "Refinement-based techniques in the analysis of information flow policies for dynamic virtual organisations," in *Adaptation and Value Creating Collaborative Networks*, pp. 314–321, Springer, 2011.

Modelling Different CPU Power States in VDM-RT

José Antonio Esparza Isasa and Peter Gorm Larsen

Department of Engineering, Aarhus University

Abstract. With the increasing proliferation of battery-powered embedded devices the need to find the most power efficient way of controlling the micro-controllers inside also increases. In this paper we demonstrate how it is possible to model and analyse the energy consumption of a VDM-RT model. This is done by enabling the CPUs to change between different energy-saving modes and then performing post-analysis of the logged traces and using the data sheets for the processor(s) used to predict the optimal usage of the selected hardware. We also indicate potential future work enhancing the Overture tool to supporting efficiently this kind of analysis.

1 Introduction

In many devices the minimal usage of power from batteries is a competitive parameter that influences the likelihood of consumers selecting your devices. Changing batteries is annoying for the user so maximizing the time between the need for this is advantageous. As a system architect in the early phases the size of the possible design space is enormous. This paper proposes the use of the Vienna Development Method – Real-Time (VDM-RT) dialect [8] and the tool support from the Overture platform [4] to assist the system architect in navigating the design space in pursuit of optimal energy consumption strategies.

The motivation for this work is an Ambient Assistant Living research project called e-stockings[1] where the aim is to produce a device which will assist elderly people with chronical health problems in the use of compression therapy. This device will be controlled by a battery-powered embedded system controlling mechanical parts [3]. The candidate micro-controllers are all able to operate in different power-saving modes, but the optimal use of these features depends on both the properties of the power-saving features of the processors and the way the deployed software utilizes these power-saving features. We believe that it is possible to make use of VDM-RT to give good direction to system architects that have this kind of challenge. In this paper we will present our initial ideas for doing so and focus on a design pattern that we think will be generally applicable for this kind of situation.

This paper proceeds with a short explanation about the specialities of the VDM-RT dialect in Section 2. Then Section 3 explains the concept of power-saving modes in micro-controllers. The main contribution of this paper come in Section 4 where the suggested design pattern is presented. After this different future work areas are presented in

[1] See `http://www.aal-europe.eu/projects/e-stocking/` for more information.

Section 5. Finally Sections 6 and 7 provides references to related work and concluding remarks respectively.

2 The VDM Real-Time Dialect

In this section we will provide a short explanation about the VDM-RT dialect, assuming that the reader already in general is familiar with basic VDM [2, 1]. VDM-RT is one of three dialects supported by the Overture platform including interpretation support [5] that will be exploited in this paper.

The VDM-RT extension is object-oriented and it supports concurrent models. It provides the necessary constructs to represent active classes and incorporates concurrency safety mechanisms. VDM-RT includes the notion of time: a system clock is running from beginning till the end of interpretation. The maximum precision allowed in the interpreter is 1 nanosecond. VDM-RT also provides the notion of processing units in the form of built-in `CPU` classes can be used to declare processing unit and speed (in Hz); different parts of the model are deployed to specified CPUs. CPUs can communicate between themselves through buses. VDM-RT constructs take time to be interpreted, this time is shorter or longer depending on CPU speed. There is also a special kind of CPU, which is present in all the VDM-RT models implicitly, the virtual CPU which per default is infinitely fast and its execution does not affect system timing.

3 CPU Power Modes

CPUs can incorporate different power modes to reduce power consumption. These modes achieve a reduction in power consumption by disabling CPU peripherals and/or reducing the CPU performance. Here we take as an example the ARM Cortex M3 processor in the PSoC5 platform. This CPU has three different power modes available: *Active*, *Sleep* and *Hibernate*. This platform presents several operational modes with different analog and digital resources available and can react to different wake up events on each mode. This feature affects power consumption.

The *active* mode permits code execution and the usage of all platform features, and is the most power consuming state. The *sleep* mode presents a power consumption three orders of magnitude lower than the active consumption and prohibits code execution. Only the bus interface is available. It is possible to switch back to *active* mode after receiving events. Finally, it is possible to enter a *hibernate* mode with the lowest power consumption without turning off the device. This mode does not permit code execution and disables all the system resources. It is possible to transit back to *active* mode after receiving a high level input in the system IO.

4 A Design Pattern to Model CPU Power Modes

The VDM-RT CPUs are constantly able to compute and communicate and thus represent a CPU that is active constantly. This abstraction is not appropriate if we are

interested in studying CPU power consumption. We propose the application of a pattern that makes it possible to represent different CPU power states. This pattern makes use of a Virtual CPU state class to represent different CPU operational modes that controls whether code can be executed or not. An overview of this pattern is shown in the UML class diagram in Fig. 1. The general idea in this pattern is that whenever a thread running application logic is scheduled-in by the VDM-RT scheduler, it will check if the state registered in the Virtual CPU state is active or sleeping before running. There is a second case in which the thread will always run but forces a change in the Virtual CPU state before doing so. Additional details will be provided in the subsections below.
This pattern makes use of the following entities:

`VirtualCPUstate`: represents the CPU state. This class is protected against race conditions and thread interference by mutexes and history counters.
`PowerLogger`: keeps track of the CPU state changes and logs them so they can be analyzed further and represented in a graph.
`ProcTRunner`: represents functionality that should be implemented as a procedural thread.
`RealTimeRunner`: represents functionality that is executed periodically in a periodic thread.

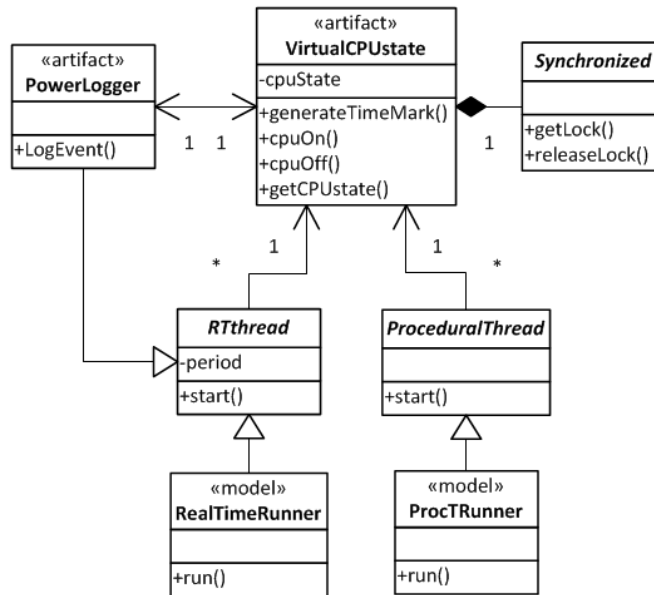


Fig. 1. Design pattern to model multiple CPU states.

The `VirtualCPUstate` class might be accessed by several threads in a concurrent manner. Therefore it is necessary to ensure thread safety in the operations that read

and modify the `cpuState` atrribute that represents the CPU operational mode. We have used mutexes to prevent data corruption.

Additionally and besides preventing the corruption of `cpuState`, we must ensure that the `getCPU` state function is executed every time there is a change in the power model. We have reinforced this policy by using permission predicates and history counters. This is shown in listing 1.

```
per turnOn =>  #req(turnOn) - #act(turnOn) = 1;
per turnOff => #req(turnOff) - #act(turnOff) = 1;
per getCPUstate => #active(getCPUstate) = 0  and
    cpuOn => #fin(turnOff) = #fin(getCPUstate);
```

Listing 1: Protection in VirtualState with history counters.

Whenever there is a change in the state of the Virtual CPU a method pushes this change to the `PowerLogger` class. The operation responsible for this is preceded by a **`duration(0)`** statement to avoid interfering the application logic timing.

This pattern sets a basic infrastructure to model upon different behaviour. We have modelled two different scenarios that are explained in the subsections below.

4.1 Modelling functionality that makes a CPU become active

In this case we model a scenario in which whenever certain functionality has to be executed the CPU is awakened. This is modelled in the `run` method modelled in the thread. This is modelled as shown in listing 2: the thread marks the state in the `VirtualCPUstate` class as `on`, executes the application logic and marks the CPU state as `off` again. The state changes are preceded by a 0 nanosecond duration statement, hence we are not accounting for the transition time between states.

```
duration (0) state.turnOn();
executeLogic();
duration (0) state.turnOff();
```

Listing 2: Conditional controlling the execution.

4.2 Modelling functionality that runs if CPU is active

In this second scenario we model the situation in which the application logic runs only when the CPU is active. We have used the structure presented in listing 3. This structure is modelled as part of the thread operation `run` that is executed when it is scheduled-in. In this case when the thread is thread is scheduled-in it will check the state of the Virtual CPU and determine if it is able to execute its application logic or not.

```
if not state.getCPUstate()
then duration (0) IO`print("\nCPU is off");
else executeLogic();
```

Listing 3: Conditional controlling the execution.

5 Further Work

The design pattern presented in this paper is a way to overcome at the modelling level current limitations of the Overture platform. We would like to take this idea further and possibly incorporate changes in the Overture tool and the VDM-RT language.

5.1 Unaddressed issues

This design pattern does not take into account the communication aspect when the CPU is sleeping. Here we should consider two scenarios: a) Information is received at the communication interface and this generates an interrupt that wakes up the CPU so the information can be processed and b) Information is received and discarded since the CPU is sleeping. These aspects have not been studied but are worthwhile exploring further in later work.

In previous work we studied the possibility of studying power consumption in mechatronic systems by applying co-simulation [3]. The co-simulation approach should be revisited to review its applicability to study computation and communication power consumption. Our initial thesis is that this could be especially beneficial for systems with multiple components (such a SoC with analog and digital blocks). However additional work is needed to confirm this.

5.2 Tool and language modifications

We propose to implement the sleep functionality in the VDM-RT java engine rather than at the modelling level. Additionally we would incorporate a function to register the events that can wake up a CPU from the sleeping state.

```
1 cpu.wakeOn(event); -- Configure event to wake up CPU
2 cpu.sleep(); -- Sleep the CPU at some point
```

Listing 4: CPU sleep operations.

Events that could wake up the CPU should be provided by an external periodic thread. It should be possible to associate concrete CPUs to certain events and feed the events to the CPU at a certain point of time.

```
1 duration (0) if time = eventTime
2              then eventGenerator.feed(cpu,event);
```

Listing 5: CPU sleep operations.

Besides these language modifications, additional tool work will be needed to generate consumption graphs automatically and to produce real-time logs in a more effective way.

6 Related Work

Modelling has been widely applied to study energy consumption in previous work but typically at a lower level of abstraction and by using platform specific models. The advantage of this approach is that the more detailed models are more accurate and easier to transition to a final implementation. Additionally the fact that the models are targeting specific platforms improves accuracy as well [7, 6]. Vijaykrishnan et al. make use of modelling in a joint hardware-software approach to optimize energy consumption and using virtual CPUs in [9]. In this case the authors use an ad-hoc modelling platform rather than a generic software modelling language like VDM-RT.

7 Concluding Remarks

This paper has presented a design pattern that can be used for incorporating power consumption considerations to a VDM-RT model. We believe that this can be beneficial for system architects exploring potential strategies for controlling micro-processors in different power saving modes. Naturally this is early work since no special tool support has been incorporated in Overture for supporting these ideas yet. However we believe that it would be possible to develop this so that the applicability for the Overture platform can be increased to also take power consumption considerations into account when desired.

References

1. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
2. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://www.vdmbook.com`
3. Isasa, J.A.E., Hansen, F.O., Larsen, P.G.: Embedded systems energy consumption analysis through co-modelling and simulation. In: International Conference on Modeling and Simulation, ICMS 2013. World Academy of Science, Engineering and Technology (June 2013)
4. 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`
5. 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
6. Mostafa E.A. Ibrahim and Markus Rupp and Hossam A. H. Fahmy: A Precise High-Level Power Consumption Model for Embedded Systems Software. EURASIP Journal on Embedded Systems Volume 2011(1) (January 2011)
7. Sheayun Lee and Andreas Ermedahl and Sang Lyul Min: An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors (2001)
8. 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)

9. Vijaykrishnan, N., Kandemir, M., Irwin, M.J., Kim, H.S., Ye, W.: Energy-driven integrated hardware-software optimizations using simplepower. SIGARCH Comput. Archit. News 28(2), 95–106 (May 2000)

Modelling a Smart Grid System-of-Systems using VDM

Stefan Hallerstede[1] and Peter Gorm Larsen[1]

Department of Engineering, Aarhus University
{sha,pgl}@iha.dk

Abstract. Using formal notations to analyse industrial cases that can be characterised as System of Systems is important. In this paper we report about our experience in modelling a distributed energy management system enabling energy savings for many organisations. The modelling made here using VDM also demonstrates areas where new notations such as the COMPASS Modelling Language (CML) may be beneficial.

1 Introduction

When different owners of Constituent Systems (CSs) collaborate to form a System of Systems (SoS) there will always be confidentiality and integration issues that need to be addressed appropriately. Similarly to the approach taken for "ordinary" systems [2] valuable insights can be obtained from rigorous modelling and analysis of the intended functionality (of the whole SoS using the services supplied by the separate CSs). In the COMPASS[1] project a new modelling language called CML, specially targeting the development of SoSs is being created [7]. CML is based on VDM++ [3] and CSP [4]. In order to precisely target specific problems of SoS modelling, different project members model different SoSs using the baseline technologies VDM++ and Overture to investigate the respective advances needed in the design of CML. This paper reports on one such (small) case study.

The case study deals with an SoS offering automated and intelligent energy saving and smart grid services. This is achieved by means of a cloud service infrastructure, integrated and interacting with a continually growing set of third party CSs. Furthermore, some systems can be more tightly integrated allowing additional services to be provided. However, providing such additional services requires dedicated hardware. The case study focuses on loosely integrated systems where the integration is more challenging.

Section 2 presents an informal description of the case study. Section 3 gives a short overview of the VDM model using a few UML class diagrams as well. Finally, Section 4 provides a few concluding remarks about this work and its future development.

2 The Smart Grid Case

One of the primary functions of a smart grid SoS is to control farm[2] appliances to perform tasks such as load shifting, that is, to use resources when they are cheap. Fig. 1 shows a sketch of a smart grid SoS with two farms A and B. The term "smart" is used to indicate that the

[1] COMPASS is an acronym for "Comprehensive Modelling for Advanced Systems of Systems".

[2] In order to simplify the language we only talk about "farm" appliances. Of course, this restriction is only for presentation purposes.

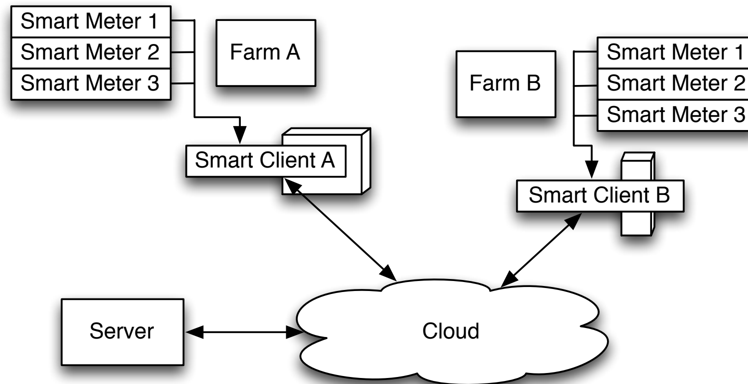


Fig. 1. A Smart Grid Consisting of a Server, the Cloud and two Farms

corresponding devices carry out some amount of data processing. A smart client consists of a gateway and several agents that connect to the various meters. The appliances controlled are not pictured. This is to emphasise that the smart grid at its heart is a data processing problem. The appliances are not the main concern of the smart grid. They are the farmers' concern. The control logic is evaluated by a cloud-supported server with varying complexity and may interact with one or more of the constituent systems in order to fulfil its tasks. The control logic is specified by means of rules. A simple example of a rule is "to switch on and off lights at certain times of the day". A specific problem of the smart grid SoS is that the control logic continually evolves, as does entire SoS independently at each farm. New rules and kinds of rules must be incorporated into the existing SoS without breaking it. The associated correctness criterion only concerns features of the SoS actually used. Correctness does *not* need to be guaranteed with respect to all behaviours that would be possible given a set of rule types.

3 A VDM Model of the Smart Grid

Figure 2 shows the run-time architecture of the VDM++ model of the smart grid. The CSs on which the case study focuses are modelled as threads so that communication between the CSs cannot simply be modelled by plain operation calls. Some sort of synchronisation or buffering is required. The devices to be controlled are contained in the meters in this model. Each meter contains one device. Meters, agents, gateways and the server that computes instructions to be executed by the agents are modelled with their own thread. The cloud and the engines that run within the cloud are properly contained in the server. Dealing with these does not pose any SoS specific challenges. The different stakeholders are taken account of by the way the model is tested. The global testing traces make no assumptions about the concrete behaviour of the devices. These can in principle be configured to match different profiles. The test traces and profiles can be changed independently from one another.

For concrete architectures the abstract classes are sub-classed and instances of concrete classes are produced using the singleton pattern (see also Figure 3).

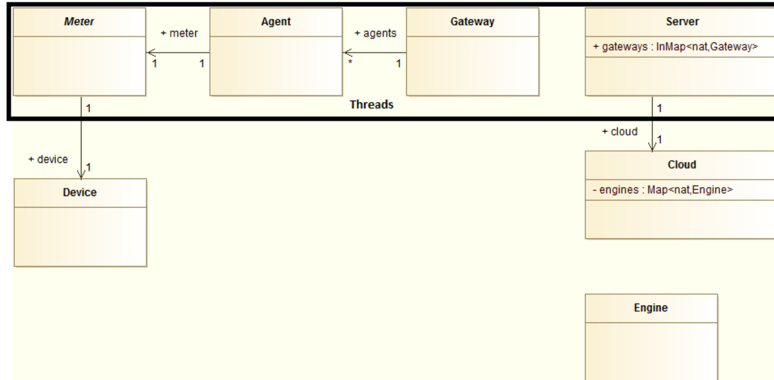


Fig. 2. Abstract Architecture of the SmartGrid VDM model

```
1  class Farm_A_Agent is subclass of Agent
2  instance variables
3  public static agent : Farm_A_Agent := new Farm_A_Agent();
4  operations
5    private Farm_A_Agent : () ==> Farm_A_Agent
6    Farm_A_Agent() == Agent(Farm_A_Freezer_Meter`meter);
7  end Farm_A_Agent
```

The connections between different CSs are modelled by instance variables referring to other constituent systems and by referring to the public static instance variables of the singletons.

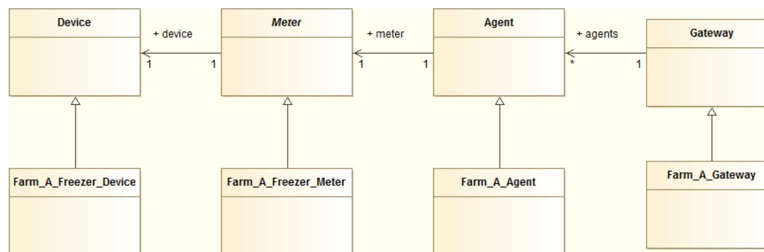


Fig. 3. Concrete Instantiation of the Architecture with two Farms

Because the `Cloud` class is not modelled as an independent CS it is contained in the `Server` class that feeds the Cloud with data and initiates computations. This can be seen from the `Step` function of the server.

```
1  class Server is subclass of Types
2  ...
```

```
3 private Step : () ==> ()
4 Step() == (
5   while true do (
6     let now = World`timerRef.GetTime() in
7     for all i in set dom gateways do (
8       let g = gateways(i) in
9       (cloud.receive_meterings(i,g.send_meterings());
10       let acts = cloud.compute(i,now) in
11         for a in acts do g.receive_action(a));
12     );
13     World`timerRef.WaitRelative(1)
14   )
15 );
16 end Server
```

The `World` class composes and drives the model. It loads scenarios describing events at the devices and rules describing agent rules and simulates the SoS with these stimuli. This is done by starting all threads pertaining to the model. The model uses the standard time abstraction of VDM called `TimeStamp` providing a model of discrete time and a convenient technique for executing the threads in lock step by establishing a thread barrier.

```
1 public World : nat * seq of char * seq of char ==> World
2 World(mytime, scenario, rules) == (
3   maxtime := mytime;
4   env := new Environment(self, mytime, SmartGrid`grid,
5    {"Farm_A_Freezer_Meter"|->Farm_A_Freezer_Meter`meter,
6     "Farm_B_Battery_Meter"|->Farm_B_Battery_Meter`meter},
7    {"Farm_A_Gateway" |-> Farm_A_Gateway`gateway,
8     "Farm_B_Gateway" |-> Farm_B_Gateway`gateway});
9   env.loadScenario(scenario);
10  env.loadRules(rules);
11  start(SmartGrid`grid);
12  start(Farm_A_Gateway`gateway);
13  start(Farm_A_Agent`agent);
14  start(Farm_A_Freezer_Meter`meter);
15  start(Farm_B_Gateway`gateway);
16  start(Farm_B_Agent`agent);
17  start(Farm_B_Battery_Meter`meter);
18  start(env);
19  start(self)
20 );
```

4 Concluding Remarks

In large parts we have used modelling techniques that are well-established in VDM. Some are entirely appropriate for the modelling of the smart grid SoS; others are not. The following

paragraphs collect our observations and preliminary conclusions. Final conclusions can only be drawn once a CML model will have been produced. Then, based on this model, we will be able to evaluate (within the limitations given by using one case study) advances of CML over VDM with respect to the modelling of SoS. The VDM model sets the base line of what has to be achieved and serves to formulate specific problems that need attention.

Architecture. We have modelled a configurable architecture by means of inheritance. Concrete architectures are produced by subclassing. This approach is common in VDM, in particular, with respect to modelling fault tolerance [6]. The only way that we deviated from the established approach is to use inheritance together with the singleton pattern. This permits us to keep all CS in different files. We believe this will be helpful when dealing with different sets of stakeholders (see also below).

Property Specification. Many SoS-wide properties can be conveniently expressed by referring to instance variables of different CSs. In VDM it is necessary to mark all those variables as "public" also exposing them to operations of other classes. It would be helpful if visibility for property specifications was more relaxed than for access by operations. We expect this to be important because CML will be a *modelling* notation in the first place where property specification is a prime concern.

Concurrency. We find that models of SoS display a high amount of concurrency. This appears mostly to be due to the independence of the different CSs. As a consequence of this, the SoS seems to require dealing with a lot of non-determinism. In this respect, the usual approach of model-improvement in VDM [5] that starts with a sequential model and turns this into a concurrent model appears less suitable. However, we still pretended to write a sequential model in the first step, although, we never executed it. Instead, we immediately turned it into a concurrent model for execution. This approach made us focus on data structures their properties and basic algorithms first. We were able to sketch the architecture without specifying threads. Once the model appeared rich and accurate enough we specified the different threads.

Time. We have used a very simple model of time in the current version of the smart grid model. Each thread executes a loop with one call `WaitRelative(1)` in each iteration. This model is not appropriate. The real-time model accompanying VDM-RT appears to too close to implementation on hardware CPUs. In SoSs we have to deal with different time bands [1] depending how "deep down" we look into the CSs. This sort of scaling time to right abstraction levels needs to be investigated. For now we just multiply `1` with constant factors to achieve this effect informally. This is certainly insufficient for serious modelling and analysis.

Communication. We have modelled communication by means of operation calls in the different threads, `Send...()` and `Receive...()`. The choice of parameters depended on whether a communication is of "push" or of "pull" type. Making this decision seems artificial and complicates the modelling. Channel-based communication such as offered by CSP will avoid this complication. It needs to be investigated how much we can profit from this possibility being available in CML.

Stakeholder Modelling. In the VDM models stakeholders are not represented at all. We have attempted to adhere to a modelling style that should make dealing with different stakeholders possible in the way that we are used to using configuration control systems. This needs to be investigated in more detail. We believe that the modelling notation should provide means to manage different stakeholders dealing, e.g., with issues such as *confidentiality* and *relevance*: Stakeholders should be explicitly granted access to different parts of a model and property specifications. They should also not be shown parts that are not of relevance to them. Models of SoS will usually be large and complex when considered as a whole.

Correctness. Appropriate notions of correctness are needed. We have specified correctness properties in the current model assuming that CSs may fail. The entire SoS will fail once any single CS fails. This is clearly unacceptable for a practical SoS. Now we could adapt the model so that CSs can no longer fail. But as a side effect of that we could no longer recognise failure of the SoS to work as expected as easily as before: failure indicated an error in the model. A solution to this problem could be to have plain CS models (that can fail) and fault-tolerant CS models and associated property specifications with certain configurations.

Acknowledgements

The work presented here is supported by the EU Framework 7 Integrated Project "Comprehensive Modelling for Advanced Systems of Systems" (COMPASS, Grant Agreement 287829). For more information see `http://www.compass-research.eu`.

References

1. Dongol, B., Hayes, I.J.: Approximating idealised real-time specifications using time bands. ECEASST 46 (2011), `http://journal.ub.tu-berlin.de/eceasst/article/view/684`
2. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
3. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), `http://www.vdmbook.com`
4. Hoare, T.: Communication Sequential Processes. Prentice-Hall International, Englewood Cliffs, New Jersey 07632 (1985)
5. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM. Intl. Journal of Software and Informatics 3(2-3) (October 2009)
6. Pierce, K., Fitzgerald, J., Gamble, C.: Modelling faults and fault tolerance mechanisms in a paper pinch co- model. In: Proceedings of the ERCIM/EWICS/Cyber-physical Systems Workshop at SafeComp 2011, Naples, Italy (to appear). ERCIM (September 2011)
7. Woodcock, J., Cavalcanti, A., Fitzgerald, J., Larsen, P., Miyazawa, A., Perry, S.: Features of CML: a Formal Modelling Language for Systems of Systems. In: Proceedings of the 7th International Conference on System of System Engineering. IEEE (July 2012)

Ken Pierce and Stefan Hallerstede, Proceedings of The 11th Overture Workshop, 2013

Department of Engineering
Aarhus University
Edison, Finlandsgade 22

8200 Aarhus N
Denmark

Tel.: +45 4189 3000