ISSN 1884-0760

GRACE TECHNICAL REPORTS

Proceedings of the 13th Overture Workshop

Fuyuki Ishikawa and Peter Gorm Larsen

GRACE-TR 2015–06 June 2015

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

The GRACE technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

Proceedings of the 13th Overture Workshop

23 June, 2015

http://overturetool.org/workshops/13th-Overture-Workshop.html

The 13th edition of the "Overture" series of workshops on the Vienna Development Method (VDM), its associated tools and applications, was held in conjunction with the FM 2015 symposium[i]. This workshop aims to provide a forum for discussing and advancing the state of the art in formal modelling and analysis using VDM and its family of associated formalisms including extensions for distributed and real-time systems.

Although VDM is one of the oldest formal methods to have enjoyed a level of industry use, it nevertheless has a lively and youthful research community, which has grown up around the development of the Overture open tools platform[ii]. On top of the Overture platform the Crescendo[iii] and Symphony[iv] tools from respectively the DESTECS[v] and COMPASS[vi] projects, as well as the new development that will take place in the new INTO-CPS project[vii] (see http://into-cps.au.dk/). The platform provides a vehicle for activity in modelling and analysis technology including static analysis, interpreters, test generation and execution support and model checking. The growth of this community has been greatly assisted by the Overture workshop series.

The 13th workshop reflected the breadth and depth of work in VDM. The invited talk by Taro Kurita (Sony) reports the evolution of its VDM usage for their development of Mobile Felica IC chips, focusing on the aspect of readability, i.e., formal models as documents.

Research contributions included topics about essential challenges in concurrency and real-time aspects. Connection and integration with a variety of directions are also covered, including requirements and stakeholders, theorem provers, and test-driven development. Last but not least, application and visionary papers are provided to promote discussions for next directions of the Overture community.

We would like to thank the authors for the interesting contributions and the PC members and reviewers for their advices to make this workshop valuable and successful.

Fuyuki Ishikawa

Peter Gorm Larsen

[i] The 20th International Symposium on Formal Methods (FM 2015): http://fm2015.ifi.uio.no/

[ii] Overture Tool: http://overturetool.org/

[iii] Crescendo Tool: http://crescendotool.org/

[iv] Symphony IDE: http://symphonytool.org/

[v] DESTECS Project: http://destecs.org/

[vi] COMPASS Project: http://www.compass-research.eu/

[vii] INTO-CPS Project: http://into-cps.au.dk/

Table of Contents

Advance in VDM Application to Development of Mobile FeliCa IC Chip Firmware - Toward Readable VDM Specification for Reliable System and Good Relationships ...1

Taro Kurita

JODTool on the Overture Tool to manage formal requirement dictionaries 3

Yoichi Omori, Keijiro Araki and Peter Gorm Larsen

VDM Animation for a Wider Range of Stakeholders 18

Tomohiro Oda, Yasuhiro Yamamoto, Kumiyo Nakakoji, Keijiro Araki and Peter Gorm Larsen

Integrating the PVSio-web modelling and prototyping environment with Overture 33

Paolo Masci, Luis Diogo Couto, Peter Gorm Larsen and Paul Curzon

Extending the Overture code generator towards Isabelle syntax 48

Luis Diogo Couto and Peter W. V. Tran-Jorgensen

Code Generation of VDM++ Concurrency 60

Georgios Kanakis, Peter Gorm Larsen and Peter W. V. Tran-Jorgensen

Generating Java RMI code for the distributed aspects of VDM-RT models 75

Miran Hasanagic, Peter Gorm Larsen and Peter W. V. Tran-Jorgensen

Improving Time Estimates in VDM-RT Models 90

Morten Larsen, Peter Wurtz Vinter Tran-Jorgensen and Peter Gorm Larsen

Case Studies on Combination of VDM and Test-Driven Approaches: Application, Model Finding and Refinement 104

Fuyuki Ishikawa

Pacemaker Parameter Tuning using Crescendo 116

Carl Gamble, Martin Mansfield, John Fitzgerald and Peter Gorm Larsen

TASTE for Overture to keep SLIM 132

Marcel Verhoef and Maxime Perrotin

Program Committee

Keijro Araki, Kyushu University, Japan

Nick Battle, Fujitsu, UK

John S Fitzgerald, Newcastle University, UK

Tomohiro Oda, Software Research Associates, Inc., Japan

Jose Nuno Oliveira, Minho University, Portugal

Nico Plat, West Consulting, Netherlands

Volker Stolz, Oslo University, Norway

Marcel Verhoef, European Space Agency, Netherlands

Advance in VDM Application to Development of Mobile FeliCa IC Chip Firmware - Toward Readable VDM Specification for Reliable System and Good Relationships (Extended Abstract for Invited Talk)

Taro Kurita

Sony Corporation, Japan taro.kurita@jp.sony.com

"FeliCa" is a contactless IC card technology developed by the Sony Corporation and is widely used in Japan. In particular, Mobile FeliCa IC chips are embedded in over 250 million mobile phones. Their applications, including electronic money, train tickets, identifications, door keys, and so on, form an essential foundation for business and daily activities in Japan.

Given the significance of the system, VDM was applied to the development of the second generation of its firmware [1, 2]. It successfully contributed to resolution of problems in the early phases, such as vagueness in the specification.

This talk discusses the succeeding development of the third generation started in 2007. This development involved many features, such as enhancement of the encryption mechanisms and adaptation of the global standard of Near Field Communication (NFC). The implementation code was three times the LOC of the second-generation. The usage of VDM was improved especially for maintainability and understandability, i.e., to facilitate the iterative process and communications among involved teams.

In the third generation, the VDM specification was considered as the sole specification document that worked as the reference for various development activities. In other words, it was not wrapped or hidden by natural-language documents, and was referred to by more engineers including those from external partner companies. This change eliminated costly and error-prone maintenance for two versions of the documents in the natural language and VDM. On the other hand, this change made readability of the VDM specification indispensable, though it had been found significant in the previous development.

To improve the readability, the specification convention was defined. This is like coding conventions, however, has different features. One of the most important and novel features was separation of the specification part and the mock-up part in the VDM model. The former part represents decisions that affect the following development activities and thus should be understood by the readers. The latter part is to make the model runnable with the interpreter for validation through specification animation or testing. Separation of these two parts is enabled by introducing auxiliary functions that denote abstract data operations (e.g., data addition) without depending on specific data structures (e.g., set union or sequence concatenation).

Table 1. Comparison of the Second and Third Generations

Generation	C Implementation [LOC]	VDM Specification [LOC]
Second	40,876	39,315
Third	126,944	55,400

(LOC excludes comments and blank lines)

Generation	Deficiencies by description	Deficiencies by comprehension	Productivity [LOC/Man-month]	Debug density [errors/kLOC]
Second	2%	16.3%	1,000	11
Third	0%	10.9%	1,000	11

There were other improvements including the following ones.

- Use of the Japanese in variable and method names that considerably decreased the amount of comments on the VDM specification.
- Management of multiple products in the specification and testing.
- Systematic methods for generating test cases from the VDM specification.

Table 1 shows comparison of the second and third generations. Increase of the specification lines was moderate by declarative description, even with the large increase of the implementation lines. Deficiencies caused by the description and the comprehension were both decreased, while the productivity and debug density were kept to a similar level.

Our experience demonstrated how readability of the VDM specification can be improved. We believe the readability is one of the key factors that support not only construction of reliable systems but also effective and harmonious relationships in the involved teams.

Acknowledgments

We would like to thank Professor Peter Gorm Larsen of Aarhus University, Shin Sahara of Hosei University and Hiroshi Sako of Designers' Den Corporation for their great assistance in the application of VDM.

References

1. Kurita, T., Chiba, M., Nakatsugawa, Y.: Application of a formal specification language in the development of the "Mobile FeliCa" IC chip firmware for embedding in mobile phone. In: The 15th International Symposium on Formal Methods (FM 2008). pp. 425–429 (2008)
2. Kurita, T., Nakatsugawa, Y.: The application of VDM to the industrial development of firmware for a smart card IC chip. International Journal of Software and Informatics 3(2-3), pp. 343–355 (2009)

JODTool on the Overture Tool to manage formal requirement dictionaries

Yoichi Omori[1], Keijiro Araki[1], and Peter Gorm Larsen[2]

[1] Department of Advanced Information Technology,
Graduate school of Information Science and Electric Engineering,
Kyushu University,
{yomori@ait,araki@csce}.kyushu-u.ac.jp
[2] Department of Engineering
Aarhus University
pgl@eng.au.dk

Abstract. The traceability between a software requirement specification in natural language and its corresponding formal specification plays an important role in development and maintenance of software. JODTool is a tool to support the activities of formal specifiers to bridge from a pre-formal specification to a formal specification. It manages a formal requirement dictionary, which is a set of tuples of a key-phrase and its definition to keep bi-directional traceability. The tool provides mainly three functionalities; 1) Key phrase marker 2) Dictionary editor, and 3) Format converter, on the Overture tool to map key phrases to VDM-SL/VDM++ specifications. We propose an evolutionary process of a round-trip between pre-formal specification and formal specification with the tool, and illustrate it with a small case study. The tool is seamlessly integrated into the Overture environment, and it lists concrete ambiguities in the SRS to be resolved, and supports prioritised modeling in VDM specification.

Keywords: Requirement Specification, Requirement Traceability, Overture tool, Formal Requirement Dictionary, VDM modeling

1 Introduction

A software requirement specification (SRS) expressed using natural language is the starting point of software development in most project. It is clear that tenders, contracts, or projects of a certain scale demand an explicit SRS. Natural language usually contains, however, contradictions and ambiguities, therefore careful reviews and repeating rewrites are required to achieve an agreement among stakeholders.

On the other hand, formal methods apply the use of mathematical notation in the specification, and the use of such specifications as a basis for the verified design, of computer systems [10]. A mathematical verification and practical use of tools are attained by formal methods at early specification phase. Each formal method provides its own language which has defined semantics to describe specifications. In other words, formal methods can eliminate ambiguities from natural language description, but considerable training is required to use it well. Therefore, documents in natural language are commonly used in communications among stakeholders including non software

engineers thorough the software life cycle, though a formal method is adopted in the software development.

Traceability is an important feature to use a natural language and a formal method together, in order to check whether the formal specification is covering the SRS in natural language. The advantage of requirement dictionary in the specification phase is well known, however it is often omitted because of its construction cost[2]. We had proposed a dictionary management tool named JODTool for formal requirements dictionary which manages the map between key phrases in SRS and their definition by formal language [16]. A formal requirement dictionary is a set of tuples made of a key phrase as a small semantic chunk and its formal semantic definition, which represents bi-directional translation between them.

JODTool extends application of lightweight formal method [11] to requirement specification process. The concept of lightweight formal method is utilisation of verification tools instead of rigid mathematical proves, which are typically constructed semi-automatically. The aims to introduce the formal requirement dictionary are;

- The semantic gap of natural language and a program would be buried by a formal specification.
- Verification by tools in small units to improve reusability and accuracy of correspondence.
- The mapping between a semantic chunk in SRS and its formal definition must be surjective.

A phrase is not a grammatical meaning here, but the minimum unit which makes it distinctly different from other phrases in the specification. For example, an "apple" is the minimum chunk in a system which distinguishes an "apple" and other obstacles, but if a target system distinguishes a "red apple" and a "green apple", the minimum chunk in the specification must be "red apple" and "green apple." In short, grammatical knowledge is insufficient to find key phrases and they should be identified by formal specifiers considering the purpose of the system and domain knowledge.

Formal specifier cannot construct the proper formal model all at once. They will write and verify a formal specification from important parts considering the priority of requirements. When the verification is successful, they extend the model to incorporate other requirements. Formal specifiers develop the specification by repeating this.

This paper proceeds by providing the related work that this work is based on in Section 2. Afterwards the notion of a formal requirement dictionary and its management possibilities on top of the Overture tool is explained in Section 3. Then the application procedure of an evolutionary formal modeling process with the tool is presented in Section 4. The case study of application to a parking deck system is reported in Section 5, and some analyses over the result is performed in Section 6. Finally, we summarize our results in Section 7.

2 Related Work

Tools to manage the mapping between requirements in natural language and program fragments are available as commercial products. Some tools to manage mapping be-

tween requirements in natural language and detailed design are available as commercial products, such as Rational DOORS [18], Mingle [19] and Caliber [17]. They store arbitrary mapping based on the traceability matrix which maps relations among listed specifications, design in some diagram or documents. However, it is difficult to assure that all requirements are turned into specification without any kinds of contradictions in large-scale software development.

The limits of natural language software specifications were pointed out, and the investigations to what extend formal descriptions can improve has been investigated [15]. Traditional application of formal methods assumes that the conversion into formal specification from requirements in natural language is one-way. On the contrary, our tool provides an easy way to support round trip development.

The concept of the domain requirements dictionary is close to the concept of the data dictionary used in the structured analysis method [3]. The data dictionary in structured analysis describes the structure of the data used in data flow diagrams, and it must contain definitions of all the data used.

Formal requirement dictionary is an extension of the requirement dictionary which is proposed in real time structured analysis [8] It is an extension of the requirements dictionary to contain a formal definition for each phrase of not only a data structure but functions or modules. The extension points are:

- Entry is not limited to a word but a key phrase.
- Semantics is defined by a formal language.
- The dictionary can be converted into a formal model.

3 Dictionary runs with Overture tool

3.1 Dictionary Management Tool

We had developed a formal requirement dictionary management tool which is named JODTool to assist formal specifiers. Current version of the tool supports the guideline to construct a model with VDM-SL [4] or VDM++ [13].

Key phrases are stored into a dictionary and the user can here complement formal information in the dictionary. It is implemented as an Eclipse plug-in whose perspective has a SRS Editor, a Formal Requirement Dictionary Editor, and an Entry Editor. We successfully integrated JODTool into the Overture tool and propose an evolutionary process to develop a formal and pre-formal specification over the environment.

Fig.1 shows interface of the tool. The tool provides 3 main functionalities:

Key phrase Registrar: The SRS Editor enables marking of all key phrases in natural language documents which are entry phrases of the formal requirement dictionary. As a result, the visibility of key phrases in the documents may be much improved for the specifiers as shown in Fig. 1.
The mouse selected regions are added to the specified dictionary as an entry phrase by one click or a short-cut command. Registration is performed easily on an arbitrary phrase.
The SRS Editor is able to handle plain text file, HTML files and a limited form of a Microsoft Excel file.

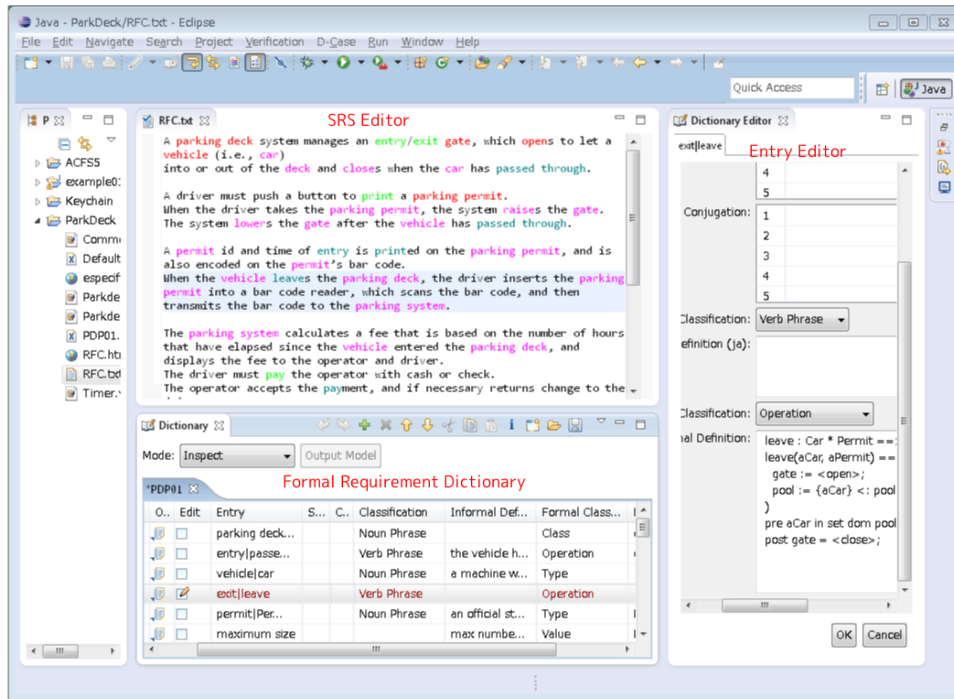

Fig. 1. The GUI of the JODTool

Entry Editor: The dictionary data can be edited through the entry editor view. The view provides basic editing operations such as copy-paste, search, sorting, for the entry, order of the entry, and dictionaries in the project, respectively.
An entry consists of a natural language part and a formal language part. The natural language part consists of a key phrase which is represented as a sequence of characters and an informal definition in a natural language. The tuple of key phrase and its part of speech gives key value in the formal requirement dictionary. The entry allows regular expressions of the entry phrases.
A key phrase can contains sub-key phrases which are compositions of the concept, synonyms, or paraphrases. It may also have conjunctions of the key phrase from the grammatical view. The dictionary provides a small structure to arrange "unstable" concepts in a natural language.
The formal language part consists of a section and an atomic portion of formal semantic definition. The entry keeps a bi-directional relation between descriptions in a natural language and a formal language.

Model Conversion: The tool supports translation from the formal definitions in the dictionary into a formal specification which can be checked by verification tool such as the Overture tool [12][3] or VDMTools [5][4].

[3] See http.//overturetool.org for details.

[4] See http://www.vdmtools.jp/en/ for details.

The current version of JODTool supports both VDM-SL and VDM++, so the formal specifier can use one of these languages to give a formal definition of the key phrases.

3.2 Relations among dictionaries

JODTool is equipped with functionality to support requirement analysis and enhanced traceability including handling of multiple dictionaries. Therefore, a dictionary is reusable not only the case of two specification sharing the same domain but also they can have overlapped parts.

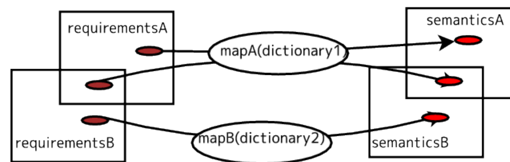

Fig. 2. Dictionary reuse

In the case where a term is defined differently in two directories it is possible to switch the dictionary as shown in Fig. 3.

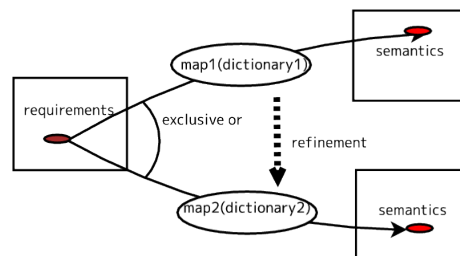

Fig. 3. Dictionary switching

Combined key phrase and sub-key phrases may share the same definition, and in such cases equivalent terms, or synonyms can be combined. The dictionary format is extended so that plural keywords can be registered into one entry. If you cannot correct the SRS directly, the dictionary can be used to resolve such ambiguities.

Variations of main key phrases are also stored in the same entry. Variations are supposed to be conjugations of verbs or declension of adjectives. The document marker check the main key phrase, sub key phrases, and all the conjugated forms as the same part.

The tool can simultaneously open multiple dictionaries corresponding to different domains, respectively. The dictionary is stored in an XML form and has a header and subsequent entries. The dictionary header has following fields:

Domain name: Domain name is an identifier of a domain and it is not limited to a problem domain, but can be arbitrary text strings.

Organization: Organization is an identifier of a project or an organization. A specification of the target depends on not only on its domain but its users and their application environment.

Input language: An input language is the descriptive language of non formal information in an entry of the dictionary. You can set multiple input languages in a dictionary.

Output language An output language is the descriptive language of the formal definition in an entry of the dictionary.

A specification process is not a one-way transition from an informal SRS to a formal model. The key phrases and sub key phrases in the dictionary are highlighted in the SRS editor on the fly as shown in Fig. 4. It becomes easier to provide feedback with corrections to the SRS, if the formal model is modified.

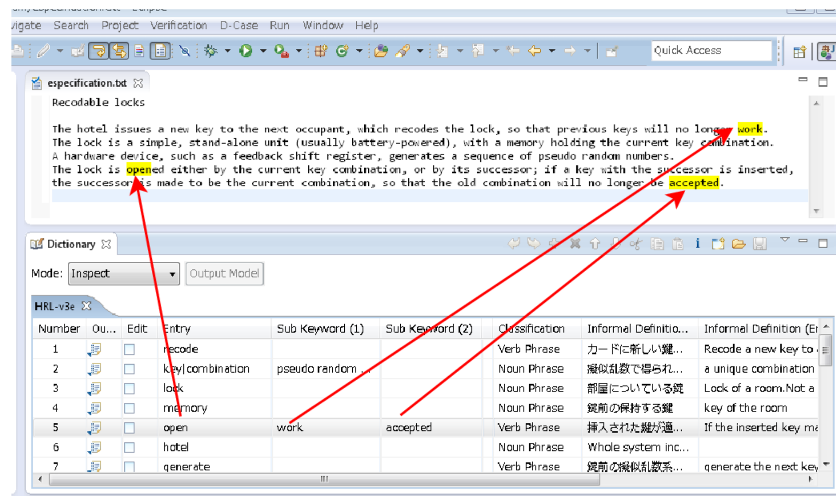

Fig. 4. Track back to SRS

3.3 Integration with the Overture tool

JODTool has been successfully integrated into the Overture tool, as shown in Fig. 1. We reorganized the namespace in the plug-in implementation, provided coexistence of the specification file and the dictionary generated by JODTool with projects of the Overture tool, delegation of VDM-SL and VDM ++ editor, etc.

The Overture tool is most popular and standard VDM integrated development environment built on Eclipse, which includes new features of the VDM languages. The integration enables formal specifiers to analyze by extracting key phrases of an SRS in natural language, to construct VDM specifications based on the set of key phrases, and to generate VDM source files from the dictionary. Overture can the perform syntax and type checking, and the formal specification and the pre-formal specification can be modified in a unified way. They can seamlessly continue the repetition to evolve the formal specification. It is possible to validate the specification by animation of explicit specifications [14] or by using proof obligations [1].

Defects of the specification found in the above process can be fed back to the SRS by identification using the reverse mapping of the dictionary. Though VDM modeling without the dictionary can point out defects in SRS, the formal requirement dictionary clarify the defects as concrete technical problems of the specification, rather than inability of the specifier. It demonstrates which part of SRS contains what type of problem because of the mathematical background of the formal method.

4 Repetitive Evolution of formal specification

A formal specifier cannot normally construct the proper formal model in one step because abstraction from a prioritised point of view and good understanding of the target system are necessary to apply a formal method properly. However, it might take time to understand the full functionality and the priorities among them in the target system. Specifiers have to communicate with the domain experts and other stakeholders, and begin the formal specification from most important part of requirements.

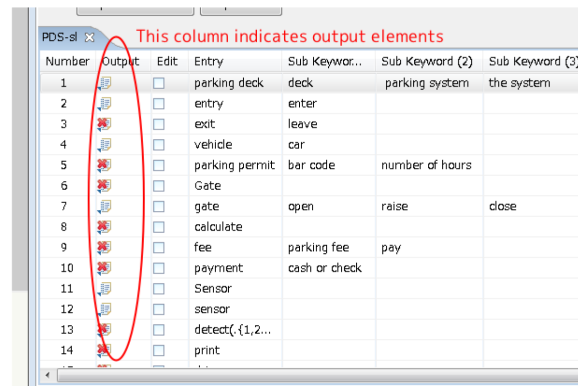

Fig. 5. Suppression of output

In the construction of a formal specification, the specifier describes and verify a partial formal model. JODTool supports such evolutionary modeling process. VDM originally provides some notations for transitional modeling, such as the "`is not`

yet specified" keyword. JODTool can control the output elements outside the model as shown in Fig. 5. Thus, the specifier do not hesitate to cut and error over the constructing model. Essential changes of requirements must act upon the model itself. When the verification is successful for the partial formal model, they can extend the model to incorporate other requirements.

We propose that it is possible to develop a formal specification the following procedure in this tool environment:

1. Extract all key phrases in the SRS.
2. Choose the high-priority elements and give them a formal definition.
3. Generate a partial model and verify it.
4. If any defects are discovered fed it back to the SRS.
5. Otherwise no problem is discovered, and the next high-priority elements can be selected to be incorporated in the model and go to step 3.
6. Finish the repetition if all key phrases are given a formal definition

Formal specifiers develop the specification by repeating this cycle. If eventually all key phrases are incorporated in the formal model, it can be said that reflection of whole the SRS into the model was completed. This does not mean the model is complete or the model is best one, but the specifier can explain objectively that the model contains all indispensable elements about the target system. It is practically important in application formal methods that clear indication of the termination condition of modeling process.

5 Case study

5.1 The Parking Deck System

We adopted a park deck problem as a case study of our method. A parking deck is a stacked garage, thus it has multiple parking space in it. The assignment is used in a course of George Mason University [7], as shown in Fig.6.

The VDM-SL model was composed of 84 lines excluding comments. No errors were discovered in the VDM model when it was checked by the Overture tool but that was expected. The main part of the specification is shown in Fig. 7.

The key concept of this model is that "Parking" is defined as a map from a car to a permit. The relation is quite simple and will not change through this whole system. Consequently a state component is defined of this type. However, this requires intuition of the specifier, because this definition stands on an abstraction of car and parking lot. From the viewpoint of enter/exit management of Park Deck, a car is a simple object and a permit is a tuple of an ID and its entry time.

All verbs in this example are corresponding to the operations. Thus, every move of this system changes its state, as they commonly do in embedded systems. The VDM-SL model was composed from the definitions of key phrases in the dictionary.

5.2 Application of evolutionary process

We applied the procedure in Section 4.

A parking deck system manages an entry/exit gate, which opens to let a vehicle (i.e., car) into or out of the deck and closes when the car has passed through. A driver must push a button to print a parking permit. When the driver takes the parking permit, the system raises the gate. The system lowers the gate after the vehicle has passed through. A permit id and time of entry is printed on the parking permit, and is also encoded on the permit's bar code.
When the vehicle leaves the parking deck, the driver inserts the parking permit into a bar code reader, which scans the bar code, and then transmits the bar code to the parking system. The parking system calculates a fee that is based on the number of hours that have elapsed since the vehicle entered the parking deck, and displays the fee to the operator and driver. The driver must pay the operator with cash or check. The operator accepts the payment, and if necessary returns change to the driver. Then, the operator enters a command to raise the gate to allow the vehicle to leave the parking deck.
You may assume that the system has the following external devices at the entry gate: a sensor to detect the presence of a car, a parking ticket printer to print the parking permit, an actuator to open and close the gate, and a sensor to detect that the car has departed.
You may assume that the system has the following external devices at the exit gate: a bar code scanner to read the permit bar code id, a display to show the parking fee to the operator and driver, an actuator to open and close the gate, and a sensor to detect that the car has departed.

Fig. 6. Park Deck Problem

1. **Extract all key phrases in the SRS**.
   We have extracted the important key phrases from the SRS using JODTool. According to the statistics of the tool, 8 noun phrase, 7 verb phrases, and 3 state variables were selected. There were many conjunctions of verbs and synonyms. For example, we identified "parking deck", "deck", "parking system", "the system" denoted the same concept.
2. **Choose the high-priority elements and give them formal definition**.
   We adopted a top-down approach to model the informal description of the system. Only the "ParkingDeck" module was generated in the first iteration. Then, the notion of "Parking" and its related state components and operations to the model was added in the second cycle. Afterwards other elements were added too and the VDM model was completed.
3. **Generate a partial model and verify it**.
   You can control the model output by the ability shown in Fig. 5. The Overture tool runs syntax and type checking process automatically, so the specifier can fix faults in the formal model in a short amount of time.
4. **If any defects are discovered fed these fixes back to the SRS**.
   We found some synonyms in the SRS so that they are stored in one entry, and put some key phrases together and built some structures at conceptual level. For example, key phrases "open", "close", "raise", "lower" were put into "gate" and its states.
   We also added a number of concepts that seemed to be missing. They came from the level of abstraction, thus most of them were type definitions which were distinct from the static state component.

```
values
MAX_CAR_NUMBER = 100;
RATE = 5;
car1 : Car = mk_token("car1");

types
Time = int;
Car = token;
Permit = Time * token;
Parking = map Car to Permit;
Gate = <open> | <close>;
Sensor = <on> | <off>;

state Store of
   pool : Parking
   gate : Gate
   sensor : Sensor
   paid : nat
   current_time : Time
    inv mk_Store(p, -, -, -, -) == 0 <= card dom p and card dom p < MAX_CAR_NUMBER
    init t == t = mk_Store({|->}, <close>, <off>, 0, 0)
end

operations
timestamp : () ==> Time
timestamp() == return current_time;

print_permit : () ==> Permit
print_permit() == def permit = mk_(timestamp(), mk_token(timestamp())) in return permit
 pre sensor = <on>
 post sensor = <on>;

approach : () ==> ()
approach() == sensor := <on>
pre gate = <close> and card dom pool <  MAX_CAR_NUMBER;

 entry : Car ==> ()
 entry(aCar) == gate := <open>; pool := pool munion {aCar |-> print_permit()}; gate := <close>; sensor := <off>
 pre sensor = <on>
 post sensor = <off> and gate = <close>;

 leave : Car * Permit ==> ()
 leave(aCar, aPermit) == (
   gate := <open>; pool := {aCar} <: pool 
 )
 pre aCar in set dom pool and calc_fee(aPermit) = paid
 post gate = <close>;

 pay : nat ==> ()
 pay (paid_fee) == (
   paid := paid_fee;
 );

 calc_fee : Permit ==> nat
 calc_fee(aPermit) ==
   def fee = (timestamp() - aPermit.#1) * RATE in return fee
 pre aPermit in set rng pool;
[EOF]
```

Fig. 7. Formal model using VDM-SL

If you do not wish to rewrite the SRS for some reason, the specifier can manage the semantics of each phrase manually.

5. **Otherwise no problem is detected in the model and the next level of priorities can be added to the model and go to step 3**.
   Repeat the cycle until the model achieve the mutable goal of verification or validation. Here, verification is the process to confirm elimination of inconsistency in each model and validation is the process to confirm elimination of inconsistency among the models of different models.
6. **Finish the repetition if all key phrases are given formal definition**.
   We eventually obtained 7 noun phrase, 5 verb phrases, 3 state components, and 9 others were picked up. Others includes type definitions, constants, and test data.

5.3 Alternate dictionaries

Pick up the billing of the example in sec. 5.1 as an example of refinement.

The entry of the formal domain dictionary is "calculates a fee", and its formal definition is Fig. 8.

```
calc_fee : Permit ==> nat
calc_fee(aPermit) == (
  def fee = (timestamp() - aPermit.#1) * RATE
  in return fee;
)
pre aPermit in set rng pool;

timestamp : () ==> Time
timestamp() == (
  return current_time
);
```

Fig. 8. Formal Billing Operation

Actual calculation part `val_fee` is cut out for this comparison. The problem is that the meaning of the sentence in natural language

a fee is calculated based on the number of elapsed hours

is ambiguous, so that various interpretations are allowed. For example, the fee calculation as Fig. 9,

If `val_fee` is defined as that, it is not obvious whether this satisfies the original specification. If formal definition of the part is Fig. 8, then Fig. 9 satisfies Fig. 8, because concrete specification is left to the latter phase. Fig. 10 satisfies Fig. 9 in the same way.

```
val_fee: Time ==> nat
val_fee (time) ==
  return time * RATE;
```

Fig. 9. Refined Formal Billing Operation(1)

```
val_fee: Time ==> nat
val_fee (time) ==
  if time <2 then return 1 else return 2 * (time -2) + 2;
```

Fig. 10. Refined Formal Billing Operation(2)

On the other hand, if formal definition of the part is Fig. 9, it can be proved that Fig. 10 does not satisfies Fig. 9, because any condition cannot satisfy Fig. 9 in Fig. 10. In this case, the specifier interpreted the part in natural language as the fee is proportional to the elapsed time.

6 Discussion

6.1 Bi-directional Mapping of Specification and Model

There have been many investigations about whether formal methods are useful for solving this kind of problems, and it is well-known as a way to reach improvement and assurance of software quality [6, 9, 21, 20].

The main difficulty of practicing formal methods is that mathematical knowledge and intuition is required, whereas native natural language can be used by most stakeholders without extra training. We propose the developing method using both specifications in a natural language and in a formal language. The stakeholders who know the formal method can understand the exact semantics, and those who are unacquainted with the mathematics can refer to the natural language description which semantics is captured by the formal method.

The duality of the formal domain dictionary helps co-existence of readability and rigidness of the specification.

Fig. 11 shows some roles of stakeholders.

Customer: The customer presents requirements to the specifier in natural language form.
Specifier: The specifier turns the requirements into a formal specification.
Developer: The developer refers to the formal specification and implements a corresponding program.

Formal specifications are desirable to avoid misunderstanding among stakeholders, but it depends on the familiarity about formal methods of each member.

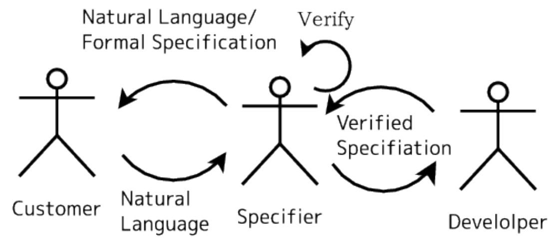

Fig. 11. Supposed Development Process

6.2 Evolutionary development

In the software development using formal methods, some steps of refinement are applied to the specification and finally this leads to executable code. (Horizontal direction from the left to the right in Fig. 12) If specification_n obeys specification_m, the relation is called refinement and it expresses with a $\sqsubseteq$ sign.

Another view is a formal model can be obtained from a corresponding specification. (Vertical direction from the upper to the lower in Fig. 12) The formal requirement dictionary gives the mapping between a specification and a model. If the dictionary changes, the model needs to be changed accordingly. Different dictionaries express different viewpoints of the target. Between the formal models sharing the same formal language, a.k.a sharing the same formal method, can be proved whether they have the refinement relation[5].

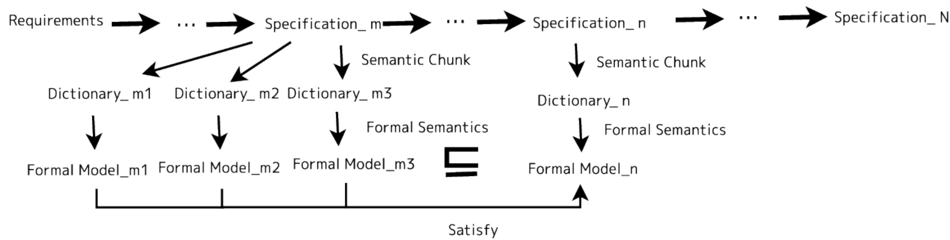

Fig. 12. Relation among specification and models

A formal requirement dictionary gives a mapping from the specification to the model, thus conditions for existence of such inverse mapping is that mapping is bidirectional. Therefore specifier can notice the necessity to keep consistency when either side is changed. Mapping must be surjective by definition, i.e. a formal definition can be given for any semantic chunk in natural language.

If mapping from the specification to the model is not injective, it is trivially not monotonic, where the same definition is given for multiple semantic chunks. To avoid

[5] When they are in different languages but sharing the same mathematical theory, this kind of relation can be apparently defined, but it is out of scope in this paper.

this case, a representative should be chosen from the group of chunks with the same formal definition, and use it in the inverse mapping.

The consistency of the specification in natural language and the specification in formal model and refinement relation between specifications is defined under the condition of bidirectional mapping from the specification to the model.

7 Conclusion and Future work

This paper introduces the formal requirement dictionary and its integration with the Overture tool. JODTool manages formal requirement dictionary and which runs with the Overture tool to verify and validate VDM models. The dictionary ensure traceability between SRS in natural language and its corresponding formal model. It supports specifier's task and relations among formal requirement dictionaries.

- In a case study, we found 18 key phrases in the SRS in the first cycle. But, there should be abstract types or constants and finally we had 24 entry in the formal requirement dictionary. This means that there is a difference between a sufficient model and a good model.
- The Overture tool provides quick syntax and type check, therefore the formal specification in VDM quickly achieve a stable model. If you need advanced verification or validation such as animation, the specifier can do that in the unified environment.
- A formal model with the tool is a good start point to begin refinement. Because formal requirement dictionary can ensure that all key phrases in the SRS are incorporated in the corresponding formal specification.

It is future work to perform examination on larger examples and to confirm reusability, considering the generic relation between natural language description and the formal specification.

Acknowledgment

Thanks to Dr. Hassan Gomaa for the publication of a Park Deck Problem as an example specification. Construction of the tool is supported by Mr. Yasuharu Yoshimura and others in Kyushu Business Corporation. This work was partially supported by MEXT Grant-in-Aid for Scientific Research(S) HBG4220001.

References

1. Aichernig, B.K., Larsen, P.G.: A Proof Obligation Generator for VDM-SL. In: Fitzgerald, J.S., Jones, C.B., Lucas, P. (eds.) FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997). Lecture Notes in Computer Science, vol. 1313, pp. 338–357. Springer-Verlag (September 1997), iSBN 3-540-63533-5
2. Daneva, M.: Erp requirements engineering practice: Lessons learned. IEEE Software 21(2), 26–33 (2004)

3. Demarco, T.: Structured Analysis and System Specification. Yordon Press (1981)
4. Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques for Software Developmen. Cambridge University Press, 2 edn. (2009)
5. 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)
6. Gerhart, S., Craigen, D., Ralston, T.: Experience with formal methods in critical systems. IEEE Software 11(1), 21–28 (1994)
7. Gomma, H.: Course assignments for software modeling and design. http://mason.gmu.edu/ hgomaa/assignments.html
8. Hatley, D.J., Pirbhai, I.A.: Strategies for Real-Time System Specification. Dorset House (1988)
9. Hinchey, M.G., Bowen, J.P. (eds.): Applications of Formal Methods. Prentice Hall (1995)
10. Jones, C.B.: Software development based on formal methods. In: Proceedings of the CRAI Workshop on Software Factories and Ada. LNCS, vol. 275, pp. 153–172. Springer-Verlag (1987)
11. Jones, C.B., Jackson, D., Wing, J.: Formal methods light. IEEE Computer 29, 20–22 (1996)
12. 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
13. Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M., Fitzgerald, J.: Validated Designs For Object-oriented Systems. Springer (2005)
14. 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
15. Meyer, B.: On formalism in specifications. IEEE Software 2(1), 6–26 (1985)
16. Omori, Y., Araki, K.: Tool support for domain analysis of the software specification in natural language. In: Proceedings of the IEEE TENCON 2010. pp. T7–3.3(CD–ROM) (2010)
17. Borland: Caliber. http://www.borland.com/products/caliber/
18. IBM: Rational DOORS. http://www-03.ibm.com/software/products/en/ratidoor
19. Thought Works: Mingle. http://www.thoughtworks.com/products/mingle-agile-project-management
20. Stefania Gnesi, T.M.: Formal Methods for Industrial Critical Systems: A Survey of Applications. Wiley-IEEE Computer Society (2012)
21. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys 41(4), 1–36 (October 2009)

VDM Animation for a Wider Range of Stakeholders

Tomohiro Oda[1], Yasuhiro Yamamoto[2], Kumiyo Nakakoji[1,3], Keijiro Araki[4], and Peter Gorm Larsen[5]

[1] Software Research Associates, Inc. (tomohiro@sra.co.jp)
[2] University of Tokyo (yxy@acm.org)
[3] Kyoto University (kumiyo@acm.org)
[4] Kyushu University (araki@csce.kyushu-u.ac.jp)
[5] Aarhus University, Department of Engineering, (pgl@eng.au.dk)

Abstract. Formal specification serves as reference to reliable definitions of concepts in a development. However, only a limited number of stakeholders are fluent in using formal specification languages. Animation is a promising technique to have stakeholders from different backgrounds understand what a formal specification means. This paper introduces three alternative tools for animation extended from VDMPad; Lively Walk-Through for the design of User Interfaces (UIs), Cloudly Walk-Through for presenting system's overview to non-engineering stakeholders, and Webly Walk-Through for Application Programmer Interfaces (APIs) for web applications.

1 Introduction

The number of lightweight applications of formal methods has increased in recent years [4, 18, 17]. The VDM dialects are some of the most frequently used specification languages suitable to lightweight use of formal specification [5]. This is also caused by VDM being supported by matured and well-maintained IDEs, namely the Overture tool [7] and VDMTools [2]. Different case studies report that animation plays important roles in both modeling and testing [11]. We have been working on expanding use of VDM interpreters [8, 12]. VDMPad is a simple web-based IDE for exploratory process of authoring VDM specification at the earlier stages of the formal specification phase [14, 15]. This paper introduces three new approaches that pioneer new dimensions of VDM animation. We believe that wider range of uses of specification animation over development phases will enhance the applicability of VDM.

After this introduction Section 2 motivates the new approaches by explaining how VDM modellers with advantage can collaborate with stakeholders that do not possess capabilities to use a formal notation. Afterwards Section 3 presents a small VDM-SL example for modelling a TV remote with zapping functionality. Section 4, 5 and 6 introduce our systems, Lively Walk-Through, Cloudly Walk-Through and Webly Walk-Through in that order. Afterwards, Section 7 describes related work. Finally, it is discussed how animation-based systems can enable collaboration between formal engineers and other stakeholders.

2 VDM Animation for Multidisciplinary Teams

A software development project is inherently multidisciplinary by roles and phases. Some stakeholders have fluency of formal specification languages while others do not. Animation instead can be one of the common languages that can be understood by people with different expertise.

Clients are typically influent of formal languages although their knowledge, experience and visions are necessary to produce a right software product. One possible solution is that a few client representatives acquire basic reading skills of a formal notation [1]. Otherwise formal engineers explain the formal specification and its implications to customers in a natural language and the unambiguous nature of formal specification does not directly benefit the clients.

The gap of fluency also exist with software developers that have not been trained in understanding formal specifications. Programmers, who read formal specification and implement it, and test engineers, who read formal specification and conduct software testing, are familiar with formal grammars through programming languages. Some of the remaining stakeholders, such as sales representative, User Interface (UI) designers and documention writers, may have difficulty to read formal expressions regardless of the fact that they can benefit from clear and rigorous representation of the formal specification.

A formal specification can play a major role in software development as a reference to reliable definitions of key concepts. Formal specifications have two levels of abstraction: the application domain and the language. While the abstraction of the language may unfortunately bring the gap of fluency into the team, the abstraction of the application domain can be and should be understood by most stakeholders. Animation is a technique that visualizes the abstraction of the application domain. Formal specification enables many techniques such as proofs and automated generations. Among them, animation is one particular technique that can be understood by a wider range of stakeholders.

3 Example Specification: TV remote with zapping function

The specification below is an executable VDM-SL specification for a TV remote controller with zapping support.

A user has a series of favourite TV stations for channel-zapping. The user can manage a channel list for zapping (called a zap list below) by adding or deleting the current channel to or from the zap list. The user can start zapping by startZap, and traverse through the zap list back and forth by pressing `prevZap` or `nextZap`.

This specification will be used as an example throughout this paper to exhibit different usages of animation capabilities.

```
types
channel = nat
  inv channel == channel >= 1 and channel <= 12
state memory of
```

```
  current : channel
  zapList : seq of channel
  zapIndex : [nat1]
init
  s == s = mk_memory(1, [], nil)
end
operations
  num : channel ==> channel
  num(x) == (current := x; return current);
  inc : () ==> channel
  inc() == return num((current mod 12) + 1);
  dec : () ==> channel
  dec() == return num(((current - 2) mod 12) + 1);
  get : () ==> channel
  get() == return current;
  setZapIndex : int ==> channel
  setZapIndex(x) ==
        (if 0 < x and x <= len zapList
        then (zapIndex := x; num(zapList(x)))
        else zapIndex := nil;
        return current);
  startZap : () ==> channel
  startZap() == return setZapIndex(1);
  nextZap : () ==> channel
  nextZap() ==
    return  if zapIndex <> nil
                 then setZapIndex(zapIndex+1)
                 else current;
  prevZap : () ==> channel
  prevZap() ==
    return if zapIndex <> nil
                then setZapIndex(zapIndex-1)
                else current;
  addZap : () ==> ()
  addZap() ==
    if current in set elems zapList
    then skip
    else zapList := zapList ^ [current];
  delZap : () ==> ()
  delZap() ==
    zapList :=
      [zapList(index)
      | index in set inds zapList
      & zapList(index) <> current];
  getZapList : () ==> seq of channel
  getZapList() == return zapList;
```

4 Lively Walk-Through: Animation for UI Designers

Collaboration between formal engineers and UI designers is critical to develop a usable interactive system [10]. UI design artifacts are essential and critical components of interactive systems. Some design decisions may affect or be influenced by functional models. A mismatch between user's cognition and the system's functionality results in a failure of performing the user's task at hand. The output of UI design is not merely a collection of graphical sketches, but also includes design decisions made by drawing those sketches. Implementation details such as icon pixels are to be sent to other professions including graphic designers, and design decisions such as geometric constraints among GUI elements are externalised as design artefacts. UI designers take domain knowledge, functional models and sometimes ethnographic research to model interactions between the system and users into account.

It is sometimes difficult to share a common understanding of the system between formal engineers and UI designers because they have different expertise. VDM animation combined with a GUI prototyping tool is a powerful vehicle for both the formal engineers and the UI designers to understand the intended functionality of a software system.

UI prototyping using an executable formal specification benefits formal engineers because it involves validation of functional models in terms of user interactions, and also UI designers because animation of formal specification gives intuitive and reasonable understanding on the intended functionality of the system. A benefit of the formal specification for UI designers is a medium to describe constraints to functional models by UI design. A rapid prototype in a programming language may also convey the system's functions, but often lacks language constructs to distinguish design decisions from implementation details. Formal specification languages including the VDM family have functionality to separate design decisions as assertions from implementation details in animation mechanisms.

Lively Walk-Through is a medium for discussion between formal engineers and UI designers. A UI prototype on Lively Walk-Through is built with (1) a VDM specification, (2) a UI sketch, (3) UI widgets, (4) LiveTalk scripts and (5) binding between UI events and LiveTalk scripts. Using the resulting UI prototype, formal specification engineers and UI designers walk-through scenarios. Lively Walk-Through records all UI events, operation calls and states into a history. The formal engineers and UI designers discuss and make agreements based on the VDM specification, UI sketches and snippets from the history. The objective of Lively Walk-Through is to create agreements between formal engineers and UI designers. UI designers do not have to understand the VDM model as such. The animation exhibits the behaviour of the specification by both the formal language and the designers' language, and the formal engineers and the UI designers together critique it pointing at the same artefacts.

Figure 1 is a screenshot of a UI prototype on Lively Walk-Through. The UI prototype was constructed by the following steps:

- Wrote the VDM specification shown in Section 3.
- Drew a UI sketch on a paper, took a photo of it, and placed the image on Lively Walk-Through.

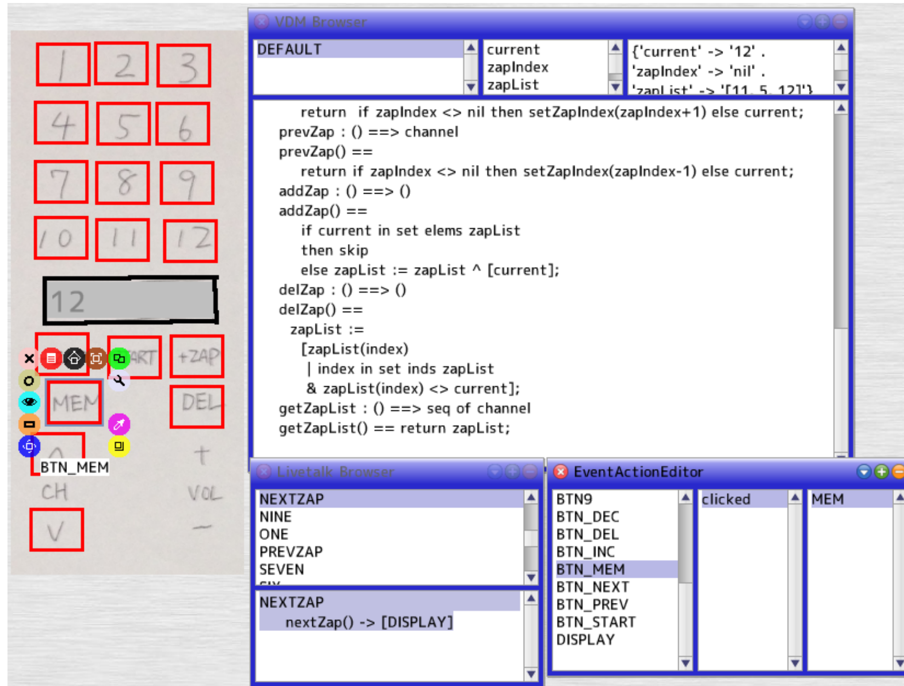

Fig. 1. UI prototyping for a TV remote controller with zapping support

```
NEXTZAP
nextZap() -> [DISPLAY]
```

Fig. 2. A LiveTalk Script for the “next zap” button

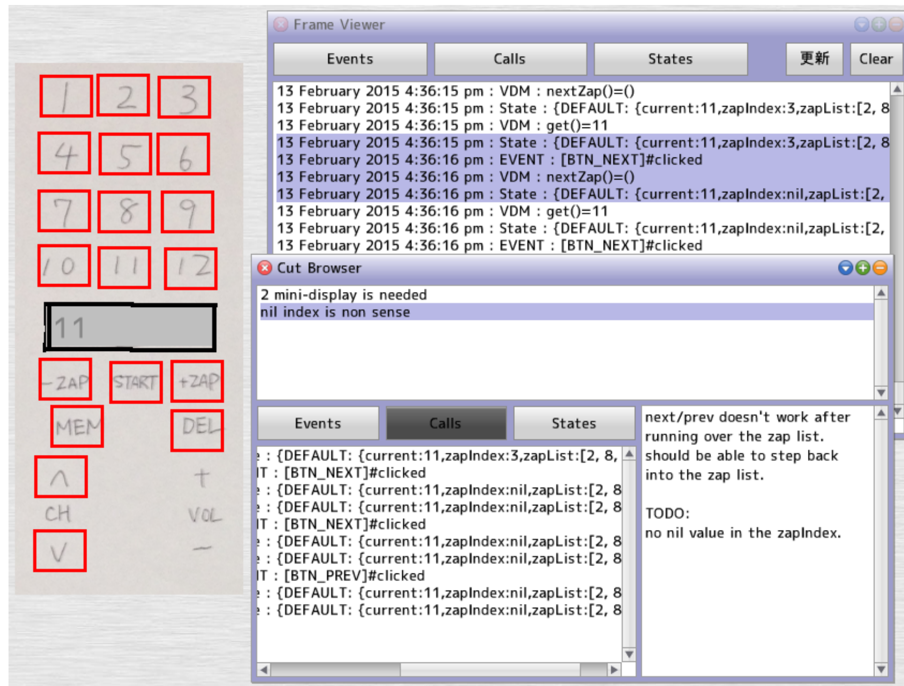

Fig. 3. Discussion between UI designer and VDM engineer

- Placed sensor rectangles (the red boxes in Figure 1) on the buttons drawn.
- Wrote actions for buttons in LiveTalk scripts (see Figure 2).
- Associated *click* event of each button to an action.

LiveTalk is a Domain Specific Language (DSL) that bridges VDM animation and UI widgets. Figure 2 shows the definition of the action of the "next zap" button. The first line is the name of the script. The second line states that the return value of the operation call "`nextZap()`" is passed to the `DISPLAY` widget. The `DISPLAY` widget will show the resulting channel. Other buttons have their actions defined in LiveTalk.

Actions defined in LiveTalk are associated with GUI's events. When a button is clicked, the button triggers a `clicked` event. The system then executes actions associated with the event. Events can be concurrently triggered, but the execution of VDM-SL operations are mutualy excluded and thus serialized by a semaphore.

Figure 3 illustrates a discussion using Lively Walk-Through. After a series of walk-through's, the UI designer points out that the zap mode is exposed to the user which leads to a confusing interaction. The UI designer also claims that the user does not need the `prevZap()` operation, but the `startZap()` and `nextZap()` should be unified into one operation for simplicity. The VDM engineer agrees and put it into the `TODO` field.

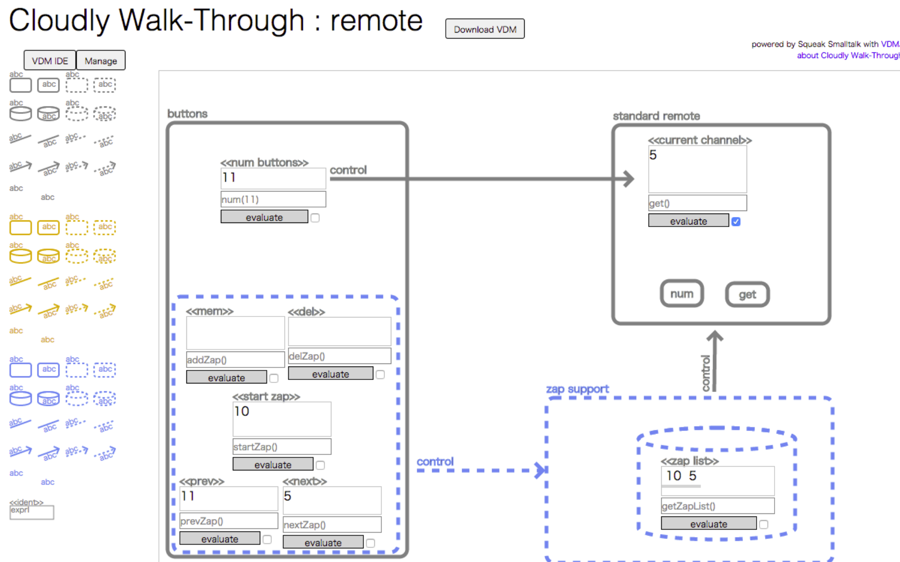

Fig. 4. Diagram with Animation on Cloudly Walk-Through shared on the web

5 Cloudly Walk-Through: Animation for Non-Engineering Stakeholders

Cloudly Walk-Through is a general diagrams editor with VDM-SL animation capabilities. The basic idea behind Cloudly Walk-Through is to explain formal specifications by supplementing them with two kinds of visual notations; diagrams and visual representation of VDM-SL values.

Some stakeholders without software engineering backgrounds, such as user representatives and marketing representatives, often have domain knowledge crucial to produce software systems that fit the application domain. Diagrams and natural language texts are often used to explain architectural designs and important design decisions to such non-engineering stakeholders. Those informal notations have advantages of less technical demands at the cost of semantic ambiguity and uncertain implications. The goal of Cloudly Walk-Through is to take advantages of both the easy-to-understand diagrams and the rigorous formal specification.

Figure 4 is a screenshot of Cloudly Walk-Through animating the remote controller with the zapping support introduced in Section 3. Gray rounded rectangles represent physical components (buttons and a controller module) of a conventional remote controller. Blue dotted rectangles are extension components (zap buttons and zap controller) for the zapping support. Small assemblies in rectangles are mini-evaluators. A VDM engineer enters a VDM expression and Cloudly Walk-Through evaluates it when the button below is pressed. The result value is rendered above in a visual representation with a larger font. The visual representation lowers the technical barrier for non-engineering stakeholders to comprehend VDM-SL's expressions.

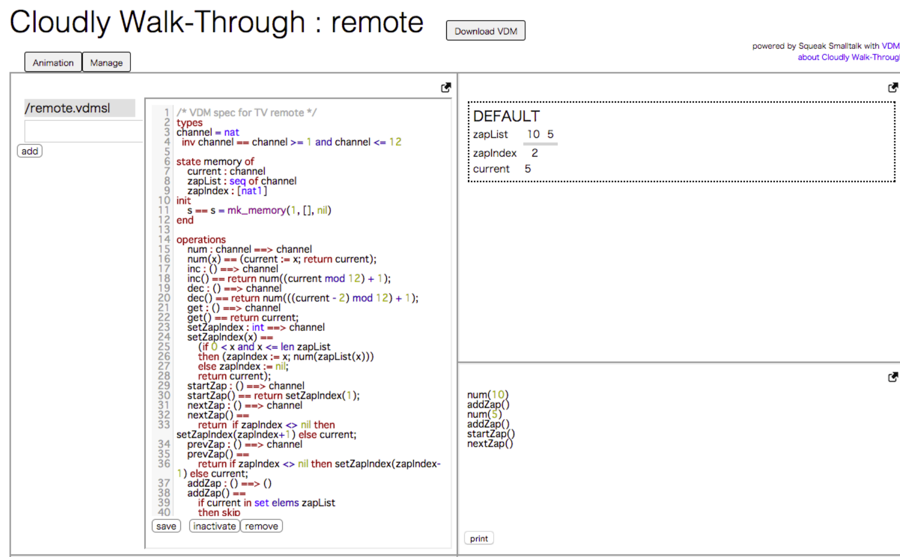

Fig. 5. VDM-SL IDE of the Cloudly Walk-Through

The animation is shared on the web. Cloudly Walk-Through manages registered users' permissions to watch or manage each animation instance. Any registered user who as the permission to watch the animation can see the on-going animation by opening the URL of the animation instance.

Figure 5 shows the VDM-SL IDE on Cloudly Walk-Through for formal specification enginneers. Formal engineering users can manage the source tree of the VDM-SL specification on the left side of the IDE. The right-top area displays the state of the on-going animation. Diagram representation of VDMPad [15] is used to render the value of each state variable. Values are grouped by dashed rectangles using modules. The top right area is a text editor called a Workspace, where the user can write and evaluate VDM expressions. Those components are similar to their counterparts in VDMPad. The difference is that the specification is stored in a source file tree on the server, and the animation state is shared by all users. In addition, Cloudly Walk-Through has a git interface to manage the source tree by git to push, pull, commit, revert and view log and status. Formal engineers can synchronize source trees with git repository to use Cloudly Walk-Through in conjunction with other formal methods tools such as the Overture tool and VDMTools. Users can also download a zip archive of the source tree by pressing a "Download ZIP" button.

6 Webly Walk-Through: Animation for Client Programs

Webly Walk-Through is a prototype Web API server where client programs can be used as an alternative until the implementation of Web API server will be completed.

Rapid development is often critical to web application services. A web API is the interface that separates the server side from the client side. The server and the client are sometimes developed by different teams in parallel. The specification of a web API is the key to successful development of web applications. Rigorous specifications can help the development of the client side as well as the server side. In conventional development, the client team typically implements stubs to test their program code and the user interface design. Webly Walk-Through serves as a Web API server by animating a VDM-SL specification of the web API. Webly Walk-Through uses JSON to transfer data between the web API server and its clients. JSON is a widely used open standard format and encoders and decoders are available in many programming languages.

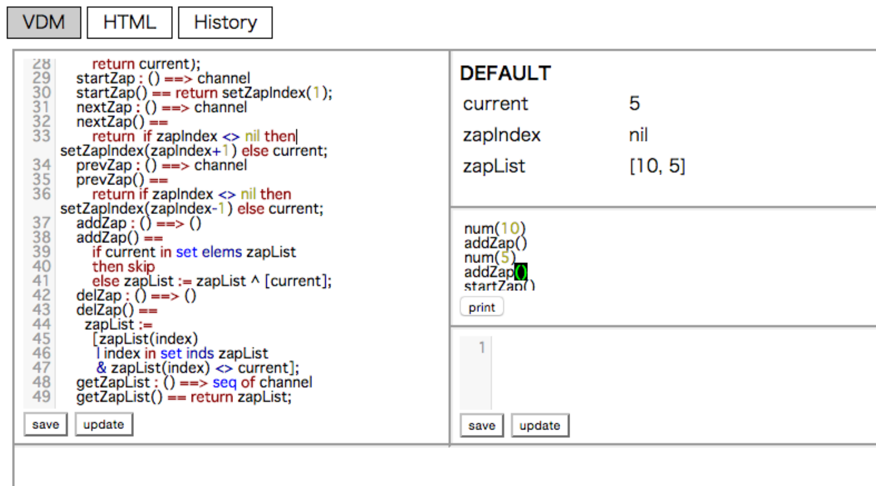

Fig. 6. VDM IDE on Webly Walk-Through

Webly Walk-Through provides three kinds of web services: (1) Web IDE for VDM-SL and web contents, (2) Web API, and (3) files with static contents. Figure 6 is a screenshot of the Web IDE. The Web IDE has three tabs; VDM, HTML and History. In the VDM tab, formal engineers can write a VDM-SL specification of the Web API. The state of the on-going animation is shown in the right-top area. The right-middle is a Workspace that can be used to evaluate arbitrary VDM expressions. Translation rules between VDM values and the corresponding JSON data are defined in the right-bottom area. In this particular example, channel numbers are the only data transfered between the server and clients, and no translation rule is needed.

Webly Walk-Through also provides support for files with static contents. The HTML tab provides a web-based static content editor to manage and modify the files. Figure 7 is a screenshot of the static content editor.

Fig. 7. File Editor on Webly Walk-Through

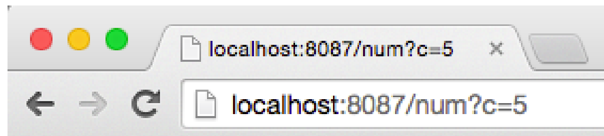

Fig. 8. The `num` operation automatically published as a Web API at http://localhost:8087/num

The Web API service is driven by VDM-SL animation. A Webly Walk-Through server publishes operations and functions at URLs `http://<hostname>:<port>/<module name>/<operation/function name>` for a modularized specification, or `http://<hostname>:<port>/<operation/function name>` for a flat specification. The API call may be requested as a Get method with a query part or post method with the form-data format. When the server receives an API call with parameters, each parameter in a JSON format is translated into a VDM value according to user-defined translation patterns. The server then evaluates the operation or function call with the translated arguments. The resulting value is translated into a JSON format and sent to the client program. Figure 8 is a screenshot of a web browser dispatching an API. The argument to the `num` operation is `5`, which is identical both in JSON and VDM-SL. The operation `num(5)` is then evaluated and the return value `5` is sent back to the web browser in JSON format, which is again identical to its VDM-SL counterpart.

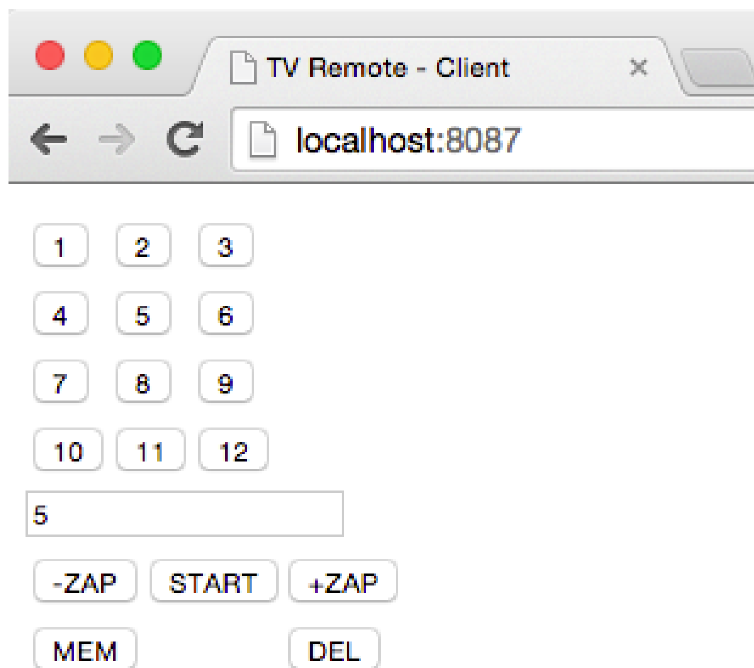

Fig. 9. The client UI that uses the Web API animated by Webly Walk-Through

The Webly Walk-Through server provides web pages with static contents such as html, css and javascript files. Figure 9 shows a screenshot of the UI page for the remote

controller. The page has a JavaScript code that invokes Web API's in the specification, such as num, startZap, nextZap and so on.

The API calls received by the server are recorded along with translated arguments and return values into the history page shown in Figure 10. VDM expressions evaluated in the IDE page are also recorded.

Webly Walk-Through

powered by Squeak Smalltalk with VDMJ
about Webly Walk-Through

VDM | HTML | History

```
1
2  [VDM] addZap()==>()
3  [API]DEFAULT`num(5=>5) ==>5=>5
4  [API]DEFAULT`addZap() ==>()=>
5  [API]DEFAULT`num(10=>10) ==>10=>10
6  [API]DEFAULT`addZap() ==>()=>
7  [API]DEFAULT`startZap() ==>1=>1
8  [API]DEFAULT`nextZap() ==>5=>5
9  [API]DEFAULT`prevZap() ==>1=>1
10 [API]DEFAULT`num(1=>1) ==>1=>1
11 [VDM] getZapList()==>[1, 5, 10]
12 [API]DEFAULT`getZapList() ==>[1, 5, 10]=>[1,5,10]
```

clear

Fig. 10. Webly Walk-Through IDE shows the access log on the server with states and expressions of the VDM spec

7 Related Work

VDMPad [14, 15] is a simple Web IDE for VDM-SL. VDMPad was designed to be lightweight authoring environment for VDM-SL engineers at the earlier stages of the specification phase. The user writes, tests and debugs a specification by animating the specification. Lively Walk-Through, Cloudly Walk-Through and Webly Walk-Through are implemented based on VDMPad. Although the three Walk-Through systems provide specification editors, the systems put more values on the animation part.

PVSio-web [16] is a UI prototyping tool to animate executable formal specification generated from a graphical diagram notation called Emucharts. A VDM-SL generator from Emucharts was recently developed [9]. PVSio-web aims at development of safety-critical systems, such as medical devices, with precise UI design. Lively Walk-Through assumes more sketchy UI prototypes at the earlier stages of the development. PVSio-web, Lively Walk-Through and Cloudly Walk-Through share a common objective, that is, to engage stakeholders other than formal engineers with lightweight formal methods.

BMotion Studio [6] is a graphical animation builder aiming at communications between developers and domain experts. It enables to build a graphical representation using labels, images and buttons that drives an Event-B specification.

The Remote Control Interface of VDMJ enables Java program to control models in VDM [12]. The resulting GUI can be used by non-engineering stakeholders. The Java code needs to be written by engineers who understand both VDM and Java. Overture VDMPP GUI automatically generates a simple UI to display state variables and push buttons that invoke operations and functions without any GUI programming effort.

The Crescendo tool [3] is a co-simulation environment that bridges between physical model simulations and executable formal specifications written in VDM-RT. The Crescendo tool has 3D animator of Continuous-Time simulation model. The users of the Crescendo tool can be considered multidisciplinary in the sense that Continuous-Time models are based on physical systems including differential equations while Discrete-Event models are typically of computer domain expressed in VDM.

DisCo [13] is a simulator that combines a logical event level in formal specification and graphical animation for agile development of games.

8 Discussion

Formal specifications serve as reliable references in the lightweight use of formal methods. Formal specifications can be a strong tool to organise domain knowledge from domain experts. Animation has a potential to enable collaboration for a diversity of stakeholders who participate in development and use of formal specification when combined with visualisations and programming languages. Figure 11 illustrates the relations between the different kinds of animation systems and expertise in software development.

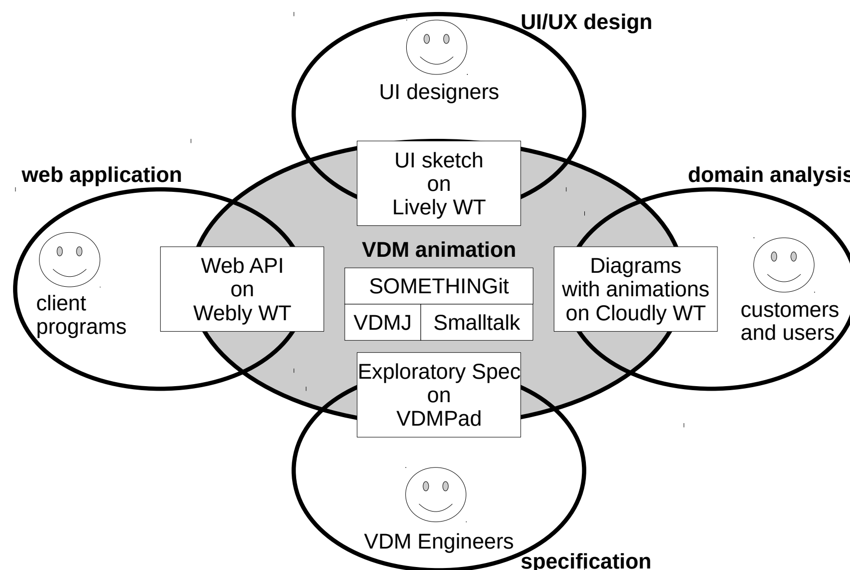

Fig. 11. Animation systems and expertise in software development

Animation with a graphical presentation is the key to having non-engineering stakeholders participating in the development and use of formal specification. Lively Walk-Through combines executable specifications with UI designs. UI designers may not have formal engineering skills, but can interact with executable formal specifications through animated UI prototypes. Cloudly Walk-Through visualises execution of specifications to have domain experts review formal specifications. Domain experts in this context typically do not have formal engineering skills. Graphical presentations used in the animation systems include casual diagrams, domain-specific diagrams, sketchy UI designs and precise UI prototypes. One possible direction for future work is a collaboration with massive data visualisation/analysis techniques such as plots, graphs, geometries and volume data.

An interface with programming languages is the key to expanding use of specification animation to wider scenes of systems development. Webly Walk-Through combines formal specifications with implementation code in order to build rapid prototypes. SOMETHINGit, a foundation library of Lively Walk-Through, provides ability to evaluate VDM expression as a part of Smalltalk programs. These systems assume that users have engineering skills and their tasks are thus more useful for software engineers. One possible future work direction is to embed a formal specification and its interpreter into runtime code so that the system can be debugged referring to the specification with regard to the suspicious execution context at hand.

9 Concluding Remarks

In this paper, three animation systems, Lively Walk-Through, Cloudly Walk-Through and Webly Walk-Through have been introduced as tools to extend the reach of formal specification towards stakeholders other than formal engineers. They are all derived from the sources of VDMPad but have different objectives and target users. As systems development requires cross-disciplinary collaborations, a media for communication between formal engineers and other stakeholders is needed. The three systems are our attempt to design such communication media using animation mechanism of VDM-SL.

The three systems have not yet been used in practical developments. However, we hope to have opportunities to apply them in industrial projects in the future. We also plan to elaborate further on usage of animation in additional phases of software development.

Acknowledgments This work is supported by Grant-in-Aid for Scientific Research (S) 24220001.

References

1. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer Enclave Protection Software. In: Proceedings of the 1st IEEE International Symposium on Secure Software Engineering (ISSSE'06). IEEE (March 2006)
2. 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)

3. Fitzgerald, J., Larsen, P.G., Verhoef, M. (eds.): Collaborative Design for Embedded Systems – Co-modelling and Co-simulation. Springer (2014), http://link.springer.com/book/10.1007/978-3-642-54118-6
4. Gilliam, D., Powell, J., Bishop, M.: Application of Lightweight Formal Methods to Software Security. In: Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise (WETICE'05). IEEE (2005)
5. Kurita, T., Nakatsugawa, Y.: The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. Intl. Journal of Software and Informatics 3(2-3), 343–355 (October 2009)
6. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B Models with B-Motion Studio. In: Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems. pp. 202–204. Springer-Verlag (November 2009)
7. 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
8. 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
9. Masci, P., Couto, L.D., Larsen, P.G., Curzon, P.: Integrating the PVSio-web modelling and prototyping environment with Overture. In: Ishikawa, F., Larsen, P.G. (eds.) Submitted to the 13th Overture Workshop. Oslo, Norway (June 2015)
10. Nakakoji, K., Yamamoto, Y.: chap. Conjectures on How Designers Interact with Representations in the Early Stages of Software Design. Chapman & Hall (2013)
11. Nielsen, C.B.: Dynamic Reconfiguration of Distributed Systems in VDM-RT. Master's thesis, Aarhus University (December 2010)
12. Nielsen, C.B., Lausdahl, K., Larsen, P.G.: Combining VDM with Executable Code. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) Abstract State Machines, Alloy, B, VDM, and Z. Lecture Notes in Computer Science, vol. 7316, pp. 266–279. Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-30885-7_19, ISBN 978-3-642-30884-0
13. Nummenmaa, T., Tiensuu, A., Berki, E., Mikkonen, T., Kuittinen, J., Kultima, A.: Supporting agile development by facilitating natural user interaction with executable formal specifications. SIGSOFT Softw. Eng. Notes 36(4), 1–10 (Aug 2011), http://doi.acm.org/10.1145/1988997.2003643
14. Oda, T., Araki, K.: Overview of VDMPad: An Interactive Tool for Formal Specification with VDM. In: International Conference on Advanced Software Engineering and Information Systems (ICASEIS) 2013 (Nov 2013)
15. Oda, T., Araki, K., Larsen, P.G.: VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In: Plat, N., Gnesi, S. (eds.) FormliSE 2015. In connection with ICSE 2015, Florence (May 2015)
16. Oladimeji, P., Masci, P., Curzon, P., Thimbleby, H.: PVSio-web: a tool for rapid prototyping device user interfaces in PVS. In: FMIS2013, 5th International Workshop on Formal Methods for Interactive Systems, London, UK, June 24, 2013 (2013)
17. Ubayashi, N., Nakajima, S., Hirayama, M.: Context-dependent product line engineering with lightweight formal approaches. Science of Computer Programming (2012)
18. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys 41(4), 1–36 (October 2009)

Integrating the PVSio-web modelling and prototyping environment with Overture

Paolo Masci[1,*], Luis Diogo Couto[2], Peter Gorm Larsen[2], and Paul Curzon[1]

[1] School of Electronic Engineering and Computer Science
Queen Mary University of London, United Kingdom
{p.m.masci,p.curzon}@qmul.ac.uk

[2] Department of Engineering
Aarhus University, Denmark
{ldc,pgl}@eng.au.dk

Abstract. Tools are needed that overcome the barriers preventing development teams using formal verification technologies. We present our work integrating PVSio-web with the Overture development and analysis environment for VDM. PVSio-web is a graphical environment for modelling and prototyping interactive systems. Prototypes developed within PVSio-web can closely resemble the visual appearance and behaviour of a real system. The behaviour of the prototypes is entirely driven by executable formal models. These formal models can be generated automatically from Emucharts, graphical diagrams based on the Statechart notation. Emucharts conveniently hides aspects of the formal syntax that create barriers for developers and domain experts who are new to formal methods. Here, we present the implementation of a VDM-SL model generator for Emucharts. An example is presented based on a medical device. It demonstrates the benefits of using Emucharts to develop a formal model, how PVSio-web can be used to perform lightweight formal analysis, and how the developed VDM-SL model generator can be used to produce a model that can be further analysed within Overture.

Keywords: Prototyping, VDM-SL, PVSio-web.

1 Introduction

Formal verification technologies can help developers to discover design problems early in the development process of safety critical systems. These technologies, however, usually require significant mathematical sophistication, and many developers perceive this as a barrier that overweighs the advantages of using such tools.

PVSio-web [1,2] is a new research tool developed to ease the use of formal methods technologies when developing safety-critical *interactive* systems, i.e., ones that involve interaction between devices and human users. It provides a graphical environment that allows developers to rapidly generate interactive prototypes resembling the visual appearance and behaviour of the real system (see Figure 1). Underneath the interface, the

* Corresponding author.

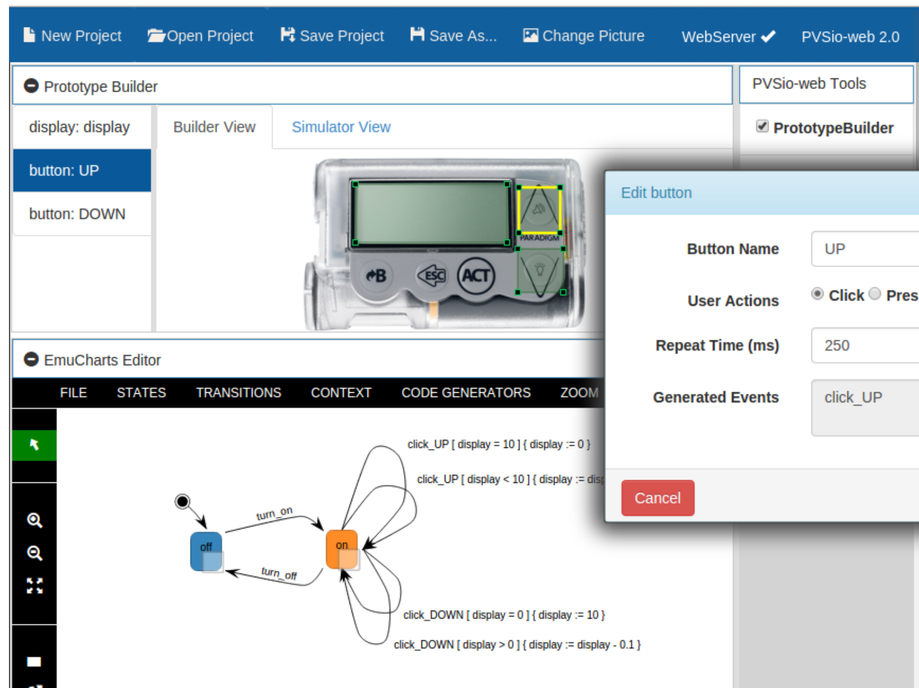

Fig. 1: Screenshot of the PVSio-web graphical environment while creating an interactive prototype based on a formal model.

tool uses advanced formal methods technologies for modelling and analysis. PVSio-web has been successfully used to demonstrate previously undetected design flaws in medical devices [3], and to clarify the causal relationships between user interface issues and software defects [4].

In its current implementation, PVSio-web builds on the PVS [5] theorem proving system for modelling and analysis. However, the architecture of PVSio-web is general, and allows one to link the environment with other formal methods tools.

We report on our work on integrating PVSio-web [1] with the Overture [6] development and analysis environment for VDM. This benefits both PVSio-web and Overture users. PVSio-web users gain direct access to an extensive set of tool features and case studies developed by the VDM community. Overture users gain the modelling and prototyping functionalities offered by PVSio-web, which enable: *validation of formal models* with domain specialists before starting a verification process; *demonstration of formal analysis results* to domain specialists in a way that is easy to comprehend; *lightweight formal analysis* of user interfaces based on user-centered design methods. Here, we focus on the integration of a core PVSio-web tool, the Emucharts editor, with Overture. Using the Emucharts editor, developers can specify the behaviour of a PVSio-web prototype using graphical diagrams, and automatically generate executable PVS models from these diagrams. We have successfully extended the Emuchars editor

to enable generation of VDM models. This basic integration already opens several exciting options, including automatic translation of VDM models from/to PVS, as well as means to explore the behaviour of VDM models using PVSio-web prototypes. The contributions are:

- A new PVSio-web extension for generating executable VDM specification language (VDM-SL) models.
- An example application based on a medical device. A formal model of the device is specified using a graphical Emucharts diagram that hides aspects of the formal syntax; then, a device prototype based on the Emucharts is generated that enables lightweight formal analysis; finally, a VDM-SL model is generated from the same Emucharts, enabling full formal analysis in Overture.

Related Work. VDMPad [7, 8] is a web-based integrated development environment for developing VDM-SL models. The tool provides: a textual model editor for viewing and editing models; a model animator for model debugging and testing. The tool supports the exploratory development of formal models, allowing lightweight formal analysis and permissive checking. In contrast to VDMPad, PVSio-web is specifically designed for modelling and analysis of interactive (human-computer) systems. Our tool thus offers functionalities for generating realistic interactive prototypes that can closely resemble the visual aspect and behaviour of a real system. The behaviour of these prototypes is based on formal models executed within the PVSio [9] animation environment of PVS. SCR [10] and B-Motion Studio [11] are also related work in that both tools provide a way to obtain graphical prototypes from formal models. Using SCR, one can formally specify the behaviour of a system, use visual front-ends for demonstrating the system behaviour based on the specifications, and use a group of formal methods tools for the analysis of system properties. With B-Motion Studio, one can create simple graphical visualisations based on Event-B models. SCR and B-Motion Studio are, however, not integrated with Overture. In addition, these tools lack specialised functionalities needed for the analysis of user interfaces (e.g., deployment of prototypes on mobile devices, and logging of user interactions).

Organisation. The remainder of the paper is organised as follows. We first overview the PVSio-web Emucharts Editor in Section 2. The core of the paper, illustrating a VDM-SL model generator for Emucharts, is presented in Section 3. We then give, in Section 4, a small example with a medical device. The example illustrates how Emucharts can be conveniently used to develop a formal model of the data entry system of the device (in subsection 4.2), how PVSio-web supports lightweight formal analysis (in subsection 4.3), and how the developed VDM-SL model generator can be used to produce an executable VDM-SL model that can be further analysed within Overture (in subsection 4.5). Finally, Section 5 provides a number of concluding remarks and indicates the future plans with this work.

2 Emucharts Editor

The PVSio-web Emucharts Editor is a tool for developing models of interactive systems. Models are specified using graphical diagrams called Emucharts, based on Statecharts [12]. Figure 1 shows a snapshot of the Emucharts Editor in use developing a

diagram specifying the behaviour of a medical device. Using the Emucharts Editor, developers can:

- Draw labelled boxes representing *states* of the system. State labels are strings representing the names of the different modes of the modelled system.
- Draw labelled arrows representing *transitions* between states. Transition labels are in the form `t [ cond ]` { `actions` }. The transition name (`t`) is a symbolic constant identifying the name of the modelled event. The transition condition (`cond`) is a Boolean expression defining the circumstances under which the transition is taken. The transition actions (`actions`) are expressions defining how the system state changes when the transition is taken.
- Define *variables* representing the structure of the system state. State variables are tuples: *(name, type, value, v0)*. Variable names are unique. Variable types can be basic types (e.g., **`bool`**, **`int`**, **`real`**), or user-defined types (e.g., *records*, *lists*). As in modern programming languages, a variable's value can be retrieved by referencing the variable name, and can be updated using assignment expressions (the assignment operator is :=). Each variable has an initial value, *v0*, given as the last element of the tuple.

In the Emucharts Editor, a virtual palette provides the essential elements for drawing the diagram (i.e., boxes and arrows), as well as tools for editing labels of diagram elements, and erasing elements from the diagram. Variables, constants, and functions are declared in a table called *context*, separately from the graphical diagram.

The Emucharts Editor was developed using the Model-View-Controller [13] design pattern, which creates a clear separation between the graphical front-end of the tool, and the logic for generating formal models. The editor, in fact, has two main components. The first is a Visual Editor, which handles both interactions with the user when drawing a diagram, and the look-and-feel of the graphical elements of the diagram. The second element is a Model Generator, which allows developers to translate visual diagrams into formal models.

In this work, we extend the Model Generator, and introduce a new module for producing executable VDM-SL models that can be imported and analysed within Overture.

3 The VDM-SL Model Generator

Our VDM-SL model generator is for Emucharts diagrams representing deterministic event-driven state machines. That is, the state machine has: a finite number of states, each representing a mode of the modelled system; a finite set of transitions, each modelling events that change the system state; and a single initial state, modelling the starting state of the state machine. The state machine can be in only one state at a time, and perform only one transition in each state for each possible input.

The rules for generating VDM-SL models from Emucharts diagrams are as follows, and illustrated in an example in the next section:

- A VDM-SL module is generated for each Emucharts diagram. The name of the module is the name of the Emucharts diagram.

- A VDM-SL state block `EmuchartState` is generated for specifying the state of the VDM-SL model. The record includes a field for each variable declared in the Emucharts diagram. The name and type of each field is the name and type of the variable from which the field has been generated. Two additional record fields, `current_state` and `previous_state`, are also automatically generated: the former represents the current machine state; the latter represents the previous machine state.
- A VDM-SL `mk_EmuchartState` record constructor is available to initialise the state of the VDM-SL model with the initial values of the variables declared in the Emucharts diagram.
- A VDM-SL enumerated type `MachineState` is generated for each Emucharts state. The enumerated type constants are the Emucharts state labels.
- A VDM-SL transition function of type `EmuchartState` → `EmuchartState` is generated for each unique transition name in the Emucharts diagram. The function argument models the current state of the VDM-SL model. The function return models the next state of the VDM-SL model after the execution of the transition function.
- The body of each VDM-SL transition function is a sequence of conditional *if-then-else* blocks. The Boolean expression used in each conditional block is the conjunction of two elements: the transition condition specified in the transition label; and a Boolean expression based on the current Emucharts state. The body of each conditional block is a series of modifier expressions (`mu(...)`) that update the current model state according to the transition actions specified in the diagram. The modifier expressions are chained to each other using the *let-in* construct.
- A VDM-SL permission function of type `EmuchartState` → `bool` is generated for each function of the VDM-SL model. The body of the permission function is the disjunction of the Boolean expressions used in the top-level *if-then-else* blocks that make up the body of the transition function.
- A VDM-SL operation is generated for each transition function.

4 Example

In this section we use PVSio-web to develop a device prototype that can be formally analysed. The aim is to demonstrate that:

- Emucharts diagrams conveniently hide the technical details of formal languages, and thus make formal verification technologies more accessible to non-experts of formal methods.
- PVSio-web enables rapid generation of a realistic prototype that allows developers to perform an early evaluation of the device, when a physical prototype of the device is not readily available.
- The VDM-SL model generator enables automatic generation of VDM-SL executable models that can be formally analysed within Overture.

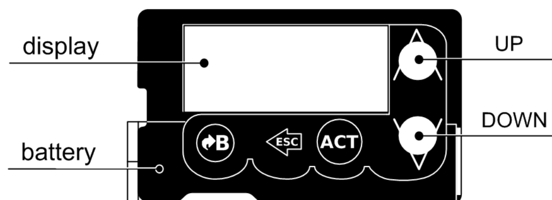

Fig. 2: Blueprint of an insulin pump with UP and DOWN buttons.

4.1 Description of the system

The considered system is an insulin pump used to treat type 1 diabetes. The device allows its user to specify therapy parameters such as the amount of insulin to be injected to keep the blood glucose level under control (bolus dose). The pump is battery-powered, and is turned on by inserting a battery in to the device.

Its data entry system consists of two buttons (UP and DOWN) and a display — a blueprint of the device is in Figure 2. Here, we focus on the behaviour of the device only for data entry of bolus doses. When entering a bolus dose, a click on the UP button increments the display value by 0•1. Similarly, a click on the DOWN button decrements the display value by 0•1. The maximum bolus dose is 10 units.

In the following sub-sections, we develop an Emucharts diagram that models the behaviour of this data entry system. The Emucharts diagram is used within PVSio-web to drive the behaviour of an interactive prototype based on a PVS model (further details about how these prototypes are generated can be found in [1,2]). The same Emucharts is then used to generate an executable VDM-SL model that can then be further analysed within Overture.

4.2 Emucharts Diagram

An Emucharts diagram modelling the described behaviour of the device is shown in Figure 3. The diagram includes two states, *on* and *off*, modelling whether the device is powered on or off. The *off* state is the initial state. In the diagram, this is represented using a default initial transition that enters the *off* state.

A transition *turn_on* changes the device state from *off* to *on*. This models the action of inserting a battery into the device. Similarly, a transition *turn_off* changes the device state from *on* to *off*. This models the action of removing the battery from the device, or a depleted battery.

Two state transitions *click_UP* model the behaviour of UP button clicks. One transition models button clicks when the display value is less than 10. In this case, the display value is incremented by 0•1. The transition condition is therefore $display < 10$, and the transition action specifies the new value of the display using the assignment expression *display := display + 1*. The other transition is for handling the boundary case at 10. In this case, a click on the UP button resets the display value to 0.

Two other transitions *click_DOWN* model the behaviour of the DOWN button. One transition is for values above 0, and decrements the current display value by 0•1. The

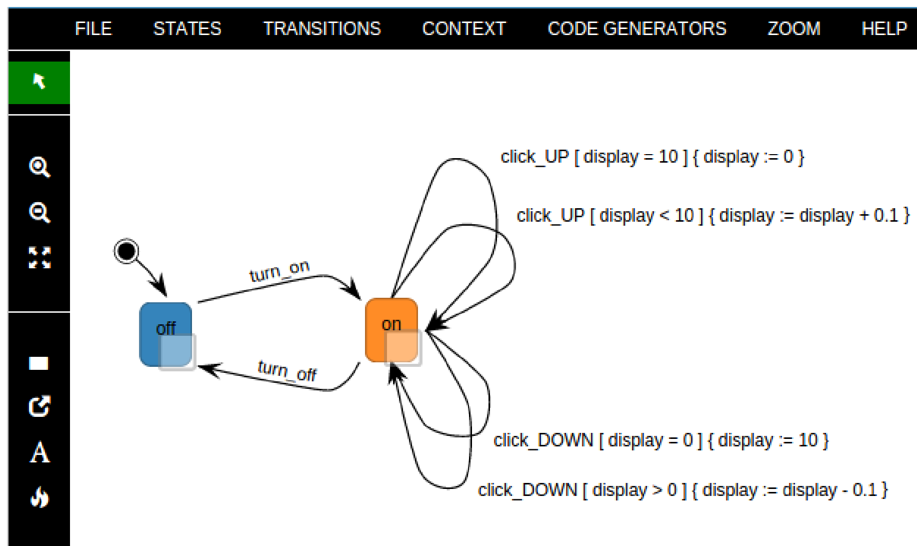

Fig. 3: Snapshot of the Emucharts Editor while drawing a diagram modelling the behaviour of the data entry system described in sub-section 4.1.

other transition is for the boundary case at 0. In this case, a click on the DOWN button changes the display value to 10 (this makes the behaviour of the DOWN button symmetric with respect to the UP button).

Finally, a variable *display* is declared in the Emucharts context for modelling the display value. The type of the variable is *real*, and the initial value is 0.

4.3 Generating and Analysing an Interactive Prototype

The behaviour modelled with the Emucharts diagram is now used as a basis to generate a realistic prototype that resembles the look and feel of the final product. This prototype enables lightweight formal analysis for early evaluation of the device behaviour.

The prototype is generated within PVSio-web using the Prototype Builder front-end. This is done by loading a realistic picture of the device in the tool, and creating interactive areas over the picture (see Figure 1). Three interactive areas are created in this case. The first is for the display, and is associated to the Emucharts variable `display`. Two more capture the user pressing the buttons in the picture of the device. These interactive areas translate the button presses into commands for animating the formal model associated with the Emucharts diagram. This formal model is automatically generated from the diagram, and executed within PVS [5] using its the native PVSio [9] animation environment.

Once the prototype is generated, one can explore the formal model by clicking buttons of the device, seeing the results of the interactions on the device display (see Figure 4). Using the prototype, a lightweight formal analysis can be performed before starting the full formal analysis. For example, one can perform an expert walkthrough [14]

Fig. 4: The insulin pump prototype executed within PVSio-web.

of the device. It is a usability inspection method performed by human-computer interaction specialists for identifying issues with the user interface of a system. By exploring the behaviour of the prototype, for example, the following conceptual issue can be easily identified with few exploratory input key sequences: when the display is 0 and the down button is pressed, the display value rolls over to 10. This behaviour is unsafe, as a single accidental button press while programming the bolus dose could lead to accidental overdoses [4, 15]. As a matter of fact, a real medical device on the market has been recalled because of this design issue [16, 17].

It is worth noting that the prototype has been generated without a full model of the system. Therefore, these prototypes can be generated at the early stages of device development, allowing developers to identify conceptual design issues in advance, and fix them before committing to potentially expensive design decisions.

4.4 Generating a VDM-SL Model

The newly developed VDM-SL model generator is now used to generate a VDM-SL model from the same Emucharts diagram used for the interactive prototype. The generated model can be imported within Overture for further formal analysis (type checking, analysis of proof obligations, generation of test cases, etc.). The steps illustrated in Section 3 are now illustrated for the diagram. The full VDM-SL model generated from the diagram is given in the Appendix.

The VDM-SL model generator creates the type definitions first. An enumerated type *MachineState* is generated that includes two enumerated constants, one for each state represented in the Emucharts diagram.

```
MachineState = <off> | <on>;
```

Listing 1.1: MachineState type

A state block *EmuchartState* is then generated that includes: a field *display* of type *real*, which models the variable defined in the Emucharts context; two fields *current_state* and *previous_state* of type *MachineState*, which store information about the current and previous active state of the state machine.

```
state EmuchartState of
  current_state: MachineState
  previous_state: MachineState
  display: real
```

Listing 1.2: VDM-SL model state

A function *init* is then generated that defines the initial model state. The initial value of the display is 0, the initial value of the current state is *<off>*, as specified in the Emucharts context.

```
init s == s = mk_EmuchartState(<off>, undefined, 0)
```

Listing 1.3: VDM-SL initial state

Functions representing transitions of the state machine are then generated. Transition functions with the same name are automatically merged into the body of a single VDM-SL function. For example, a single function *click_UP* is generated that models the two *click_UP* transitions specified in the Emucharts diagram.

The body of the generated VDM-SL function is, at the top level, a series of *if-then-else* statements. Each conditional statement is generated from a transition function included in the diagram. The Boolean expressions used in the conditional statement are based on both the transition conditions specified in the transition labels, and on the structure of the diagram (in particular, information about which state the transition is leaving from). For example, a Boolean expression generated for the *click_UP* function is $s.current_state = <on>$ and $s.display < 10$, as the arrow representing the transition leaves the *on* state, and the label of the transition includes a condition $display < 10$.

```
click_UP: EmuchartState -> EmuchartState
click_UP(s) ==
 if (s.current_state = <on>) and (s.display < 10) then ...
 elseif (s.current_state = <on>) and (s.display = 10) then ...
 else undefined
```

Listing 1.4: VDM-SL transition function (overall structure)

The body of each conditional block is then generated. Each block always starts with function *leave_state*. This is an auxiliary function that updates field `previous_state` of the VDM-SL model state with the label of the state that the transition leaves. Each block ends with another auxiliary function, *enter_into*, that updates `current_state`

with the label of the state that the transition enters. The actions specified in the Emucharts diagram are state updates, therefore they are translated using the VDM-SL *mu* operator. Consider transition *click_UP* relative to the case when the display is less than 10. The transition leaves and enters the same state (<on>), and the action specifies that the display value is incremented by 0.1 when the transition is executed.

```
... let new_s = leave_state(<on>, s) in let
        new_s = mu(new_s, display |-> s.display + 0.1 )
    in enter_into(<on>, new_s) ...
```

Listing 1.5: VDM-SL transition function (example state update)

Finally, permission functions are automatically generated by the VDM-SL model generator to restrict the domain of the VDM-SL transition function, and thus enable verification of pre- and post-conditions. For example, the permission function for *click_UP* returns the disjunction of all conditions used in the body of the *click_UP* function. This makes the domain of function *click_UP* explicit, and is used by the VDM interpreter to perform essential sanity checks related to how the function is used in the model.

```
per_click_UP: EmuchartState -> bool
per_click_UP(s) ==
   ((s.current_state = <on>) and (s.display < 10)) or
   ((s.current_state = <on>) and (s.display = 10));
```

Listing 1.6: VDM-SL permission function

4.5 Analysis in Overture

We now carry out an analysis of the generated VDM-SL model using two features of Overture: Proof Obligation Generation and Combinatorial Testing [18].

Overture generates Proof Obligations for a VDM model to ensure the internal consistency of the model. Example checks involve assessing the legal use of types and functions in the model. Besides validating core aspects of the semantics of the VDM-SL model, in our case the analysis is also useful for validating the correct implementation of the VDM-SL model generator. Applying the Proof Obligation Generator to the generated model yields four proof obligations, all of them ensuring legal application of the various state transition functions. For example, for the VDM-SL operation representing transition *turn on*, a proof obligation is generated (see Listing 1.7) to ensure that function `turn_on` is correctly used according to its permission. This proof obligation, as well as the others generated for this example, are trivially true, thus confirming that the generated model is well-formed.

```
pre_turn_on(EmuchartState) => pre_turn_on(EmuchartState)
```

Listing 1.7: Sample proof obligation to ensure correct use of functions.

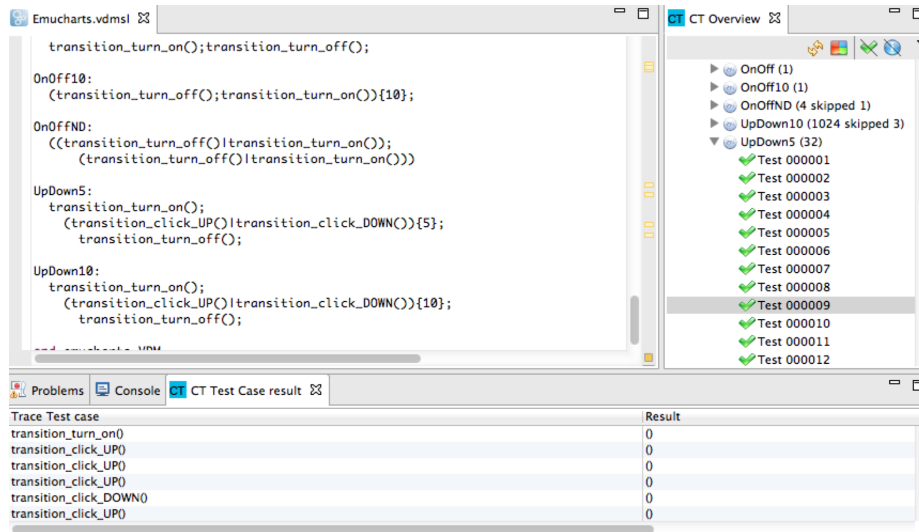

Fig. 5: The Overture Combinatorial Testing feature.

The Overture Combinatorial Testing tool generates test cases for the formal model from traces, allowing one to quickly specify and execute multiple usage scenarios. This can be extremely useful for validating the behaviour of the model against device prototypes or the final product. In Overture, test cases are specified using a trace notation that is akin to regular expressions. In Listing 1.8 we show an example trace for the model developed in the previous sections. It specifies that test cases are generated to explore the following use case: the device is turned on; then, the up and down buttons are randomly pressed 10 times; and then, the device is turned off. This trace expands to 1024 test cases that ensure all combinations of up and down are explored. The results for execution of this combinatorial test trace (and others) are shown in Figure 5. Most tests pass except for 13 which are inconclusive due to violated pre-conditions on the outside test calls such as attempting to turn off a device that is already off (note also that 4 tests are skipped because they share a sequence of calls with an inconclusive test and thus are pointless to execute).

```
traces

UpDown10:
  transition_turn_on();
  (transition_click_UP() | transition_click_DOWN()) { 10 };
  transition_turn_off();
```

Listing 1.8: Sample regular expression.

5 Concluding Remarks

We have illustrated the results achieved to date on integrating PVSio-web with the Overture platform. The integration allows developers to automatically generate VDM-SL models from a state machine description created using the PVSio-web Emucharts Editor. Formal models are thus created without the developer having a deep understanding of the VDM syntax. Also, because the Emucharts Editor incorporates model generators for other formal languages (PVS [5], MAL [19], PIM [20]), the developed tool can be conveniently used to translate VDM-SL state machine models from/to these other formal languages. Future work will extend this initial integration to give Overture and PVSio-web users even more benefits. For example, we plan to further extend the semantics supported by the model generator, e.g., to support diagrams specifying non-deterministic choices, and hierarchical state machines. Another extension relates to the ability of importing VDM models that are manually crafted by developers. This will ease reuse of models and examples already developed by the VDM community. Besides the Emucharts Editor, we plan to integrate two other components of PVSio-web with Overture. One component is the PVSio-web Prototype Builder, which handles the execution of prototypes developed within PVSio-web. The current implementation of this component uses PVSio as execution environment. We will link the Prototype Builder to the Overture interpreter. This will allow Overture users to send commands to the Overture interpreter by interacting with realistic prototypes resembling the real system being modelled, rather than by typing commands in the Overture interpreter console. The other component is the PVSio-web Co-Simulator, which enables integrated simulation of models developed using different modelling and analysis tools. We aim to explore how this component can be integrated with the VDM tool Crescendo [21] for collaborative modelling and simulation. Further work is also needed to determine how best to incorporate PVSio-web within future releases of Overture.

Acknowledgments. This work is part of CHI+MED (EPSRC grant EP/G059063/1).

References

1. Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. PVSio-web 2.0: Joining PVS to Human-Computer Interaction. In *27th International Conference on Computer Aided Verification (CAV2015)*. Springer, 2015. Tool and application examples available at `http://www.pvsioweb.org`.
2. Patrick Oladimeji, Paolo Masci, Paul Curzon, and Harold Thimbleby. PVSio-web: A tool for rapid prototyping device user interfaces in PVS. In *5th International Workshop on Formal Methods for Interactive Systems (FMIS2013)*, 2013.
3. Paolo Masci, Yi Zhang, Paul Jones, Paul Curzon, and Harold Thimbleby. Formal Verification of Medical Device User Interfaces Using PVS. In *ETAPS/FASE2014, 17th International Conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2014.
4. Paolo Masci, Patrick Oladimeji, Paul Curzon, and Harold Thimbleby. Tool demo: Using PVSio-web to demonstrate software issues in medical user interfaces. In *4th International Symposium on Foundations of Healthcare Information Engineering and Systems (FHIES2014)*, 2014.
5. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.

6. Peter G. Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The overture initiative integrating tools for VDM. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.
7. Tomohiro Oda and Keijiro Araki. Overview of VDMPad: An Interactive Tool for Formal Specification with VDM. In *Proc. of International Conference on Advanced Software Engineering and Information Systems (ICASEIS)*, 2013.
8. Tomohiro Oda, Keijiro Araki, and Peter G. Larsen. VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In *To appear in Proc. of FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2015.
9. Cesar Muñoz. Rapid prototyping in PVS. Technical Report NIA Report No. 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, 2003.
10. Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. SCR: A toolset for specifying and analyzing software requirements. In *Computer Aided Verification*, pages 526–531. Springer, 1998.
11. Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising Event-B Models with B-Motion Studio. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, pages 202–204. Springer-Verlag, November 2009.
12. David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
13. Glenn E. Krasner and Stephen T. Pope. A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.
14. Jakob Nielsen. Usability inspection methods. In *Conference companion on Human factors in computing systems*, pages 413–414. ACM, 1994.
15. Abigail Cauchi, Andy Gimblett, Harold Thimbleby, Paul Curzon, and Paolo Masci. Safer 5-key number entry user interfaces using differential formal analysis. In *BCS-HCI '12 Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers, BCS-HCI 2012, 12-14 September 2012, Birmingham, UK*, pages 29–38. British Computer Society, 2012.
16. Medtronic. Important medical device safety information regarding the safe use of the Medtronic insulin pump. `http://www.medtronicdiabetes.com/res/img/pdfs/Insulin-Delivery-Through-Main-Menu-Button-Keypad_US-Customer-Letter.pdf`, 13 March 2014.
17. US Food and Drug Administration (FDA). Class 2 Recall Medtronic MiniMed Paradigm REALTime and Paradigm REALTime Revel CGM System and MiniMed 530G System. Manufacturer and User Facility Device Experience Database (MAUDE), Recall Event ID 68277, 22 August 2014. `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRES/res.cfm?id=127259`.
18. Peter G. Larsen, Kenneth Lausdahl, and Nick Battle. Combinatorial testing for VDM. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 278–285. IEEE, 2010.
19. José C. Campos and Michael D. Harrison. Interaction engineering using the ivy tool. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS09)*, pages 35–44. ACM, 2009.
20. Judy Bowen and Steve Reeves. Modelling safety properties of interactive medical systems. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS13, pages 91–100. ACM, 2013.
21. Peter G. Larsen, Carl Gamble, Kenneth Pierce, Augusto Ribeiro, and Kenneth Lausdahl. Support for Co-modelling and Co-simulation: The Crescendo Tool. In *Collaborative Design for Embedded Systems*, pages 97–114. Springer, 2014.

Appendix: Full VDM-SL Model

```
module emucharts_MedtronicMinimed530G_VDM
exports all
definitions

types
  -- machine states
  MachineState = <off> | <on>;
  -- emuchart state
  state EmuchartState of
    current_state: MachineState
    previous_state: MachineState
    display: real
  -- initial state
  init s == s = mk_EmuchartState(<off>, undefined, 0) end

functions
  -- utility functions
  enter_into: MachineState * EmuchartState -> EmuchartState
  enter_into(ms, s) == mu(s, current_state |-> ms );
  leave_state: MachineState * EmuchartState -> EmuchartState
  leave_state(ms, s) == mu(s, previous_state |-> ms );

  -- transition functions
  per_turn_on: EmuchartState -> bool
  per_turn_on(s) == ((s.current_state = <off>));
  turn_on: EmuchartState -> EmuchartState
  turn_on(s) ==
    if (s.current_state = <off>)
    then let new_s = leave_state(<off>, s)
          in enter_into(<on>, new_s)
    else undefined
  pre per_turn_on(s);

  per_turn_off: EmuchartState -> bool
  per_turn_off(s) == ((s.current_state = <on>));
  turn_off: EmuchartState -> EmuchartState
  turn_off(s) ==
    if (s.current_state = <on>)
    then let new_s = leave_state(<on>, s)
          in enter_into(<off>, new_s)
    else undefined
  pre per_turn_off(s);

  per_click_DOWN: EmuchartState -> bool
  per_click_DOWN(s) == ((s.current_state = <on>) and (s.
      display > 0)) or ((s.current_state = <on>) and (s.
      display = 0));
  click_DOWN: EmuchartState -> EmuchartState
```

```
  click_DOWN(s) ==
    if (s.current_state = <on>) and (s.display > 0)
    then let new_s = leave_state(<on>, s) in let
             new_s = mu(new_s, display |-> s.display - 0.1 )
          in enter_into(<on>, new_s)
    elseif (s.current_state = <on>) and (s.display = 0)
    then let new_s = leave_state(<on>, s) in let
             new_s = mu(new_s, display |-> 10 )
          in enter_into(<on>, new_s)
    else undefined
  pre per_click_DOWN(s);

  per_click_UP: EmuchartState -> bool
  per_click_UP(s) == ((s.current_state = <on>) and (s.display
     < 10)) or ((s.current_state = <on>) and (s.display=10));
  click_UP: EmuchartState -> EmuchartState
  click_UP(s) ==
    if (s.current_state = <on>) and (s.display < 10)
    then let new_s = leave_state(<on>, s) in let
             new_s = mu(new_s, display |-> s.display + 0.1 )
          in enter_into(<on>, new_s)
    elseif (s.current_state = <on>) and (s.display = 10)
    then let new_s = leave_state(<on>, s) in let
             new_s = mu(new_s, display |-> 0 )
          in enter_into(<on>, new_s)
    else undefined
  pre per_click_UP(s);

operations
  transition_turn_on: () ==> ()
  transition_turn_on() == EmuchartState := turn_on(
     EmuchartState)
  pre pre_turn_on(EmuchartState);

  transition_turn_off: () ==> ()
  transition_turn_off() == EmuchartState := turn_off(
     EmuchartState)
  pre pre_turn_off(EmuchartState);

  transition_click_DOWN: () ==> ()
  transition_click_DOWN() == EmuchartState := click_DOWN(
     EmuchartState)
  pre pre_click_DOWN(EmuchartState);

  transition_click_UP: () ==> ()
  transition_click_UP() == EmuchartState := click_UP(
     EmuchartState)
  pre pre_click_UP(EmuchartState);
end emucharts_MedtronicMinimed530G_VDM
```

Extending the Overture code generator towards Isabelle syntax

Luís Diogo Couto and Peter W. V. Tran-Jørgensen

Department of Engineering, Aarhus University, Denmark
{ldc,pvj}@eng.au.dk

Abstract. Overture has a Code Generation Platform (CGP), designed with extensibility in mind but this extensibility has never been thoroughly tested before. In this paper, we explore the extensibility of the Overture CGP by developing code generation support targeting an Isabelle embedding of VDM. We compare our solution to an existing hand-coded VDM to Isabelle translation based on direct traversals of the VDM AST and show that using the CGP led to a decrease in code volume of 86%. We also report various extensibility improvements that have been incorporated into the CGP as part of our work.

Keywords: VDM, code generation, Isabelle, extensibility

1 Introduction

The Overture tool[1] for VDM [6] has a Code Generation Platform (CGP) that was originally developed targeting the Java language but was designed with extensibility in mind. The intent of the CGP is to make it easy to contribute new Code Generation (CG) support for new languages to Overture [12]. Currently, the CGP supports the original Java code generation as well as an experimental generation of C++. The extensibility features of the CGP have never been thoroughly tested since C++ generation is similar to Java generation.

In this paper, we further explore the extensibility of the CGP by developing experimental support for generation of Isabelle syntax, which differs from Java more significantly than C++ does. The reason for this is that Java and C++ are both imperative OO languages and Isabelle is not. The process for developing this translation is also generalised into a standard methodology for developing CGP extensions.

There are two reasons for choosing Isabelle: there is already a usable existing embedding of VDM in Isabelle that we can reuse and a corresponding translation that runs on Overture models [3]. This translation was handwritten and as such will provide a good basis of comparison to see if it is really worthwhile to use the CGP. The comparison shows that using the CGP leads to a code volume reduction of 86%.

The remainder of this paper is structured as follows: the code generation platform as well as the existing Isabelle embedding and translation are described in section 2. The steps taken by the developer to construct the new CG extension are described in section 3. Relevant details of the Isabelle translation are discussed in section 4. The results

[1] http://overturetool.org

of the work in terms of the new Isabelle translation and extensibility improvements to the CGP are reported in section 5 and evaluated in section 6. Finally, we discuss future work in section 7 and conclude in section 8.

2 Background

2.1 Isabelle Embedding

This subsection presents the target language of the translation: an Isabelle embedding of VDM. Isabelle [13] is a framework for implementing logical formalisms and the VDM embedding being targeted is one such formalism. It was originally developed for the COMPASS Modelling Language (CML) [15] in the COMPASS project [7] and is built on an Isabelle mechanisation [8] of the UTP semantics used for CML [10].

CML is a combination of VDM and CSP [9]. In particular, the types, values, expressions and functions of CML are lifted from VDM. State is similar although it is handled somewhat differently – state in CML is composed of multiple independent variables much like VDM++ rather than a single record structure. Additionally, CML does not support the **`let be st`** construct due to its non-deterministic nature. The remaining differences between CML and VDM are related to the reactive and Object Oriented (OO) features of the language. Neither are relevant for this translation.

The Isabelle embedding of CML/VDM is a deep embedding, which means that it gives an explicit semantics to each construct of CML/VDM in Isabelle. In other words, rather than translating from VDM to another formalism, each construct in VDM is defined in the embedding and then given a semantics using formalisms available in Isabelle – specifically, higher-order logic.

Furthermore, the parsing capabilities of Isabelle give significant flexibility when defining the syntax of the VDM constructs in the embedding. The end result is that the embedding has its own syntax which is quite similar to that of the VDM language itself. The primary differences lie in separator characters such as `"` to distinguish between Isabelle and VDM syntax, `^` to identify VDM variables and `@` to identify VDM types.

In addition to the syntactical similarities there is also a near one-to-one correspondence between constructs in the source and target languages which facilitates the translation process. However, while CML has OO features the embedding does not support OO so it is suitable for representing VDM-SL models only.

Finally, we briefly describe the manually written existing translation, based on the visitor framework of the Abstract Syntax Tree (AST). The translation visitors traverse the AST and produce an intermediate data structure used to store relevant translation information for each node including its syntax and dependencies. Afterwards, the data structure is used to generate the Isabelle syntax, either with direct conversion to strings or with auxiliary methods and classes for the processing of more complex nodes. Further details about the existing Isabelle translation as well as the embedding are available in [7].

2.2 Code Generation Platform

The reason for using the CGP, and what makes it a viable solution for developing code generators, is found in the way the CGP represents and works with the generated code.

From the VDM AST the CGP constructs an Intermediate Representation (IR) of the generated code, which forms a tree structure that is independent of any particular target language.

Initially, each node in the IR has a one-to-one correspondence to a node in the VDM AST. Subsequently, the IR is subjected to a series of *transformations* in order to change the tree structure into a new form that is easier for a particular code generator to produce code from. More specifically, each transformation represents a rewriting of the IR with the purpose of changing the IR into a form where each node in the resulting tree structure maps easily into the target language. One advantage of this approach is that transformations operate directly on the IR, and therefore they can be shared among code generators. As an example, the Java and C++ code generators use many of the same transformations to eliminate functional-styled constructs in the IR such as quantified expressions and collection comprehensions.

The IR is generated from an AST specification file using the *AstCreator* tool [1]. In addition to the IR nodes, the *AstCreator* also generates mechanisms to walk the tree using visitors [5] as well as functionality to change the tree structure by allowing parts of it to be replaced. Transformations are themselves implemented as extensions to the visitors generated by the *AstCreator*. What characterises a transformation is that in addition to traversing the tree structure, it also manipulates it.

After the IR has been fully transformed, it is handed over to a language-specific backend generator in order to finalise the code generation process. The CGP provides a framework for syntax generation that serves to facilitate production of code in the target language. This framework is based on the Apache Velocity template engine and used for mapping each node in the IR into concrete syntax [14]. This is handled by the *template manager*, which associates each type of IR node to a template file, that describes the code to be produced.

Code generators extending the CGP may need extra nodes in addition to those already defined by the platform. Therefore, the CGP allows new nodes to be added via the *AstCreator* extension mechanism [4]. This mechanism allows the *AstCreator* to produce nodes and visitors that allow construction and traversal of hybrid trees, .i.e. tree structures composed of both IR nodes defined within the CGP and new nodes contributed via an AST specification extension file. In addition to adding new nodes, the CGP also allows existing IR nodes to be extended to include new fields. Finally, the template manager can be redefined to support syntax generation of new nodes added by the user.

3 Methodology

Based on the description of the CGP in subsection 2.2 we now outline the steps used to develop the Isabelle syntax generator. These steps constitute a general methodology for development of code generation support in Overture using the CGP. Others who want to use the CGP to develop code generation support for another target language may benefit from following these steps.

We start out by listing the steps to be carried out by the developer and afterwards we elaborate on each of them.

1. Set up the CGP extension
2. Add new nodes
3. Transform the IR
4. Generate syntax
5. Validate the translation

The first step in the process is only necessary once. The remaining steps are done in an iterative manner. The approach is to start with a very small VDM example and go through the steps until the example is completely translated. Afterwards, the example should be expanded as little as possible and the steps repeated. This is done iteratively until the new CGP extension is complete.

Step 1 - Set up the CGP extension: Broadly speaking, the setting up of the CGP extension consists of subclassing the base code generator class – `CodeGenBase` – that is the common extension point of the CGP. The base code generator is responsible for driving the code generation and providing access to the IR and various settings. It is also responsible for storing data used and generated throughout the code generation process.

Next, it is necessary to construct a new template manager for the extension. This can be done by subclassing the base template manager. This will provide access to the basic CGP template structure which manages an initial collection of template locations. If additional template locations are necessary, the template manager can be used to configure them.

Finally, it is worth setting up a basic test infrastructure to drive the development process. This test infrastructure is responsible for processing a VDM source, passing the respective AST to the code generator and validating the translation outcome.

Step 2 - Add new nodes: If the target language construct being translated is sufficiently different from those of the base IR, then it is likely that a code generator needs extra nodes. If necessary, these can be provided by extending the IR as described in subsection 2.2. Once the extension is defined, the *AstCreator* tool must be invoked in order to generate the extension nodes.

Step 3 - Transform the IR: Constructs that are not supported by the code generator need to be transformed away, using either base IR nodes or extended nodes generated in the previous step. This is done by implementing one or more necessary transformations. It is recommended that transformations be as small as possible so that each transformation only changes the IR in terms of one concept such as removing comprehensions or reordering definitions.

Step 4 - Generate syntax: Once the IR is in a form suitable for code generation, syntax can be generated using the syntax generation framework of the CGP. This is done by creating the Apache Velocity template files for each of the nodes that is to be translated and updating the template manager accordingly.

Step 5 - Validate the translation: Validation of the translation should be done by means of the test infrastructure by comparing the translation output to a reference. Alternatively, executable translated code may also be compiled and executed to ensure it produces the right result. This test should then be stored to use as regression in the continued development of the CGP extension.

4 Translations and Transformations

The Isabelle embedding we are targeting is very similar to VDM in the sense that most constructs in VDM are present in the embedding. As such, the initial version of the IR is already close to what is needed for generation – most nodes in the IR already map directly to a construct in the target language. Therefore, there is a relatively small number of operations that need to be performed over the tree.

The first set of operations is also the simplest and most common: direct syntax translations. These translations can be applied directly to the initial IR nodes that already map directly to a construct in the embedding. A few of them are shown in Table 1. These translations take advantage of the fact that the Isabelle embedding of VDM defines its own syntax which is quite close to that of VDM. In general, the syntax is the same as that of source VDM, except for the following:

– all constructs are delimited by `"` to identify them as user-defined syntax in Isabelle
– variables names are delimited by `^` to mark them as model variables
– types are prefixed by `@` to mark them as model types
– string literals are delimited by `''`

VDM	Isabelle embedding
`x`	`"^x^"`
`int`	`"@int"`
`f(1)`	`"f(1)"`
`"foo"`	`"''foo''"`
`if b then s1 else s2`	`"if ^b^ then ^s1^ else ^s2^"`

Table 1: VDM constructs and their Isabelle embedding counterparts.

To achieve these translations, all that is necessary is to specify the target syntax in the Velocity templates and the CGP handles everything else. Most templates are simple since most translations only need to add minor pieces of Isabelle syntax. A few translations require some extra logic – for example, sequences of type `char` are handled differently from all other sequences – and this is achieved through a handful of auxiliary static methods callable from within the template engine.

The second set of operations consists of tree transformations, of which the first is reordering of definitions. Isabelle does not allow forward referencing in its definitions so any dependency of a definition must be processed before the definition itself. When

generating syntax, the CGP processes definitions in the IR in the order in which they appear so it is necessary to reorder the IR nodes according to their dependency relation. For example, consider the VDM functions shown in Listing 1.1. The initial IR generated for this example would have to be re-ordered as shown in Figure 1.

```
1 f : int -> int
2 f (x) == if x = 0 then 0 else g(x);
3
4 g : int -> int
5 g (x) == x/x;
```

Listing 1.1: A simple forward dependency example.

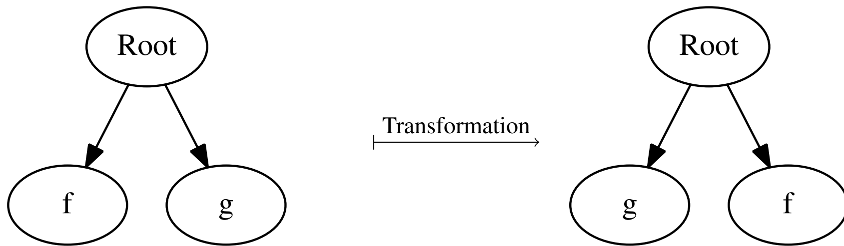

Fig. 1: Dependency sorting transformation.

Dependency sorting is implemented as a CGP transformation that takes an IR module node (the top level element of the IR), constructs a dependency graph of its definitions and then applies a topological sort algorithm [2].

The final operation over the IR is also related to dependency handling, specifically the dependencies between mutually recursive functions. Isabelle can cope with mutually recursive functions but these must be identified as such and grouped together for processing.[2] In order to provide grouping of mutually recursive functions, we construct another transformation that constructs a dependency graph for the function definitions and afterwards applies an algorithm for computing strongly connected components [11]. Thus, the VDM functions in Listing 1.2 would be transformed as shown in Figure 2.

[2] Although Isabelle supports them, the VDM embedding cannot currently cope with mutually recursive functions. However, we have implemented the transformation nonetheless as it was a good way to test the extensibility of the CGP.

```
1  odd: nat -> bool
2  odd (x) == if x = 0
3      then false
4      else even(x-1);
5
6  even : nat -> bool
7  even (x) == if x = 0
8      then true
9      else odd(x-1);
```

Listing 1.2: A simple example of mutual recursion.

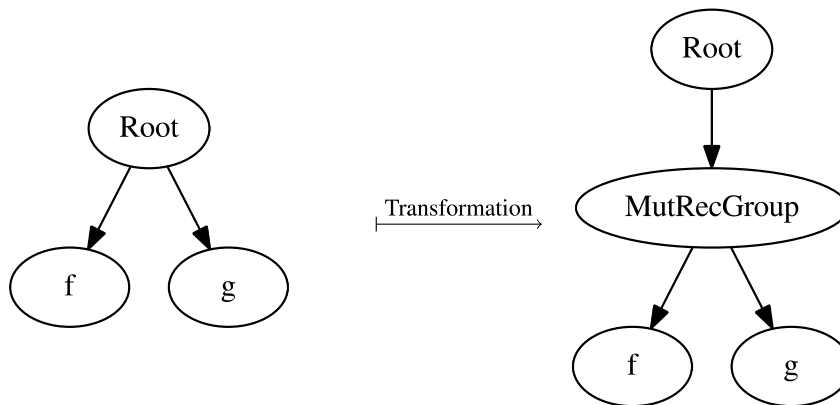

Fig. 2: Mutual recursion grouping transformation.

It is worth noting that the base IR module node does not support mutual recursion groups. As such, we extended the IR to add a new field for it. The mutual recursion transformation takes a base module node as its input and produces an extended module node.

5 Results

5.1 New Isabelle Generation

This section presents the translation from VDM to Isabelle. The translation is demonstrated by means of a complete example, shown in Listing 1.3. Much of the translation is straightforward syntax conversion, however, the example demonstrates the two main issues discussed in section 4: reordering definitions due to dependencies and grouping mutually recursive functions.

Functions `g()` and `f()` shown in lines 3-7 of the VDM model are translated to functions `f()` and `g()` in the Isabelle embedding shown in lines 5-13. Note that the two functions have changed to that `f()` comes before `g()` in the Isabelle source. This is because `f()` is a dependency of `g()` and so must be processed first.

Functions `odd()` and `even()` shown in lines 9-17 of the VDM model are also translated to functions in the Isabelle embedding, shown in lines 15-31. However, the functions in the embedding are delimited by the **`begin_mutrec`** and **`end_mutrec`** keywords which identify them as a block of mutually recursive functions. In Isabelle, such functions must be delimited as they are processed together.

5.2 Code Generation Extensibility Improvements

In addition to constructing the new extension, a series of improvements to the extensibility of the CGP were also carried out. The first set of extensibility improvements had minor impact on the CGP and was related to changing the visibility of various classes and class members. Prior to this work, we were uncertain of which parts of the CGP needed to be exposed to extensions. While it would have been possible to simply expose everything, that would make the CGP too complex to use. By carrying out this work we were able to discover which features to expose and were able to safely keep the rest encapsulated inside the CGP.

As an example of the above, the template manager has a field that defines the folder structure used so store template files. This field was not visible to extensions and that forced an extension to follow the same structure as the base CG with no ability to redefine it. By making the field visible to subclasses, it became possible for each extension to define its own template folder structure.

The second change to increase extensibility had greater impact on the design of the CGP and was related to transformation application. Originally, the CGP was only capable of transforming the internal part of a node. In other words, the root node of the tree could not be changed. This was insufficient for our extension because it was necessary to have a different class at the root of the tree. To address this, the CGP was modified to support transformations that convert between different node types at the root of the tree and thus it became possible to perform transformations between any two arbitrary trees. This new kind of transformation was named *total transformation* and the existing ones were preserved as *partial transformations*. One advantage of the *partial transformation* is that it can rely on the root node of the tree to remain the same and know what kind of node it is. This reduces the amount of conversions that are required to perform the transformation. The *total transformation* is more powerful but will always take as input and produce as output a generic tree node. The CGP was enriched with functionality to help cope with this by converting between generic and specific root nodes via the adapter pattern [5].

6 Evaluation

To assess the effectiveness of using the CGP for Isabelle translation, a simple comparison of volume – measured in Lines of Code (LoC) – was performed between the two

```
1  functions
2
3  g : nat -> nat
4  g (x) == f(x);
5
6  f : nat -> nat
7  f (x) == x;
8
9  odd: nat -> bool
10 odd (x) == if x = 0
11      then false
12      else even(x-1);
13
14 even : nat -> bool
15 even (x) == if x = 0
16      then true
17      else odd(x-1);
```

(a) VDM model.

```
1  theory A
2    imports utp_cml
3  begin
4
5  cmlefun f
6    inp x :: "@nat"
7    out "@nat"
8    is "^x^"
9
10 cmlefun g
11   inp x :: "@nat"
12   out "@nat"
13   is "f(^x^)"
14
15 begin_mutrec
16
17 cmlefun odd
18   inp x :: "@nat"
19   out "@bool"
20   is "if (^x^ = 0)
21     then false
22     else even((^x^ - 1))"
23
24 cmlefun even
25   inp x :: "@nat"
26   out "@bool"
27   is "if (^x^ = 0)
28     then true
29     else odd((^x^ - 1))"
30
31 end_mutrec
32
33 end
```

(b) Isabelle translation.

Listing 1.3: VDM model and respective Isabelle translation.

versions. LoC is an imperfect measure of volume and does not particularly capture effort or productivity. However, it can be effectively and accurately measured and does provide a reasonable measure of the size of an implementation, which is sufficient for our comparison.

Table 2 presents a summary of results. In this table, *Manual* refers to the original visitor-based translation and *CGP* refers to the translation we have implemented. The comparison does not consider components from the original translation that are re-

sponsible for processing CML-exclusive elements that have no counterpart in VDM. To facilitate comparison, we have broadly grouped the sources of both versions into three groupings:

data Refers to classes implementing the intermediary data representation between source and target syntax

process Refers to classes that are used to help process or analyse the intermediary representation

syntax Refers to classes that provide or define the target syntax for final translation printing

	Manual	CGP	ΔLoC_{abs}	ΔLoC_{rel}
data	981	27	954	97.25%
process	2427	538	1889	77.83%
syntax	1395	86	1309	93.84%
Total	4803	651	4152	86.45%

Table 2: Volume comparison between translation implementations measured in LoC.

Looking at the data in Table 2, it is clear that utilising the CGP allows for an implementation with much less volume – a reduction of 86%. There are gains in every grouping but the largest ones are in the internal representation – 97%. This is because the *Manual* version utilises a handwritten data structure, whereas the *CGP* version reuses the IR and the only code necessary is that for defining the necessary data extensions. Likewise, most of the machinery for processing both the source language and the IR is reused from the CGP. Particularly, the construction of the IR from the source AST is handled entirely by the CGP. The *syntax* grouping is also much smaller in the *CGP* version – a reduction of 93% –since it uses the template engine in the CGP which allows for significantly more concise expression of syntax.

7 Future Work

In the future, there are two main avenues for improving this work: the translation itself and the extensibility of the CGP. Beginning with the translation, the most immediate improvement is to expand the coverage of VDM constructs. This is to some extent tied to the support of the embedding but there is a significant number of supported constructs that are not translated. For most of these it is only a matter of adding the relevant templates, although there is also the matter of making the dependency calculator more generic, which should not present a problem.

On the topic of the embedding, it would be worthwhile to switch to a pure VDM embedding. While the similarities between CML and VDM make the current embedding suitable for an initial translation, it would be beneficial to migrate to a dedicated

embedding for VDM that could be maintained and evolved separately as necessary. Furthermore, the current embedding contains multiple definitions supporting the reactive aspects of CML that are unnecessary from a VDM perspective. Finally, a dedicated VDM embedding would allow for syntax that is even closer to that of VDM. Work is already underway on adapting the CML embedding into a pure VDM one.

Returning to the translation, there is a potentially problematic issue in that it is only possible to generate syntax for all definitions of the same kind together in one pass. This is a problem when needing to print definitions of various kinds according to the order of dependencies. The issue is related to the IR being structured as lists of definitions of the same kind. It would need to be altered to support generic definition lists – for example, type and function definitions would be stored in the same list.

In terms of translation, it would also be worthwhile to translate proof obligations along with the model thus allowing them to be discharged in Isabelle. The proof obligations are encoded as ASTs using only the expression subset of VDM. Therefore it should be possible to translate them with the existing machinery and require only some additional syntax to turn them into proof goals for Isabelle.

With regards to the CGP itself, the work presented in this paper has suggested two improvements to be carried out. The first is an architectural refactoring of the CGP. At the moment the CGP is directly tied to the Java code generator and that component must be reused as part of reusing the CGP. While this does not limit the ability to construct new extensions, it does expose a significant amount of Java-related functionality that is not necessary. Therefore, it would be beneficial to refactor the code generator into a *core* component that provides the CGP and a *javacg* component that provides code generation to Java.

Another improvement is related to transformations of the IR. Currently, a new extension must provide all of its transformations and develop them from scratch. It stands to reason that some translations are required for multiple extensions – for example, dependency sorting may be needed in other target languages – so it would be beneficial to reuse existing transformation. However, most transformations make assumptions about the target language and the order in which transformations are applied. This makes it quite challenging to reuse them since none of these assumptions hold in all situations.

8 Conclusion

This paper has presented a VDM to Isabelle translation using the code generation platform of Overture. The initial results show that the translation can be written with a significantly smaller amount of code (86%). Additionally, the use of the platform confers various benefits such as improved maintainability of the intermediary data structure and more easily adjustable syntax (via templates instead of Java strings). Also, any general improvements made to the CGP will be propagated to the translation as well.

The successful development of the Isabelle translation stands as proof of the extensibility of the CGP. Some issues were identified and addressed in order to increase extensibility. Specifically, a more generic transformation mechanism was implemented with support for changing the root node of the tree.

Our initial results show that it is quite worthwhile and beneficial to use the CGP for syntactical translations. The work presented here not only validates the extensibility of the CGP but it also provides a good basis for developing a complete VDM to Isabelle translation.

Acknowledgements

The authors would like to thank Simon Foster, Richard Payne and Sune Wolff for their valuable insight and comments on this paper.

References

1. ASTCreator (2014), https://github.com/overturetool/astcreator
2. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edn. (2001)
3. Couto, L.D., Foster, S., Payne, R.: Towards Certification of Constituent Systems through Automated Proof. In: Workshop on Engineering Dependable Systems of Systems (EDSoS) (May 2014)
4. Couto, L.D., Tran-Jørgensen, P.W.V., Lausdahl, J.W.C.K.: Migrating to an Extensible Architecture for Abstract Syntax Trees. In: 12th Working IEEE / IFIP Conference on Software Architecture (May 2015)
5. E.Gamma, R.Helm, R., J.Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
6. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005), http://overturetool.org/publications/books/vdoos/
7. Foster, S., Payne, R.J.: Theorem Proving Support - Developers Manual. Tech. rep., COMPASS Deliverable, D33.2b (September 2013)
8. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: Unifying Theories of Programming, pp. 21–41. Springer (2015)
9. Hoare, T.: Communication Sequential Processes. Prentice-Hall International, Englewood Cliffs, New Jersey 07632 (1985)
10. Hoare, T., Jifeng, H.: Unifying Theorieses of Programming. Prentice Hall (April 1998)
11. JGraphT (2015), http://jgrapht.org/
12. Jørgensen, P.W., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: The Overture 2014 workshop (June 2014)
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
14. Apache Velocity (2015), http://velocity.apache.org/
15. 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)

Code Generation of VDM++ Concurrency

Georgios Kanakis, Peter Gorm Larsen, Peter W. V. Tran-Jørgensen

Department of Engineering, Aarhus University,
Finlandsgade 22, 8200 Aarhus N, Denmark
gkanos, pgl, pvj@eng.au.dk

Abstract. Code generation from VDM++ to Java has not previously been supported for the concurrency constructs in Overture. The primary challenge here is that the synchronisation primitives in VDM++ and Java are significantly different. In order to bridge this gap it makes sense to create a run-time environment coping with access to invocation of operations that are controlled by synchronisation constraints. This paper explains how the existing VDM++ to Java code generator has been extended to support the concurrency constructs of VDM++ and how such a runtime environment is incorporated. This new feature is also demostrated using a model of a POP3 based email system.

1 Introduction

When one has invested time in producing a formal model of a system it is natural to be interested in whether it is possible to automate any steps towards the final realisation. In this connection one natural candidate to consider is whether it is possible from the formal model to automatically generate the code for the final system. The feasibility of such a fully automatic translation depends on a number of issues and among those are the level of abstraction chosen for the model [1]. It is also necessary to enable a coupling between such generated code with parts that has not been modelled such as user interface aspects as well as existing legacy code. In any case, it is essential that there is a proper balance between the effort invested, the insight gained and the value produced [2]. For a language such as VDM++ there is typically quite a lot of design detail included in the model so it may actually be worthwhile to code generate such models to production code.

In the Overture tool [3,4,5] a code generator from a subset of VDM++ to Java already exists [6]. However, the original version of this framework did not support the concurrency parts of VDM++. The concurrency constructs of VDM++ are based on the notion of threads, but the synchronization constraints are based on logic using constructs called permission predicates. The synchronization principles in VDM++ are different from those available in Java. Thus, there are a number of challenges to overcome to enable automatic code generation of these constructs and ensure the semantics to be preserved [7]. In this paper we present how the existing Overture code generation platform can be extended with support for the concurrency constructs in VDM++ including the synchronization primitives. This is based on the Master thesis work developed by the first author of this paper [8].

After this introduction Section 2 provides the reader with the necessary background to understand the ways in which threads can be synchronized in VDM++ and Java, respectively. Then Section 3 provides a brief overview of the existing code generation platform in Overture. Afterwards Section 4 presents the core of this paper by explaining how the existing code generation platform has been extended to support the concurrency constructs in VDM++. Then Section 5 presents a case study for the concurrency extensions of the code generator. Finally Section 6 presents a few concluding remarks and future directions for this work.

2 Background

2.1 Synchronization in VDM++

The VDM++ dialect was developed to enable modeling of object oriented systems. In addition, concurrency was introduced based on threads in the models. The language provides two main synchronization constructs for coordinating threads. The permission predicates [9] are boolean expressions that are defined for an operation to control when it is allowed to proceed to its execution. A permission predicate thus acts as a guard that controls the execution of an operation by suspending it when it is **`false`** and allows it to proceed when it is **`true`**. Permission predicates may use instance variables and history counters to define their boolean value. History counters are self-contained counters that count the number of invocations, executions and terminations of an operation. Furthermore, the language provides **`mutex(`***op1,..,opN***`)`** [9] definitions which allow the mutual exclusion between operations.

2.2 Synchronization in Java

Java supports concurrency since version 1.2. However, Java is using different primitives to synchronize threads. It provides the monitor construct [10] which treats a method or a block of code as critical section that only one thread can access at any point in time. If the monitor is used by a thread, all other threads have to wait until the monitor is released by the thread using it. There is no condition to control the next thread which will access the monitor but it is arbitrarily chosen by the Java Virtual Machine (JVM). Another more versatile primitive that Java provides is the lock construct [10] which is used on blocks of code to provide mutual exclusion. Locks, also, allow control over the priority of the execution of threads. Furthermore, the lock construct allow a thread to withdraw its request for a lock if it is already acquired by another thread.

2.3 Inheritance differences between Java and VDM++

Another important area where the two languages bear significant differences is for class inheritance. In VDM++, it is possible to have a class that inherits from multiple super classes. However, this is not possible in Java where a class can only extend one super class but can implement multiple interfaces. The common ground between the two languages with respect to inheritance is that both support multi-level hierarchies. The multiple inheritance of VDM++ will affect the active user classes of Java generated code

which they will have to extend the Java **Thread** runtime class to obtain Java thread functionality. In the proposed approach, the runtime class is extended on the top most super class of the active user class. If it inherits from multiple classes, it would not be possible to extend all most top super classes with the Java **Thread** class. Furthermore, multiple inheritance is not supported by the sequential code generation platform.

3 The Code Generation Platform

The VDM++ concurrency code generator developed as part of this work is an extension to the sequential VDM++ to Java code generator available in the Overture tool. Furthermore, the sequential VDM++ to Java code generator is built on top of the Overture code generation platform and therefore the extension for the code generator for the VDM++ concurrency constructs is also based on this platform.

3.1 The code generation platform

Figure 1 shows the overall architecture of the Overture code generation platform, where a code generator targeting a particular language such as Java or C++, is developed as a backend extension to the platform. The goal of using a platform is to promote reuse and reduce the effort needed for developing VDM code generation support for multiple target languages. This is achieved by using a generic representation of the code generated model referred to as the Intermediate Representation (IR) – a tree structure that is independent of any target language.

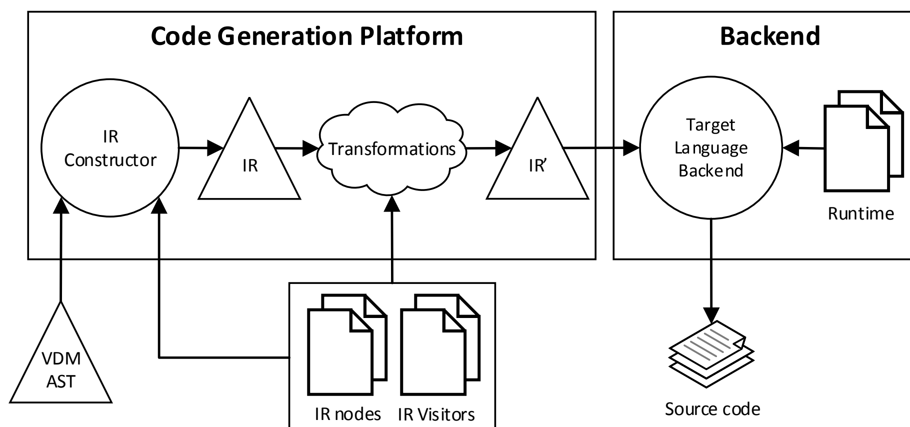

Fig. 1. An overview of the architecture of the code generation platform

The code generation platform subjects the IR to a series of semantically preserving *transformations* that change the tree structure of the IR into a form that is easier for a

target language specific backend to code generate. Transforming the IR includes rewriting constructs that are non-trivial to code generate by replacing them with constructs that are easier for a backend to express in terms of the target language.

Since the difficulty of code generating a particular VDM construct depends on the target language, the code generation platform allows a backend to configure and develop its own transformations. As an example, Java does not have a single construct that can be used to code generate a VDM set comprehension. Instead the Java backend transforms each set comprehension out of the IR by rewriting this construct into an imperatively styled form.

Although transforming the IR potentially results in a larger tree structure the goal is to achieve a final version of the IR where every nodes can be trivially code generated by the backend. In order to bridge the gap between VDM++ and Java concurrency, the work presented in this paper partly involves the development of transformations to rewrite the synchronization constructs of VDM++ into a form that is easier for the Java backend to code generate directly (see subsection 4.4).

3.2 Developing a backend

When the transformation process is completed the IR is handed over to the backend (see Figure 1), which is responsible for finalizing the code generation process. As a last thing, each node must now be mapped into the target language.

The Java code generator uses the Apache Velocity template based technology to map the IR nodes into Java source code [1]. In particular, the Java backend specifies a template for each type of node in the IR that it provides support for. In addition to the concrete Java syntax, the template file uses template code to access information about the state and context of the node in order translate it into Java code.

As shown in Figure 1 a backend may use a code generation runtime library to support the generated code. As an example, the Java code generator uses a runtime library that provides Java implementations for some of the VDM types and operators that are not directly supported in the Java standard library. To mention a few examples, the Java runtime library includes implementations of tuples and tokens as well as the sequence modification operator. As described in section 4 part of code generating VDM++ concurrency involves developing an extension to the Java code generator runtime library in order to provide threads that conform to the VDM semantics as well as auxiliary data structures to maintain history counters.

4 Extending The Code Generation Platform

Permission predicates are described in section 2, a permission predicate has a boolean value when an operation is invoked, the invoked operation is able to proceed to its execution only if the permission predicate is valid. Thus, the operation has to check the value of the permission predicate before it continue to its execution. However, in the sequential code generator when a generated method is called, it is immediately allowed

[1] The Apache Velocity Website: http://velocity.apache.org

to proceed to its execution because their is no way to block it without the support for permission predicates. Even if a permission predicate is defined for its corresponding operation in the VDM++ model. E.g. a generated method needs to wait until all previous activations of the same method are finished. In the VDM++ model, the permission predicate that it is defined is **per** *op1* = **#act**(*op1*) - **#fin**(*op1*) = 0 but it is not present at the generated code.The reason is that the sequential code generator does not support the permission predicate construct. In addition, the history counters used in the permission predicate is not natively supported in Java. Thus, the generated methods does not have this information available when they are invoked.

The information that is stored by history counters must also be recorded in the generated code. Therefore, the runtime environment delegated this responsibility to an external class as a central point of control for these counters. This class acts as a bridge for the generated concurrency constructs of the VDM++ language and the Java language available constructs.

4.1 Bridging the gap between VDM++ and Java

The external `Sentinel` class, as part of the runtime environment, can be linked to the generated classes either as an interface or as a super class. The latter was chosen because the history counters need to be modified according to the state of a method. This functionality was also included in the external class in the form of methods to manage the history counters.

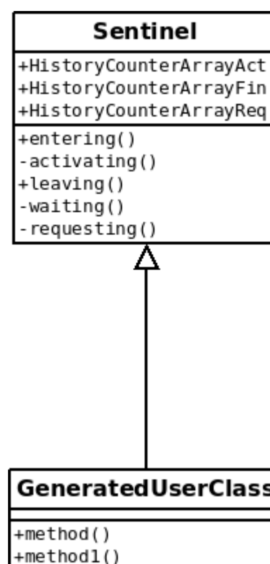

Fig. 2. The `Sentinel` class is extended by user defined classes.

In the `Sentinel` class, each type of history counter is maintained in an array. The array is used to hold history information for the multiple methods that can exist in the user defined classes, each possition of the array can store the information for one method. The history counter arrays are managed by the methods also defined in the `Sentinel` class. This class is extended by all user defined classes that are code generated in order to provide them with access to the history counters. Figure 2 shows the inheritance between the user defined class and the external `Sentinel` class. In addition, the figure presents some of the arrays for the history counters defined in the `Sentinel` class as well as the methods maintaining them.

4.2 Methods in the Sentinel class

Managing the history counters is done in methods in the `Sentinel` class. The first method is *`entering()`* which is used by the methods to initiate the history events. This method invokes the three private methods *`requesting()`*, *`activating()`* and *`waiting()`* for updating the appropriate history counter. Their purposes are to apply synchronization between the threads that invoke a thread-safe method and to apply the update of the history counters. These methods change the values of all history counters apart from **`#fin`** which is maintained by the *`leaving()`* method to change.

`Leaving()` is called in the body of the thread-safe method within a **`try - finally`** statement. The **`finally`** statement guarantees that the *`leaving()`* method will be invoked at the end of the execution of the class method and therefore the counter will be updated even in case an exception is raised.

4.3 Evaluating Permission predicates

The use of the `Sentinel` class does not have the ability to evaluate permission predicates. The problem can be solved by creating a second external class in the runtime environment that implements a mechanism to evaluate the permission predicates. Unfortunately, the use of a second super class is not possible in Java. Therefore, an interface is developed that provides the functionality for the user to define classes that can evaluate their permission predicates. The interface is named `EvaluatePP`

The `EvaluatePP` interface contains only one abstract boolean method, *`evaluatePP()`*, that every user defined class must implement. This method implements a mechanism to evaluate the permission predicates for a given method. The implementation of this method is generated in the user defined classes automatically by the code generator through an IR transformation.

The implementation of the *`evaluatePP()`* method contains multiple **`if-else`** statements that select the called method and return the boolean value of its predicate. For operations that do not have permission predicates defined the default value **`true`** is returned as specified in the VDM++ language manual to indicate that the invocation can proceed. Listing 1.1 provides an example of an implemented *`evaluatePP()`* method with a defined permission predicate generated with in the **`if-else`** statement.

```
1  public Boolean evaluatePP(final Number fnr) {
2   if (Utils.equals(fnr, 0L)) {
3    return sentinel.fin[((A_sentinel) sentinel).opA]>=0L;
4   }
5    ...
6   else
7   {
8    return true;
9   }
10 }
```

Listing 1.1. The *evaluatePP()* method as it is generated by the code generator

The permission predicates need to be reevaluated constantly when a history counter or an instance variable is updated. Thus, an additional method is defined in the `Sentinel` class to act as a way to enable reevaluation of the permission predicates. The *stateChanged()* method notifies all threads that a value in a history counter has been updated. Thus, every method that updates a history counter have to invoke the *stateChanged()* method after the update has been made. This method is used to enable the reevaluation of the permission predicates when an instance variable is updated. The semantics in the VDM++ language are not defined for the initiation of the reevaluation of permission predicates after an instance variable update, however into the generated code the reevaluation of predicates is enabled immediately after an instance variable has been updated.

Need for an inner class The `Sentinel` class acts as a super class of the user defined classes, this creates a inheritance problem when a user define class is also an active class. For active classes in Java to spawn threads, they need to extend the Java `Thread` class. In addition, it is not possible to implement the `Sentinel` class as an interface because the functionality to manage the history counters is needed. However, the solution was found in the use of an inner class within the user defined classes. This approach actually sustained the functionality needed for the history counters and allowed the active classes to extend the `Thread` class for spawning threads. Figure 3 shows the final inheritance scheme that the user defined class will follow when the concurrency constructs of VDM++ are code generated. The main user class is extending the Java `Thread` class and implements the `EvaluatePP` interface while the inner class extends the `Sentinel` class to maintain maintenance of the history counters.

4.4 IR Concurrency Transformations

The concurrency extension for the code generator uses four transformations. The following description briefly explains the purpose of the four transformations and how they enable the concurrency constructs of the VDM++ language to be supported.

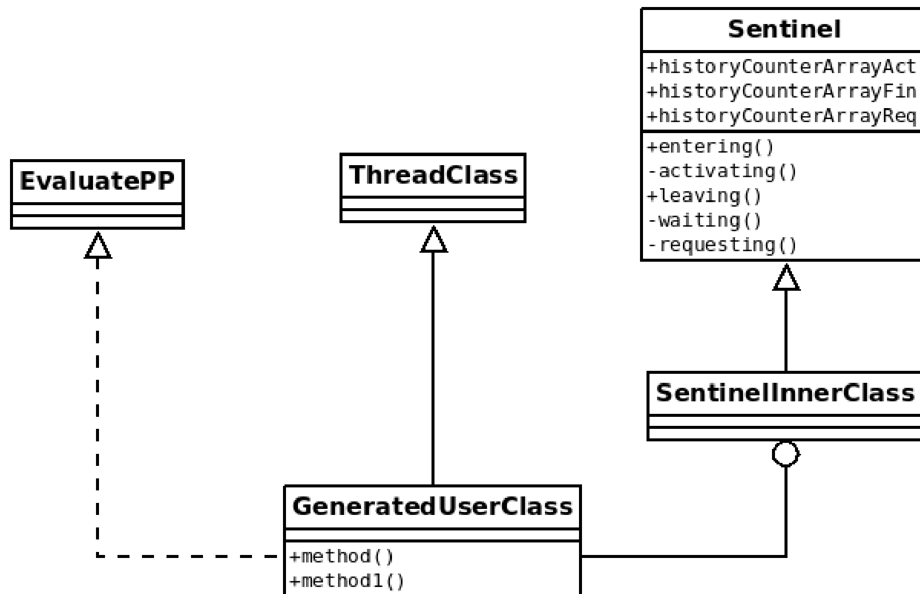

Fig. 3. The user defined class inheritance hierarchy with the inner class for the VDM++ concurrency constructs

- **Main Class Transformation**: Adds the implementation of the *evaluatePP()* method, adds code within class methods to enable history information to be stored and adds a field to the user define class that enables the link with the `Sentinel` class.
- **Inner Class Addition**: Adds the inner class that extends the `Sentinel` class to the user defined classes.
- **Mutex Transformation**: Changes the **`mutex`** definition into permission predicates form.
- **Instance Variable Predicate reevaluation** : Triggers re-evaluation of the permission predicates for the instance variable updates.

Main Class Concurrency Transformation This transformation is applied on the user define classes adding nodes to the IR for concurrency constructs. In particular,it adds a reference to an instance of the `Sentinel` class to hold the history counters for the methods in the class and adds the implementation of the *evaluatePP()* method for the evaluation of the permission predicates. In addition, another major IR rewrite performed by the main class transformation is adding code within the body of the methods of the user defined class to allow the history counters to be stored in the `Sentinel` class for each method. As seen in listing 1.1, the *evaluatePP()* method is being implemented with a permission predicate that is present for one method in the user generated class.

In listing 1.2, the *entering()* and *leaving()* method invocations that are added in the body of the user defined method is shown, lines 2 and 6 respectively. The

invocations of the methods in the `Sentinel` class is done with a numeric parameter which represents the index of the method in the arrays that holds the history counters. In addition, it can be observed in line 5 that the *`leaving()`* method is enclosed in a **`finally`** statement as described in subsection 4.2.

```
1  private void opA() {
2  sentinel.entering(((ExampleClass_sentinel) sentinel).opA);
3   try {
4    //initial body of the method
5   } finally {
6    sentinel.leaving(((ExampleClass_sentinel) sentinel).opA);
7   }
8  }
```

Listing 1.2. method invocation enabling the history counter maintenance

Inner Class Transformation This tranformation adds an inner class that is subclass of the `Sentinel` class to the user defined classes. The inner class and the instance of type `Sentinel` that the main class transformation creates in the user defined class provide access to the arrays and methods in the `Sentinel` class. Furthermore, the inner class transformation creates integer constants in the inner class that they represent the methods in the user defined class, these constants are used as parameters to the *`entering()`* and *`leaving()`* methods presented in Listing 1.2.

Mutex transformation Internally, the VDM++ language treats mutexes as permission predicates and conjunct them with existing permission predicates for a user defined method. This is handled by the replacing of each **`mutex`** definition with the equivalent permission predicate. This transformation is performed by the *MutexDeclTransformation* [8]. In case an operation already has a permission predicate the transformation from **`mutex`** is combined with the existing constraints in a conjunction.

Instance Variable Predicate reevaluation The VDM language manual [9] does not explicitly describe the way the initialization of the reevaluation will happen after the instance variable update. Thus, this transformation add the invocation to the *`stateChanged()`* method after any update to instance variables. The call to the *`stateChanged()`* method in the `Sentinel` class will enable the reevaluation of permission predicates the same way the reevaluation of the permission predicates occurs for the update of the history counters by notifying blocked threads to reevaluate their permission predicates.

5 Case Study - POP3

A model of a POP3 email system (model originally used in [11]) is used as for case study as it contains multiple active classes and multiple synchronization constraints that must be code generated by the concurrency code generation platform. Figure 4 presents

the sequence of actions that the POP3 client and server perform. In addition, it is shown by the sequence that the initiator is the client which send a message to the server as well as it is the terminator of a sequence of actions when it receives the response of the server.

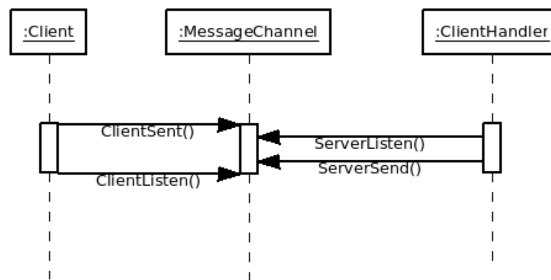

Fig. 4. Sequence of activations between the Client and the POP3 mail Server

Defined synchronization in VDM++ POP3 Model. The POP3 model defined permission predicates for the four actions that the client and the server performs in order the message exchange shown in figure 4. The actions that are synchronized are the `client send`, `client listen`, `server send` and `server listen`. As an example, listing 1.3 presents the permission predicate defined for the operation that controls the action `client send`. In this listing, it is seen that the action should wait until all the other operations that control the other actions have completed.

```
1 per ClientSend => #fin(ServerSend) = #fin(ClientListen)
2                   and #fin(ClientSend) = #fin(ServerListen)
3                   and #fin(ServerSend) = #fin(ClientSend) ;
```

Listing 1.3. The permission predicate for the *`ClientSend()`* operation

Similar permission predicates are defined in the model of the other three operations performed in the sequence of actions.

Generated Synchronization in Java. The sequence of actions between the client and server should be preserved in the generated code. The appropriate code generation of the permission predicate defined in the model is evidence that the sequence of actions will be preserved. Listing 1.4 presents the code generated for the permission predicate presented in listing 1.3. It is seen that the generated permission predicate follows the same pattern as the VDM++ permission predicate using the **`#fin`** history counter and the same equality among the methods that need to be synchronized.

```
(sentinel.fin[((MessageChannel_sentinel) sentinel).ServerSend]
== sentinel.fin[((MessageChannel_sentinel) sentinel).
    ClientListen])
&&
(sentinel.fin[((MessageChannel_sentinel) sentinel).ClientSend]
== sentinel.fin[((MessageChannel_sentinel) sentinel).
    ServerListen])
&&
(sentinel.fin[((MessageChannel_sentinel) sentinel).ServerSend]
== sentinel.fin[((MessageChannel_sentinel) sentinel).ClientSend
    ]);
```

Listing 1.4. The generated permission predicate corresponding to the one presented in listing 1.3

Similar code is generated for the rest of the VDM++ permission predicates. The generated code is similar with the one presented in Listing 1.4. However, the important aspect of it is to perform the same functionality and produce the same sequence of action as the VDM++ model produce. This leads to the execution of the model and the generated code in order to observe the output results.

Changes to the generated classes. The permission predicates are not the only changes that they are needed for obtaining the VDM++ synchronization to the generated code. In addition to the generated permission predicates, the generated methods need to include the calls to the `Sentinel` class enabling the maintenance of the history counters. In Listing 1.5 the generated method for the `client send` action is presented, it is observed in lines 2 and 14 the calls to the `Sentinel` *`entering()`* and *`leaving()`* methods respectively.

```
public void ClientSend(final Object p) {
 sentinel.entering(((MessageChannel_sentinel) sentinel).
     ClientSend);
 try {
  if (debug) {
   Boolean ignorePattern_5 = io.echo("***> ClientSend");
  }
  Send(((Object) p));
  if (debug) {
   Boolean ignorePattern_6 = io.echo("***> fin ClientSend");
  }
 } finally {
   sentinel.leaving(((MessageChannel_sentinel) sentinel).
       ClientSend);
 }
}
```

Listing 1.5. Generated `cliend send` method with concurrency features

Similar code is generated for all the operations in the POP3 model to maintain the history counters of every method. Furthermore, every class in the model has an inner class that extends the `Sentinel` class to provide access to the arrays and the methods maintaining the history counters. In Listing 1.6, an example of a generated inner class for the `MessageChannel` class is presented with the constants created for the methods in the enclosing class. In case of overloaded methods, only one constant is generated for the same method name to preserve the way history counters and permission predicates treat overloaded operations in the VDM++ language.

```
1  public static class MessageChannel_sentinel extends Sentinel {
2    public static final int Send = 0;
3    public static final int Listen = 1;
4    public static final int ServerSend = 2;
5    public static final int ClientListen = 3;
6    public static final int ClientSend = 4;
7    public static final int ServerListen = 5;
8     ...
9    public MessageChannel_sentinel(final EvaluatePP instance) {
10    init(instance, function_sum);
11   }
12 }
```

Listing 1.6. The generated inner class of the `MessageChannel` class

The constants are used as parameters to the *`entering()`* and *`leaving()`* methods representing the indexes of the arrays in the `Sentinel` class.

Observing the results The model outputs the sequence in which the client and the server exchange messages, their sequence is following the diagram presented in Figure 4. The output is printed to the console view of the Overture tool. The initial version of the model printed different type of messages not only these for the sequence of the activations described figure 4. These messages included the type of commands the cliend send, the responses from the server and the termination of the communication between them. The model is slightly modified to print only the messages referring to the actions of the sequence it performs to the console. The rest types of the messages are logged into a file.

The modification is performed to the model to simplify the process of comparing its results with these from the generated code because the rest of the messages are without synchronization but are only affected by the choices the deterministic interpreter [12] of the tool makes. The Overture interpreter is designed deterministic in order to allow error to be easily reproduce for a specific input. On the other hand, this cannot be achieved in general in the generated code because a concurrent program can produce every possible execution path due to the underline system architecture and the non-determinism of the JVM [13].

Listing 1.7 shows the output in the console of Overture tool that is produced by the POP3 model for the sequence of actions. It is seen that it follows the sequence in

Figure 4. This output is going to be compared with the one that the generated code is producing.

```
1 Creating POP3ClientHandler
2 ***> ClientSend
3 ***> fin ClientSend
4 ***> ServerListen
5 ***> fin ServerListen
6 ***> ServerSend
7 ***> fin ServerSend
8 ***> ClientListen
9 ***> fin ClientListen
```

Listing 1.7. The results for the sequence of actions produced by the model

After the results are obtained from the POP3 model, the model is code generated into Java with its concurrency features. The generated Java code is executed and the results are printed to the console. Listing 1.8 presents the output with the sequence of the actions the generated Java code has produced.

```
1 Creating POP3ClientHandler
2 ***> ClientSend
3 ***> fin ClientSend
4 ***> ServerListen
5 ***> fin ServerListen
6 ***> ServerSend
7 ***> fin ServerSend
8 ***> ClientListen
9 ***> fin ClientListen
```

Listing 1.8. The results produced by the Java generated code of the POP3 server

The similar functioning of the synchronization mechanisms in the VDM++ model and the generated code causes the output in listings 1.7 and 1.8 to be identical. The results of the concurrency extension of the code generation platform can be further proven by changing the Overture interpreter to output all the possible execution paths with their results. If the result of the generated code belongs in the set of the results of the interpreter then the implementation produces accurate results [14].

6 Concluding Remarks and Further Work

This paper has presented an extension of the sequential Java code generator of the Overture tool enabling the automatic generation of the concurrency aspects of a VDM++ model. However, the results of the generated code may be slightly different than the one chosen in the Overture interpreter. The point is that in general a VDM++ model

including concurrency may yield non-deterministic results whereas the interpreter on purpose has been designed to deterministically select one possible model [12]. In fact the original version of the POP3 example contained parts that are not tightly synchronized, so initially it was thought that the behavior of the generated code was not correct. Thus, the generated code behaves semantically similar to the model if tight synchronization is applied to the model, whereas in case of looser synchronized models the results may vary due to the determinism of the Overture interpreter. Despite the extension of the code generation platform for the concurrency features of VDM++, there is ongoing work with the VDM-RT notation. In theory it would be possible to let the interpreter explore what the result would be in all possible (including non-deterministic) models but this would have severe consequences for the performance of the execution speed [14].

In the VDM-RT notation additional concurrency constructs are included. Initial integration with code generation of the distribution aspects of VDM-RT [15] has been made already. The concurrency constructs VDM-RT inherits from the VDM++ are supported in the VDM-RT code generation as a result of the work presented in the thesis [8] of the first author of this paper. Additionally, asynchronous operations that VDM-RT provides are also supported and they are code generated as methods which are spawned as a new thread when invoked. However, there is more work to carry out. The VDM-RT includes the notion of time and two additional types of threads, the periodic and the sporadic threads. These features of the VDM-RT language are not yet supported by the code generation platform. In addition to VDM-RT concurrency code generation, other areas can be investigated in the future.

Future Work

The future plan for this work is to improve the existing extension for the concurrency features of VDM++ with a feature that is recently introduced into the language: The manual termination of a thread using the thread `stop()`. Java used to have a `stop()` method to manually terminating a thread but it was deprecated because it was inherently thread unsafe. Since the deprecation of the `stop()` method, Java has no way to manually terminate a thread directly. The manual termination of a thread is handled indirectly by interrupting it and catching the generated exception.Thus, an indirect way should be implemented in the code generator in order to be able to support the manual termination of threads as it is specified in VDM++.

In addition, further integration with the code generator for VDM-RT is necessary in order to support the features of VDM-RT that are currently unsupported. Also, the notion of time have to be considered in the code generator for VDM-RT and its semantics to be preserved in the generated code.

The IR of the code generation platform is independent from any target language. This means that the back-end can generate code into different target language provided the necessary language templates. The reuse of the existing IR transformations for other target languages can be examined. The code generation of concurrency in the C++ language can be the next step for this work.

Another area which can be considered for further future work is the possibility to hide the concurrency feature of Java entirely in the runtime environment. This approach will not modify the user defined class in contrast to this approach which adds a lot of

functionality in the user defined classes. Also, it will be an opportunity to compare the two different approaches in respect to some properties like code readability, execution time, etc.

References

1. Kramer, J.: Is Abstraction the Key to Computing? Communications of the ACM **50**(4) (2007) 37–42
2. Fitzgerald, J.S., Larsen, P.G.: Balancing Insight and Effort: the Industrial Uptake of Formal Methods. In Jones, C.B., Liu, Z., Woodcock, J., eds.: Formal Methods and Hybrid Real-Time Systems, Essays in Honour of Dines Bjørner and Chaochen Zhou on the Occasion of Their 70th Birthdays, Volume 4700, Springer, Lecture Notes in Computer Science (September 2007) 237–254 ISBN 978-3-540-75220-2.
3. 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) (January 2010) 1–6
4. 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)
5. Luís Diogo Couto, Peter Gorm Larsen, Miran Hasanagić, Georgios Kanakis, Kenneth Lausdahl and Peter W. V. Tran-Jørgensen: Towards Enabling Overture as a Platform for Formal Notation IDEs. F-IDE 2015 workshop (June 2015)
6. Jørgensen, P.W., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: The Overture 2014 workshop. (June 2014)
7. Oppitz, O.: Concurrency Extensions for the VDM++ to Java Code Generator of the IFAD VDM++ Toolbox. Master's thesis, TU Graz, Austria (April 1999)
8. Kanakis, G.: Concurrency code generator for the VDM++ Language. Master's thesis, Aarhus University, Department of Engineering (December 2014)
9. Larsen, P.G., Lausdahl, K., Battle, N.: The VDM-10 Language Manual. Technical Report TR-2010-06, The Overture Open Source Initiative (April 2010)
10. Oracle: Java platform standard edition 7 documentation. *https://docs.oracle.com/javase/7/docs/* (2014)
11. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
12. 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. Volume 6991 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer-Verlag (October 2011) 179–194 ISBN 978-3-642-24558-9.
13. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java language specification Java SE 8 edition (March 2014)
14. Larsen, P.G.: Evaluation of Underdetermined Explicit Definitions. In M. Naftalin, T. Denvir, M.B., ed.: FME'94: Industrial Benefit of Formal Methods, Springer-Verlag (October 1994) 233–250
15. Hasanagić, M.: Code Generation for Distributed Systems Modelled in VDM-RT. Master's thesis, Aarhus University, Department of Engineering (December 2014)

Generating Java RMI code for the distributed aspects of VDM-RT models

Miran Hasanagić, Peter Gorm Larsen and Peter W. V. Tran-Jørgensen

Department of Engineering, Aarhus University,
Finlandsgade 22, 8200 Aarhus N, Denmark
miran.hasanagic@eng.au.dk, pgl@eng.au.dk, pvj@eng.au.dk

Abstract. Generating code for the distributed aspects of VDM-RT, that enables modelling of distributed objects, has not been addressed before. The main challenge is to ensure that distributed objects can communicate. In order to support such communication the distributed technology Java RMI is used as part of generating the Java code. This paper presents the prime challenges and their solutions in order to generate Java code for a distributed system modelled in VDM-RT. Additionally, an example is presented in order to show how different aspects of a VDM-RT model are code generated.

Keywords: VDM-RT, Distributed System, Java RMI, Code Generation

1 Introduction

In the development of distributed systems the use of VDM Real-Time (VDM-RT) can be advantageous [16, 15]. In VDM-RT information about the distribution of functionality is only present inside the **`system`** class. Thus, the actual functionality described in the "real" classes does not need to consider where the operations to be invoked is placed in a distributed setting. It would be valuable to be able to automate the production of code from such a distributed VDM-RT model. However, this has never been enabled before but in this paper we show how this is now possible on the Overture platform [8].

In the process recommended for developing distributed embedded systems using VDM the final realisation has so far been made using manual coding [9]. Even with a manual step at the end of the development this approach may be worthwhile. However, it would naturally be advantageous if it would be possible to automate that last step. The work presented here is extending the existing code generation platform for Overture [5] with a capability for generating distributed systems [4]. This is done using Java Remote Method Invocation (RMI) to manage the distribution of functionality to different computing nodes.

After this introduction Section 2 provides the necessary background to understand how distributed systems can be represented in VDM-RT while Section 3 presents the formal semantics of the distributes aspects of VDM-RT. Aftewards, Section 4 shows how RMI can be used with Java for a distributed system, while Section 5 introduces the Overture code generation platform that is extended in the work presented here. Then the actual extension enabling the automatic generation of the distribution aspects to the

code generator is presented in Section 6. This is followed with Section 7 discussing the code generation of VDM-RT. Finally, Section 8 delivers concluding remarks and provide pointers to potential future work.

2 Distributed aspects of VDM-RT models

VDM-RT models system architecture in a special **`system`** class using language constructs for CPUs and buses. CPUs are characterized by speed and scheduling policy and allocated by objects of active classes. Execution of a VDM-RT model is initiated from a special *virtual* CPU, which is connected to every other CPU in the **`system`** class. The virtual CPU is normally used for deployment of environment processes and different from other CPUs in the sense that it executes and communicates infinitely fast.

An object is deployed on a CPU by passing it to the `deploy` operation of the CPU instance. Buses connect CPUs and enable communication at user-specified bandwidths using predefined communication protocols. Objects can invoke operations of other objects deployed on different CPUs, which causes data to be transmitted on the connecting bus. Operations are synchronous by default but can be made asynchronous.

The VDM-RT interpreter maintains a global notion of time, which is referred to using the **`time`** keyword. Simulation time progresses by a default number of nanoseconds as functions and operations are invoked. The default increase in time can, however, be overruled using the **`cycles`** and **`duration`** statements, which enable specification of execution delays relative to processor speed or as absolute time measures, respectively.

3 Formal Semantics in VDM-RT

This section briefly introduces some of the distributed aspects of VDM-RT semantics [10]. The top level structure is shown below. A VDM-RT model consists of the aforementioned CPUs and BUSses, and additionally the current time which the model has reached and classes created in the model.

$$\begin{array}{rrcl} VDMRT :: & cpus & : & CPUs = Id_c \xrightarrow{m} CPU \\ & busses & : & Busses = Id_b \xrightarrow{m} Bus \\ & time & : & Time \\ & classes & : & Classes = Id_{cl} \xrightarrow{m} Class \end{array}$$

Each CPU has three fields: the deployed instances, all threads and the execution speed. Every BUS also has three fields: the CPUs it connects, the communication speed and a queue of call and return messages, tagged with the target CPU.

$$\begin{array}{rrcl} CPU :: & objects & : & Id_o \xrightarrow{m} Object \\ & threads & : & Id_t \xrightarrow{m} Thread \\ & speed & : & \mathbb{N}_1 \end{array}$$

$$\begin{array}{rrcl} Bus :: & cpus & : & Id_c\textbf{-set} \\ & speed & : & \mathbb{N}_1 \\ & queue & : & (Id_c \times (CMessage \mid RMessage))^* \end{array}$$

The actual semantics is provided in a Structural Operational Semantics (SOS). The big step rule for the evaluation of a VDM-RT model is shown in figure 1[1]. Line two in figure 1 shows that messages are delivered on the busses to their target CPU, when two CPUs communicate.

Big Step

$$\frac{\begin{array}{ll} vdmrt_1 = commitPendingValuesAndUpdateTime(vdmrt, \tau) & (1) \\ vdmrt_1 \overset{busses}{\longrightarrow} vdmrt_2 & (2) \\ vdmrt_3 = createPeriodicThreads(vdmrt_2) & (3) \\ vdmrt_4 = doContextSwitches(vdmrt_3) & (4) \\ vdmrt_4 \overset{exec}{\longrightarrow} (vdmrt_5, \tau_b) & (5) \\ \tau_b' = min(\tau_b, minPendingCommitTime(vdmrt_5)) & (6) \end{array}}{(vdmrt, \tau) \overset{vdmrt}{\longrightarrow} (vdmrt_5, \tau_b')}$$

Fig. 1. Definition of the Big Step rule.

In [10] both a local and a remote call are described using SOS. However, in this section the focus is purely on the remote calls, which generates network traffic, because the focus is on the distribution aspect. The information is queued on the bus as messages.

Call messages are $CMessage$ constructs, while return messages are $RMessage$ constructs. The constructs for $CMessage$ and $RMessage$ are shown below, respectively. The $CMessage$ contains information about the target object, target operation in that object, operation arguments, identifier of the CPU and thread of the caller and when the message is send. The $RMessage$ contains the value returned by the operation, the identifier of the CPU and thread of the caller and the send time.

$$\begin{array}{lrcl} CMessage :: & obj & : & Id_o \\ & op & : & Id_{op} \\ & args & : & VDMValue^* \\ & replyto & : & [Id_c \times Id_t] \\ & sendTime & : & Time \end{array}$$

$$\begin{array}{lrcl} RMessage :: & value & : & VDMValue^* \\ & replyto & : & Id_c \times Id_t \\ & sendTime & : & Time \end{array}$$

Figure 2[2] illustrates abstractly how a remote synchronous call is performed. This figures shows that a remote call is handled in the same way as a local call, which is described subsequently. For the remote synchronous call the most important SOS is shown in figure 3[3]. The lines 7-10 in figure 3 describe that a created $CMessage$ is added to the bus, and the status of the calling CPUs thread is changed to WAITING, in order to wait for the return of the remote operation call. For an asynchronous remote call the calling CPU does not wait for the remote call to complete.

[1] This figure is borrowed from [10]
[2] This figure is borrowed from [10]
[3] This figure is borrowed from [10]

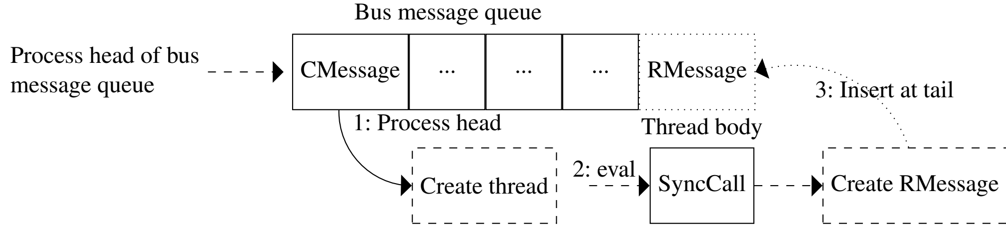

Fig. 2. Illustration of the semantic evaluation of a *CMessage* from the bus queue.

Stmt Call Op Remote Sync

$$opTarget = (ccpu, oid, op) \tag{1}$$
$$argsTimed = [(value, \delta_e) \mid arg \in args \land (classes, cpus, pending, o \vdash [\![e]\!] = (value, \delta_e))] \tag{2}$$
$$args = [value \mid (value, \text{-}) \in argsTimed] \tag{3}$$
$$mk\text{-}Op(\text{-}, params, ret, body, pre, post) = classes(cpu.objects(oid).class).ops(op) \tag{4}$$
$$rest' = [mk\text{-}Wait(target)] \curvearrowright rest \tag{5}$$
$$busses(bus) = mk\text{-}Bus(\{ccpu, c\} \cup connected, speed, queue) \tag{6}$$
$$cmsg = mk\text{-}CMessage(oid, op, args, (c, t), \tau) \tag{7}$$
$$busses' = busses \dagger \{bus \to mk\text{-}Bus(\{ccpu, c\} \cup connected, speed, queue \curvearrowright [(ccpu, cmsg)])\} \tag{8}$$
$$cpu' = changeThreadStatus(cpu, t, \textsc{Waiting}) \tag{9}$$
$$\delta' = sum([\delta_e \mid (\text{-}, \delta_e) \in argsTimed]) + RemoteSyncCallTime \tag{10}$$

$$\tau, classes, cpus, c, t, o \vdash$$
$$([mk\text{-}SyncCall(target, opTarget, args)] \curvearrowright rest, pending, cpu, busses) \xrightarrow{stmt} (rest', pending, cpu', busses', \delta')$$

Fig. 3. Definition of the Stmt Call Op Remote Sync rule.

4 Network communication using Java RMI

Java RMI [13] enables two objects located on different Java Virtual Machines (JVMs) to communicate transparently, e.g. as if they are located on the same JVM. This technology enables both to send primitive types, objects by value and objects by reference. In order for an object to be sent by reference it has to be a `UnicastRemoteObject`, and has to be instantiated from a class that implements a corresponding interface which extends some Java RMI properties. The methods defined inside this interface are accessible remotely.

A JVM has to obtain a reference to a remote object in order to invoke its methods defined by the interface. This can be achieved by using a registration service in which objects can be register by a unique identification (ID). Such a service is provided by the RMI registry [2], which will be used as the registration service.

The roles of two objects communicating using Java RMI are a client-server relationship. The invoking object is acting as a client, while the invoked object is acting as a server. The invoked object delivers a service in the form of a method that possibly returns a value to the client object.

An important part in order to establish a network connection using Java RMI is a registration service, which will be referred to as a RMI registry, in which remote objects are registered with an ID. Then every object which must be invoked outside its own address space has to be registered in the RMI registry. When a client object needs to invoke a method of a remote object, it first looks up the remote object in the RMI registry using its unique ID of the remote object. The RMI registry then returns a proxy to the remote object, and afterward the client object can use to this proxy to invoke methods of the remote object as if it is a local object.

The overall process of establishing communication between a client object and a server object is illustrated in figure 4. This figure illustrates that all network communication between the entities is handled by Java RMI, and the following steps described are from this figure. In step 1 the Server connects to the RMI registry using Java RMI, and registers an object in the RMI registry. In this step the server has the role of a client and the RMI registry has to role of a server. Afterwards, in step 2 the Client connects to the RMI registry, and looks the objects it wants a remote reference to up. In this step the Client has the role of a client and the RMI registry has the role of a Server with respect to the client-server model. Finally, in step 3 the Client can invoke a method on the remote object as if the remote object is in its local address space.

In order to establish a Java RMI connection between a client object and server object, the following has to be created: An communication contract between the objects and the implementation of the server object and the client object. The details of each step is presented below.

Define a remote contract: When two objects have to communicate, they need to agree on which "services", e.g. method name(s) and possible parameter(s) as well as a possible return value for the method(s), that the server object provides to the client object. This contract is implemented as an **`interface`** in Java, and both the server and client need to have access to this it.

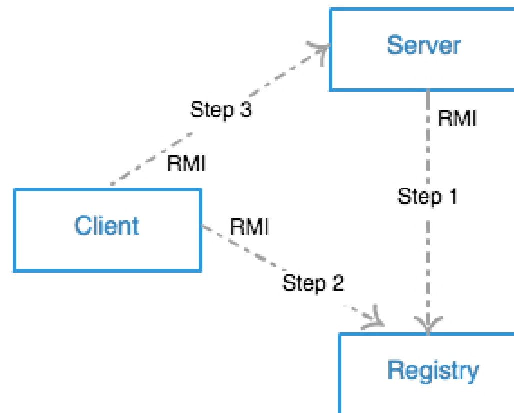

Fig. 4. Example showing how communication between a client and a server is established using Java RMI.

Server object: The methods defined in the contract need to be implemented in a **`class`**. Then an objects is instantiated from this **`class`**. Furthermore, the object has to be registered in a RMI registry with a unique ID.

Client object: The client needs to know the correct ID of the server object in the RMI registry to create a stub of the remote object. After the client has obtained a stub, it can invoke the methods of the remote object, which are defined in the "contract". The contract is used to create a stub on the client side, when communicating with the RMI registry.

The above indicates that an object instantiated on a JVM is a Java class type, while a remote object located on another JVM is represented by a Java interface type.

5 Extending the Overture Code Generation Platform

The VDM-RT code generator extends the existing VDM++ code generator with support for code generation of deployment and remote invocation of functions and operations. Since the VDM++ code generator is developed as a backend extension to the Overture code generation platform, developing code generation support for VDM-RT has involved working with this platform.

5.1 The code generation platform

Overture uses a code generation platform to make it easier to contribute code generation support for new target languages. To do this, the code generation platform constructs an Intermediate Representation (IR) of the generated code from the VDM Abstract Syntax Tree (AST). The IR is independent of any particular target language and therefore functionality for interacting with the IR can in principle be shared among code generators,

or backends, of different target languages. The workflow of extending the code generation platform follows a step-wise process where the IR is gradually transformed into a structure, which is easy for a backend to generate code from. This is done by applying a series of transformations to the IR each of which are implemented by subclassing the base class visitors [3] provided by the code generation platform.

When the IR has reached a form suitable for code generation, mapping each of the IR nodes into code in the target language should ideally be a trivial task. To facilitate the process of producing code in the target language the code generation platform provides a small syntax generation framework. This framework provides functionality for mapping each IR node supported by the backend into syntax in the target language.

5.2 Extending the Intermediate Representation

The IR defines an AST that can be extended with support for additional IR nodes. The code generation platform also enables the IR to be transformed using a visitor based approach, which is convenient as it allows the backend to rewrite the IR into a form that is easier for a backend to code generate.

In this work we take advantage of the possibility to extend the IR with new nodes: Constructs such as the remote registry service and the remote interface describe generic concepts that exist in some form or another in RMI based middleware technologies such as Java RMI, CORBA or ICE [12, 17]. In order to allow the RMI paradigm to be represented at the IR level, new nodes have been added to represent some of the important concepts from the RMI paradigm.

RMI based technologies also distinguishes between types used to represent objects that are accessed locally and remotely. Since VDM-RT does not distinguish between types of remote and local objects we use a transformation to substitute local class types with the equivalent remote interface in order to guarantee that remote objects can be passed as arguments in the generated Java code.

5.3 Extending the Java backend

Based on the existing IR we instantiate an extended IR that holds statically derived information about the system architecture and object deployment. Subsequently the extended IR is handed over to the Java backend, which uses the syntax generation framework of the code generation platform to finalise the code generation process by mapping each of the IR nodes into Java source code. The syntax generation framework is based on Apache Velocity template based technology [14]. This framework stores the syntax for each of the supported IR nodes in template files and enables access to information about an IR node and its context using the Velocity Template Language.

Finally, to also provide code generation support for the IR nodes introduced as part of this work, additional templates have been added to the Java backend. These templates represent node construct such as the remote interface and make use of Java RMI to map each of the nodes into Java code.

6 Generating Code for VDM-RT models

The Code Generator (CG) presented in this paper extends the existing VDM++-to-Java CG that is part of the Overture platform [6, 5]. This VDM++-to-Java CG is used in order to generate Java code for the functionality of a single CPU, while the CG presented in this paper enables network communication between objects. This is possible since the collection of objects deployed to a CPU can be viewed as a single VDM++ model, which possibly depends on objects located on another CPU. In addition, since both Java RMI and VDM-RT are based on the RMI communication paradigm, Java RMI is a good first choice for a distribution technology. Subsequently the term VDM method refers to both VDM functions and operations, because Java only has methods.

The code generation process of a VDM-RT model can be divided into two main parts:

1. Generating the static VDM-RT model and preserving its semantics, when using Java RMI for supporting network communication.
2. Ensuring that the generated Java code can start execution similar to the VDM-RT interpreter.

The following two subsections discuss and present how these two main parts of the code generation process can be solved, respectively.

6.1 Static VDM-RT model

Extracting distribution information from a VDM-RT model
As described in section 2, the distributed aspects of VDM-RT are modelled inside the **`system`** definition. For this CG the information necessary from the **`system`** definition is which instantiated objects are deployed to which `CPU` and how the `CPU`s are connected. This information will be referred to as a Deployment Map (DM) and a Connection Map (CM), respectively.

The DM can be extracted by analysing the deployment of objects inside the constructor of the **`system`** definition, as shown in the example in listing 1.1. In addition, the CM can be extracted by analysing the bus structure, as shown in listing 1.2. Hence the DM and CM for this example are:

`CPU` name	DM	CM
cpu1	{a1, a2}	{cpu2, cpu3, cpu4}
cpu2	{b1}	{cpu1}
cpu3	{a3}	{cpu1, cpu4}
cpu4	{b2}	{cpu1, cpu3}

Table 1. Example of a DM and CM for the VDM-RT listings 1.1 and 1.2.

The DM and CM provide the CG with enough information in order to generate Java RMI code for the distributed aspects of a VDM-RT model. However, it shall be noted

that realising the real bus structure of a VDM-RT model is not possible when Java RMI is used for the communication, because all generated JVMs are connected by a single ethernet connection. So it is enough to know that two CPUs are connected when extracting the CM, because a multi bus architecture cannot be supported by this CG.

```
1  ...
2  public C: () ==> C
3  C() ==
4   (
5    cpu1.deploy(a1);
6    cpu1.deploy(a2);
7    cpu2.deploy(b1);
8    cpu3.deploy(a3);
9    cpu4.deploy(b2);
10  );
11 ...
```

Listing 1.1. Example of the constructor for a **system** definition in VDM-RT called C.

```
1 ...
2   -- CPUs are connected
3   bus1 : BUS := new BUS(<FCFS>, 1E3, {cpu1, cpu2});
4   bus2 : BUS := new BUS(<FCFS>, 1E3, {cpu1, cpu3, cpu4});
5 ...
```

Listing 1.2. Example of a BUS structure inside a **system** class in VDM-RT.

Code Generating VDM-RT classes

When code generating a VDM-RT class, it has to be ensured that an object of it is accessible both locally and remotely with respect to a CPU. Hence in order to discuss the difference between a model and generated code, the notions of *local* and *remote* objects with respect to a CPU in a VDM-RT model are defined. A *local* object of a CPU is an object deployed to it. A *remote* object of a CPU is an object deployed to a connected CPU. Local and remote objects can be identified by the CG by using the information provided by the DM and CM.

In VDM-RT both local and remote objects are instances of the same class. However, in Java RMI a local object has a class type, while a remote object is defined as a Java interface (remote contract) type as described in section 4. As a consequence all VDM-RT classes are required to be generated to both a Java class and a corresponding remote contract. The Java class has to contain all the functionality of the VDM-RT class, while the remote contract is required to contain the public method signatures, because only these can be invoked outside the object. Each Java class then implements

its corresponding remote contract. The CG generates the remote contract for a VDM-RT class according to the convention using a fixed postfix: ***VDM_class_name*** _i. For example the VDM-RT class A will be generated to both a Java class called A and an interface called A_i. The CG ensures that the class A implements the interface A_i.

The CG can not know which VDM-RT methods will be invoked without interpreting the model, so it has to make all the public VDM-RT methods accessible for each VDM-RT class by adding them to the corresponding remote contract. Additionally, this approach allows the designer to use these methods afterwards in the generated Java code for an object, even though they were not used as part of the interpretation of the VDM-RT model.

Transformation of method parameters and return values

A challenge for the CG, as a consequence of Java RMI having different representation of local and remote objects, is when using an object as a return value or a parameter value of a method. The CG is required to cope with this challenge in order to support both the use of remote and local objects as parameters of a method.

The CG could support this by using method overloading when a method takes objects as parameters or a return value. However, this approach can generate many overloaded methods from one VDM method, if it contains many parameters of a class type.

Another approach could be to exploit the possibility in Java, that if a parameter of a method is an interface type, it is allowed to pass a class that implements that interface also. Since the CG ensures that every class has a corresponding interface, it is sufficient for a Java method to take the interface instead of the Java class. For example if the parameter of a method is a class type A in a VDM-RT model, then it is transformed to A_i in the generated code. However, it will not be possible to access the public variables of an object directly due to the limitation of using Java RMI, which is, however, possible in a VDM-RT model. In the generated Java code all objects from a VDM-RT model are required to be UnicastRemoteObjects. This ensures that all objects are send by their reference between JVMs when using Java RMI, which is also the case in a VDM-RT model between CPUs. The construction of the send parameters in Java RMI corresponds to the construction of the *CMessage* construct described in section 3.

Generating functionality of a single CPU

Each CPU in a VDM-RT model is generated as an individual JVM, which subsequently will be referred to as a realised CPU. This raises a challenge for the CG, because the **system** definition is globally visible for all CPUs in a VDM-RT model. A possible solution is to generate a Java class in order to represent the **system** defintion for each realised CPU, which makes the local and remote objects globally accessible for a realised CPU according to the VDM-RT model. As an example for the realisation of cpu2 shown in table 1, the generate Java class representation of the **system** defintion is shown in listing 1.3. In this listing the object called b1, is a class type A since it is a local object for cpu2. Since cpu2 is connected to cpu1, it has access to the objects a1 and a2, which are an interface type A_i because they are remote with respect to cpu2. This approach ensures that the distributed objects inside the **system** definition are globally accessible with respect to a realised CPU, just as in a VDM-RT model.

Local objects have not been instantiated and references to remote objects have not been obtained yet. The following subsection addresses this problem, and additionally proposes a solution to it. However, until now the preservation of the VDM-RT semantics has been addressed, without setting the references of local and remote object up, because they are a concrete design decision made by the CG.

```
...
public class C {
    public static A_i a1 = null;
    public static A_i a2 = null;
    public static A b1 = null;
}
...
```

Listing 1.3. Local Java class in order to represent the **system** definition for cpu2 as shown in table 1.

6.2 Generating code for the interpreting of a VDM-RT model

A CPU is initialised when it has instantiated all its local objects, and obtained a reference to its remote objects. The VDM-RT interpreter initialises all CPUs before the entry method of the VDM-RT model is interpreted. This is required as a consequence of the ability to use the distributed objects at any place and time during model interpretation. Figure 5 illustrates a potential error sequence during model interpretation, if all CPUs have not been initialised. For this reason, the CG has to provide a similar initialisation mechanism.

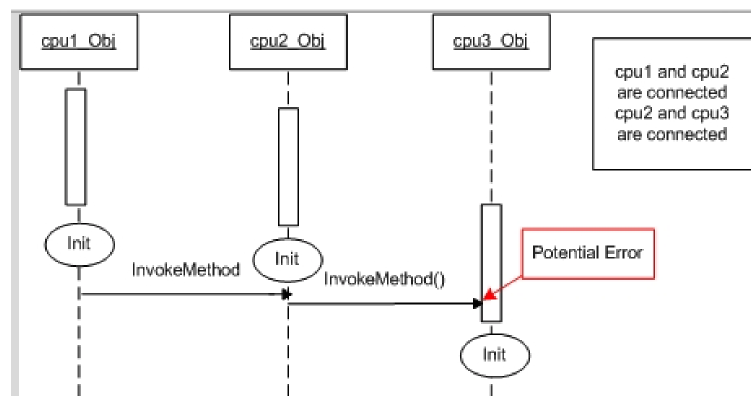

Fig. 5. Global dependencies error

This CG generates a simple initialisation mechanism when generating code for the whole distributed system, which is shown in figure 6. The main assumption for this algorithm is that all realised CPUs are connected to and store their local objects in the same registration service during the initialisation. Additionally, a realised CPU can receive the total number of registered objects in the registration service in order to support the two decisions shown in figure 6. Then the steps in this initialisation algorithm, shown in figure 6, can be described as:

1. Each CPU registers its own local objects, and waits until all distributed objects from the **`system`** definition have been registered in the registration service. For this part the CG exploits the fact that it knows the total number of distributed objects inside the **`system`** definition, when generating the code for each CPU.
2. Each CPU obtains references to remote objects with respect to the VDM-RT model. Afterwards it registers an additional object, called a `SyncToken`, in the registration service in order to indicate that it has obtained all required references. Finally, every CPU waits until all CPUs have obtained their remote references by waiting until the total number of objects in the registration service is equal to the total number of distributed objects plus the number of CPUs.
3. Each CPU can start its individual main execution.

The third step indicates that each realised CPU has it own main execution, while on the other hand a VDM-RT model has a single entry point as described in section 2. This is addressed in the following subsection.

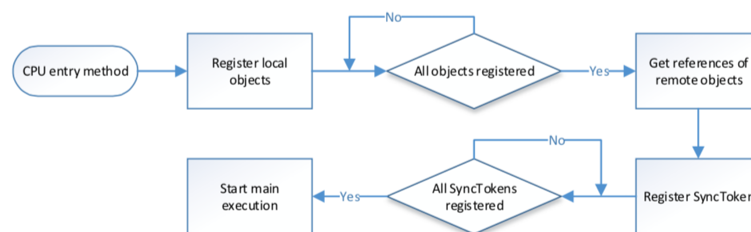

Fig. 6. Initialisation algorithm for each realised CPU

Entry Method of a VDM-RT model

In VDM-RT a global entry method from a class has to be set in order to interpret the model. Afterwards, the interpreter creates an instance of this object on the virtual CPU and starts this method. This raises a challenge for the CG, since the virtual CPU on purpose is not generated to code, because it usually is used in order to model the expected environment and store test results. In order to cope with this challenge a natural limitation, in order to generate all Java code for the functionality of a distributed system, to the modeller is put: The functionality of the modelled distributed system is not allowed to depend on objects deployed to the virtual CPU.

If the designer only uses objects deployed to real CPUs inside the entry method of a VDM-RT model, these objects can be moved to the main execution of the realised CPU they are deployed to. As an example if the entry method for a VDM-RT model is as shown in listing 1.4, and the object called `b1` is deployed according to listing 1.1, then the code generated method `b1.HelloWorld()` is moved to the main method execution of the realised `cpu2`. Currently this has to be carried out manually after the code generation, but could easily be supported by the CG automatically by using the DM.

```
...
public startDS: () ==> ()
startDS() ==
 (
  b1.HelloWorld();
 );
...
```

Listing 1.4. Example of an entry method for a VDM-RT model

7 Discussion

7.1 VDM-RT Semantics for Distribution and Java RMI implementation

Both the VDM-RT model and the generated code that uses Java RMI have some commonalities and differences, that are important to address in order to understand the challenges when realising the distribution aspects. Both af them support access to remote objects transparently. Before the main execution starts, both are required to have some kind of initialisation mechanism. Both send instantiated objects by their reference. However, the direct access to public variables of a class and the multiple bus structure cannot be supported in the generated code.

In order to preserve the semantics of the distributed aspects of VDM-RT as described in section 3, the chosen Java RMI technology has to follow the same formal semantics. An important difference is that Java RMI only uses a single BUS in order to connect all the JVMs, while a VDM-RT model may consist of more busses. However, the implementation of the **`system`** definition ensures that each CPU, a JVM, only has access to the same objects as in the VDM-RT model. Additionally, a remote call forces the calling thread on a CPU to wait for the remote object to execute its method on the remote JVM. Hence from this point of view both the VDM-RT model and Java RMI implementation behave in the same way.

As a consequence of the lossness in a VDM-RT model, multiple valid interpretations of the model may exists. However, the interpretation of a VDM-RT model is on purpose deterministic even when modelling concurrency aspects [11]. Due to the lossness in a VDM-RT and that the execution of Java code is non-deterministic, a CG may generate Java code which may not follow the same execution path for each simulation

of the same test case. However, each execution of the generated Java code shall be one of the possible interpretations of a VDM-RT model. Thus it is acceptable that there will be differences between the interpreter and the generated code for the distributed system.

7.2 VDM-RT Code Generator challenges

During the work with generating code for a VDM-RT model, a limitation of the VDM-RT notion has been identified, which is related to modelling a distributed system in which two `CPU`s are connected by multiple `BUS`ses. In such a case VDM-RT does not define which `BUS` a `CPU` has to use for communication, but picks an arbitrary `BUS` deterministically. Then VDM-RT may provide a wrong feedback about the real time aspects to the designer, if the `BUS`ses have different communication speeds. Hence this issue in VDM-RT also is relevant for a code generator. This is something where the VDM-RT notion needs improvement. Additionally, global access to variables should only be for the distributed objects declared inside the **`system`** definition. This is something that maybe should be changed in the VDM-RT language, in order to only make distributed objects inside the **`system`** definition globally accessible.

The work described in this paper focuses on code generation of the deployment aspects of VDM-RT and remote communication between objects deployed on different CPUs. We do not, in the current version of the VDM-RT code generator, take timing aspects into accounts. From this it follows that the code generator also does not support periodic threads, which rely on timings information. The code generator does, however, provide code generation support for all the concurrency mechanisms present in VDM++ as enabled by the work in [7]. Finally, the code generator does not support remote access to instance variables or values, which can easily be circumvented using accessor operations.

8 Concluding Remarks and Further Work

The work presented in this paper addressed the challenges how to generate code for the distributed aspects of a VDM-RT model, which is a novel area of research. This paper presented challenges and their solutions when using Java RMI in order to support the network communication in a code generated distributed system. Additionally, some limitations to both the code generator and the VDM-RT notion have been addressed and discussed.

Further work may include to generalise the solutions presented in this paper, in order to use another technology which is based on RMI, such as CORBA. In addition, supporting the time aspects of both a `CPU` and a `BUS` in a VDM-RT model, during code generation can be addressed. This may include to generate code towards concrete CPUs. Then the code generator can generate code with time aspects, if it knows which CPU it generates code for.

In [1] different parts of Java RMI are formalised using operational semantics. Hence future work may include to compare and proof in which cases the semantics of distribution are the same for both VDM-RT and Java RMI.

The focus of this paper was to code generate a static distributed system, where new connections and entities can not be introduced during run-time, modelled in VDM-RT. However, future work can include research for code generating dynamic distributed systems, where new connections and entities can be introduced during run-time.

Acknowledgments. The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047.

References

1. Ahern, A., Yoshida, N.: Formalising java rmi with explicit code mobility. OOPSLA (October 2005)
2. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed Systems: Concept and Design. Addison-Wesley (May 2007)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, R.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
4. Hasanagić, M.: Code Generation for Distributed Systems Modelled in VDM-RT. Master's thesis, Aarhus University, Department of Engineering (December 2014)
5. Jørgensen, P.W., Couto, L.D., Larsen, M.: A Code Generation Platform for VDM. In: The Overture 2014 workshop (June 2014)
6. Jørgensen, P.W., Larsen, P.G.: Towards an Overture Code Generator. In: The Overture 2013 workshop (August 2013)
7. Kanakis, G.: Concurrency code generator for the VDM++ Language. Master's thesis, Aarhus University, Department of Engineering (December 2014)
8. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (January 2010), http://doi.acm.org/10.1145/1668862.1668864
9. Larsen, P.G., Fitzgerald, J., 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)
10. Lausdahl, K., Coleman, J.W., Larsen, P.G.: Semantics of the VDM Real-Time Dialect. Tech. Rep. ECE-TR-13, Aarhus University (April 2013)
11. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (October 2011), http://dl.acm.org/citation.cfm?id=2075089.2075107, ISBN 978-3-642-24558-9
12. OMG: The Common Object Request Broker: Core Specification. (November 2002)
13. Sun: Java Remote Method Invocation Specification (2000)
14. The Apache Velocity website (2015), http://velocity.apache.org/
15. Verhoef, M.: Modeling and Validating Distributed Embedded Real-Time Control Systems. Ph.D. thesis, Radboud University Nijmegen (2009)
16. 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)
17. ZeroC website (2014), http://www.zeroc.com/

Improving Time Estimates in VDM-RT Models

Morten Larsen[1,2], Peter W. V. Tran-Jørgensen[1], and Peter Gorm Larsen[1]

[1] Department of Engineering, Aarhus University, Denmark
{mola,pvj,pgl}@eng.au.dk
[2] Conpleks Innovation ApS, Denmark.

Abstract. Choosing the best hardware platform for an embedded system can be difficult – especially when the success of the system relies on timing requirements. To address this, timing analysis is often used to determine how well a system performs on given hardware platforms. In VDM-RT models it is possible to specify the time it takes for a functional description to execute on a CPU that runs at a user-defined speed. This timing information is particular important when it is used in a Crescendo setting where the progress of time has significant impact on the interaction with the physical environment. This paper illustrates how performance estimations can be improved using timing information obtained by executing code generated from a VDM-RT model. We measure the time it takes to execute code generated functional descriptions on a specific hardware platform and incorporate this timing information back into the model. This increases the fidelity of the model simulation since the timing information is based on executing a real software implementation. We believe that for computation-extensive algorithms, this approach can be a valuable way to determine the best use of different hardware platforms.

Keywords: VDM, code generation, Timing analysis, C++

1 Introduction

Time often plays an important role in embedded system development and therefore embedded system models often incorporate timing information to reason about system performance. To estimate the execution time of algorithmic constructs, the VDM-RT interpreter enables timing information to be inserted into the model [7]. Subsequently, the model can be simulated to produce a log file containing all the time-stamped execution events such as operation invocations, swapping of threads etc. The log file can then be analysed visually using the RT Log Viewer and timing requirements can be validated against the log file [9].

VDM++ [4] extends the ISO standardised VDM-SL [3] with object-orientation and mechanisms for modelling concurrency. VDM Real Time (VDM-RT) further extends VDM++ with support for modelling of distributed embedded systems and introduces a global notion of time. The system architecture in VDM-RT model is modelled using special classes for CPUs and busses. CPUs are characterised by speed and scheduling policy and allocate objects of active classes. Busses connect CPUs and enable communications at user specified speed and protocols.

In VDM-RT execution of an algorithmic construct progresses time by a default number of nanoseconds. The default time delays can, however, be overruled using the cycles and duration statements, which enable specification of execution delays relative to the processor speed or as an absolute time measure, respectively.

If the system must adhere to strict timing requirements the model can be annotated with detailed timing information and simulated to check if it is able to meet the timing requirements. For large models, however, manual insertion of timing information into the model is a tedious task. To address this, we propose a technology for automatically annotating models with timing information based on measurements obtained by executing a code generated version of the model. More specifically, we measure the average time it takes for the code generated functions and operations to execute, and we use this information to time annotate the corresponding functions and operations in the model. This approach enables us to use a more time accurate version of the model to make predictions about the time behaviour of the final version of the system. Furthermore, the timing measurements obtained by executing the code generation version of the model reflect the performance of the underlying hardware platform. Therefore, we can use the outcome of simulating the time annotated model to compare different hardware platforms against each other.

A similar approach is taken in [1], where an UML state chart model is translated into a model suitable for timing analysis, enabling design time exploration of the timing performance of the model. We believe that our approach is more general due to the expressiveness of the VDM-RT modelling language. Furthermore the support for both simulation based and measurement based timing analysis in an automated setting enables more detailed analyses to be carried out. Lastly we note that the ability to analyse a model annotated with both measured, simulated and best guesses offers flexibility during the design phase where many parts of the system is unspecified.

Structure of the paper Section 2 provides a brief introduction to the Overture platform. Section 3 describes how the Overture platform has been extended to support automated time annotation of VDM-RT models. Section 4 presents the case study, used for evaluating the extension to the overture platform. Section 5 provides an overview of how timing estimates has been extracted for two different hardware platforms. Section 6 summarises the case study timing results and experiments. Section 7 continues with a discussion about the applicability of this approach in a Cyber-Physical Systems (CPSs) setting. Finally, we present ideas for future work and conclude in Section 8.

2 The Overture platform

Overture works like most other modelling tools. First, the parser constructs an internal representation of the model as an Abstract Syntax Tree (AST), and subsequently every component interacts with the AST in some fashion: The type checker analyses the model and reports errors to the user, the interpreter evaluates it and so on.

The Overture AST is generated using the ASTCreator tool. In addition to the nodes, ASTCreator also provides mechanisms to walk the tree using the visitor pattern [5]. The generated nodes also have functionality for manipulating the tree structure. This allows

parts of the tree to be replaced or new nodes can be inserted into the tree structure. In our work, we implement visitors to traverse the VDM AST and annotate the model with timing related information. The details of how we do this are provided in subsection 3.3.

Below we describe how we have extended Overture with support evaluating the performance of a VDM-RT model based on timing information derived from a code generated version of the model. First, we outline the overall structure of the approach, then we describe how we obtain the timing information, and finally we explain how we use visitors to annotate the model with this timing information.

3 Extending the Overture Platform

The overall approach showing the two proposed prototype extensions and the relation to the Overture tool is shown in figure 1.

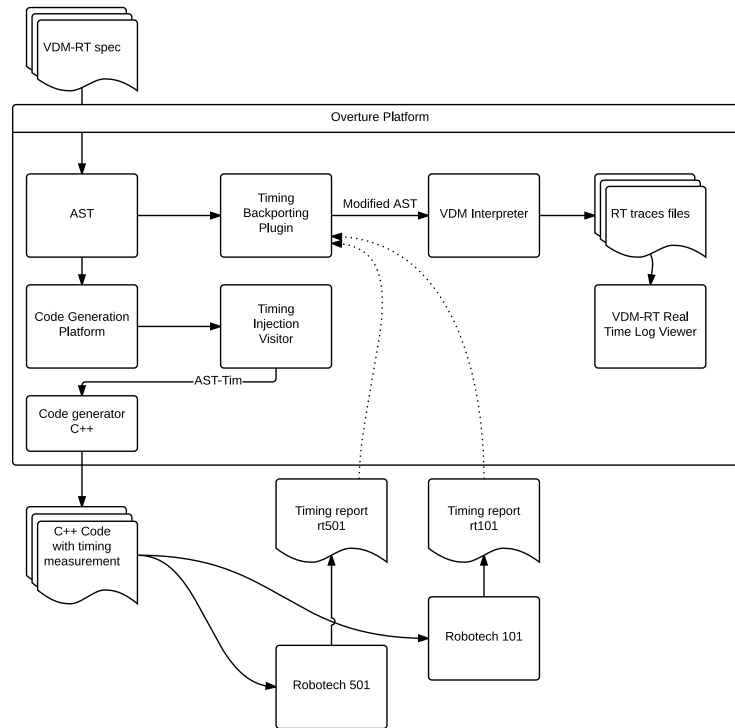

Fig. 1. An overview of the process for annotating a VDM-RT model with timing information based on measurements obtained from executing a code generated version of the model.

We have extended the prototype C++ code generator [6] with support to optionally insert code that measures the time it takes to execute user-selected functions and operations in the generated code. The timing results produced by executing the generated

code instrumented with timing measurements is then inserted into the VDM-RT AST by injecting it using **duration** statements. This is done for all the operations and functions selected by the user.

The process of inserting the **duration** statements is transparent to the user. These modifications are applied internally to the VDM-RT AST just before it is executed by the VDM-RT interpreter. The execution of the VDM-RT model produces a trace file with all the time-stamped execution events, which can be further examined using the VDM-RT Real Time Log Viewer [9].

3.1 Obtaining estimates of timing

Measuring the timing of a program running on a general purpose computer can be complicated due to both hardware optimisations such as caching, branch speculation, and the presence of an operating system preempting tasks, handling system interrupts, etc. Several methods exists for obtaining the execution time of a program. These methods can be divided into two categories: the *measurement* based and the *analysis* based [12] methods.

Measurement based methods typically instrument the source code or the program binaries with functionality to do the timing measurements. When the program is executed the entry and exit time of each function call is recorded using low overhead logging. Afterwards a trace is produced which can then be inspected either graphically or manually. The program is executed on real hardware or using a simulator, depending on the specific method being used. The advantage of using a measurement based method is that it can easily be utilised on many kinds of hardware platforms. To this end we note that running the VDM-RT interpreter and generating a VDM-RT trace file is also a measurement based method, where the hardware is represented using VDM-RT CPU and BUS classes.

The static analysis based methods, analyses the provided source code in order to obtain upper bounds on the execution time of a function or program. Typically the static analysis tools are employed in the context of embedded systems requiring hard real time guarantees. However some of the platforms targeted by the Overture tool are general purpose computers with a general purpose operating system, where no guarantee on real time performance is given.

Each of the methods has their own advantages and disadvantages, the main focus of our work is to obtain timing estimates on various hardware platforms. We select the timing based method, since it can be used for all the platforms the code generator can target.

Although commercial tools for measuring the execution time of a program such as rapiTime [8] exist, we have chosen to implement our own small C++ library to support time measurement of operations and functions. This library defines the `TimedScope` C++ class, which can be instantiated in the beginning of a method generated from a function or operation. Instantiating the `TimedScope` class, causes the class constructor to be invoked, which records the current time in a buffer along with a method ID. When the method returns the `TimedScope` object goes out of scope. This causes the class destructor to be invoked, which again records the current time into the buffer.

In order to calculate the execution times for each method invocation the time-stamps recorded upon entering and leaving a method are subtracted. Currently the library does not record which object the method is invoked for. This may, however, be an useful extension that will make the log file produced by the generated C++ code be more similar to the trace file generated by the VDM-RT interpreter.

3.2 Extending the Overture C++ code generator

We have extended the C++ code generator with a visitor responsible for inserting invocations to the timing library described above into the generated code. For each user-chosen function and operation a statement is inserted at the beginning of the corresponding method, which instantiates the `TimedScope` class with an ID. When the visitor is done inserting the time measurement statements a mapping of names to IDs is generated. This allows names and IDs to be associated when back porting the timing information into the model. The ID is derived from the method and the name of the class enclosing the method. The name generator also takes the argument types and the return type of the method into account to guard against overloaded functions and operations in the VDM-RT model.

3.3 Injecting duration statements

In order to obtain the timing information and inserting it into the VDM-RT AST an ID is constructed for each function and operation in the VDM-RT model. The construction of the ID follows the same approach as described in subsection 3.2. If the ID appears in the log file generated by the timing instrumented C++ program, the mean value is loaded and a **`duration`** statement is created and inserted into the corresponding function or operation in the VDM-RT AST.

4 Case study

In order to demonstrate our work we apply it to an algorithm for detecting rows of mink cages in a farm environment. The design questions investigated is to determine 1) how the row detection algorithm performs with respect to timing on two different hardware platforms and 2) the resulting timing related trade-offs when selecting the parameters of the row detection algorithm. Even though the main objective of the algorithm is to detect the rows of mink cages, a second, yet equally important objective, is to do the row detection within the shortest time possible.

In this case study, the rows detected by the algorithm is used for navigating the rows autonomously, which requires the information be updated regularly in order for the robot to steer the vehicle accurately down the rows. Obtaining the timing information for the individual operation calls in the row detection algorithm allows a trade-off between row detection accuracy and execution time (hardware platforms) to be investigated. Since the timing information is back ported into the model, exploring the design space can solely be done in a modelling environment.

The procedure taken for the development of this algorithm is to first collect the sensor data from the farm environment, then perform an initial tuning of the proposed algorithm such that the desired rows are detected, then determine the execution time of the tuned algorithm on different hardware platforms, using the two extensions for the Overture tool described in this paper.

4.1 Environment

The data for the algorithm has been collected from a mink farm environment, see figure 3. An example of the data collected from the LiDAR (Light Detection And Ranging) can be seen in figure 2. The LiDAR uses a rotating laser beam to measure the time of flight to reflecting targets and outputs a list of angles and an associated distance measured.

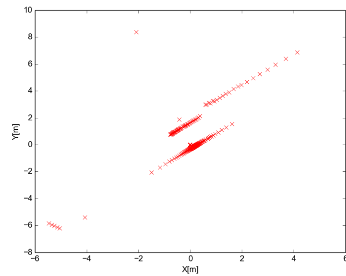

Fig. 2. LiDAR data plotted as X,Y points. The LiDAR is placed at an 45 degree angle relative to the row

Fig. 3. Image of the farm environment, the LiDAR is placed on the robot such that the vertical wooden boards are scanned.

The dataset used for our experiment consists of 137 scans which have been sampled from the farm environment. These scans contain both samples from the rows themselves but also from the area outside of the rows.

4.2 Proposed algorithm for row detection

The algorithm used in this case study is based on RANSAC [2] for finding the best matching fit to a model from a noisy data set. In our case the model is that of a line represented by a point on the line and a direction. The RANSAC algorithm for line detection selects a line based on two randomly sampled points from the set of points in the scan and counts the number of points within a certain distance to the selected line. This process is repeated for a selected number of iterations. The VDM-RT model of the RANSAC algorithm is shown in in Listing 1.

```
1  public extractLines : seq of Point`PointM ==> seq of fit
2  extractLines(points) == (
3    reset();
4    while cur_iter < max_iter do (
5      let cur_l = getRandomLine(points),
6      inliers = getInliers(points, cur_l)
7      in
8        if(len inliers > cur_n_inliers) then
9          addNewBestFit(cur_l, inliers);
10     cur_iter:= cur_iter + 1; );
11   return fits;);
```

Listing 1. The main operation of the modified RANSAC algorithm

The RANSAC algorithm shown in Listing 1 is extended with an outer loop which repeats the algorithm and removes the inliers from the set of points until it either contains less than 5 points or 5 lines have been found. The VDM-RT model of the row detection algorithm is shown in Listing 2.

```
1  public getRows: seq of Point`PointM * nat1 ==>
2    seq of (Line`LineM * nat1)
3  getRows(points,n_lines) == (
4    dcl p : seq of Point`PointM := points;
5    dcl lines: seq of (Line`LineM * nat1) := [];
6    for i = 1 to n_lines do (
7      if len p > 5 then
8      let mk_(line,inliers) = hd algo.extractLines(p) ,
9        outliers =  inds p \ elems inliers in (
10          p := [p(out_idx) | out_idx in set outliers ];
11             lines := lines ^ [mk_(line,len inliers)]; );
12   );
13   return lines; );
```

Listing 2. The main operation of the row detection algorithm

An example of the output of the row detection algorithm after being tuned for the specific case can be seen in figure 4.

4.3 Experiment setup

The experiments performed aims to demonstrate the use of timing measurement for obtaining estimates of the row detection algorithm timing. To this end we execute a test consisting of running the row detection algorithm on the 137 LiDAR scans sampled from the farm environment with the tuned parameters, which will allow us to explore the timing performance on two different hardware platforms. The parameters for the row detection algorithm are shown in table 1 and have been selected based on an initial tuning of the parameters for the data set used.

The target hardware platforms are two rugged computers designed for agricultural robotics use. The Robotech 501 (RT501) [11] and the Robotech 101 [10] (RT101). The RT501 is equipped with a 2.8GHz Intel i5-3360M CPU and has 4GB of memory. The

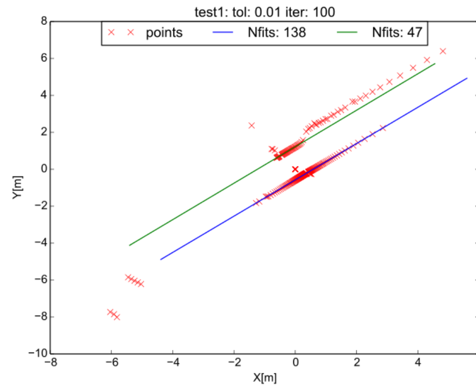

Fig. 4. Plot of the result of running the detect rows algorithm on a scan, where the two plotted lines represent the detected lines which had the most inliers.

Name	Value	Description
n_lines	4	The number of rows to find
tolerance	0.01m	The maximum distance from an inlier point to the line
max_iter	100	The maximum number of RANSAC iterations

Table 1. Parameters used for the experiments

RT101 is equipped with a 1GHz Freescale I.MX6 quad ARM CPU and has 1 GB of memory. Both computers are running Ubuntu 12.04 operating system.

5 Timing measurements

In this section we first obtain the default execution time prediction from the VDM-RT interpreter using the default timing information. Then we execute the code generated model instrumented with timing information gathering on the two hardware platforms, in order to measure the actual execution time of the code generated model. These measurements are then backported into the VDM-RT model using the VDM-RT interpreter extension described in this paper, in order to obtain a new prediction from the VDM-RT interpreter using the hardware measured timings as input. Finally we change the maximum number of iterations the row detection algorithm makes, and compare the prediction of the execution time from the VDM-RT interpreter using the measured timing information, with a measurement of the actual execution time.

5.1 Executing the model using default times

As a first indication of the timing performance of the row detection algorithm, this algorithm is executed on the test data using default time values, with the clock frequency set to 2.8GHz and 1.0GHz to simulate the Robotech 501 and the Robotech 101 platforms,

respectively. The results obtained from these two simulations can be seen in table 2 and table 3.

Operation	Mean	Median	Min	Max	stddev
getRows	$3.6ms$	$3.1ms$	$2.6ms$	$5.6ms$	$866.4\mu s$
extractLines	$892.6\mu s$	$784.3\mu s$	$95.0\mu s$	$1.7ms$	$540.1\mu s$
getInliers	$8.8\mu s$	$7.8\mu s$	$866.2ns$	$16.8\mu s$	$5.4\mu s$
getRandomLine	$70.2ns$	$69.0ns$	$69.0ns$	$267.0ns$	$6.8ns$
addNewBestFit	$11.3ns$	$11.2ns$	$11.2ns$	$11.2ns$	0

Table 2. Default timing obtained from the VDM-RT interpreter running the test with clock frequency of 2.8GHz.

Operation	Mean	Median	Min	Max	stddev
getRows	$8.3ms$	$7.3ms$	$6.1ms$	$13.1ms$	$2.0ms$
extractLines	$2.1ms$	$1.8ms$	$236.4\mu s$	$3.9ms$	$1.3ms$
getInliers	$20.6\mu s$	$18.1\mu s$	$2.2\mu s$	$39.2\mu s$	$12.6\mu s$
getRandomLine	$163.7ns$	$161.0ns$	$161.0ns$	$623.0ns$	$15.8ns$
addNewBestFit	$26.2ns$	$26.2ns$	$26.2ns$	$26.2ns$	0

Table 3. Default timing obtained from the VDM-RT interpreter running the test with clock frequency of 1.0GHz.

The results obtained by executing the VDM-RT model show that the algorithm runs 2.8 times faster on a CPU with 2.8GHz clock compared to a CPU with 1.0GHz clock, which is expected. Furthermore the results indicate that the execution time of the algorithm varies based on the input. The histograms in figure 5 show the timing measurements for each of the operations. In particular, it can be seen that the variation in execution time comes from the `getInliers` operation, which varies between $866.2ns$ and $16.8\mu s$ in execution time. The variation in execution time is important since the approach in this paper only includes the mean execution time, a large variation tells us that this operation may not be accurately modelled using a mean value.

5.2 Measuring time on the target hardware

The execution time of the different operations constituting the row detection algorithm can be seen in table 4 and table 5 for the RT501 and RT101 platform.

When comparing the measured execution times on the hardware platforms with those from the VDM-RT model execution a large difference is observed. For example the mean execution time of the `getRows` operation is $45.9ms$ on the RT501 but the prediction by the VDM-RT interpreter is $3.6ms$ which is more than a factor 10 off. Furthermore the measured execution times show that the difference for the `getRows` between the RT501 and RT101 is also approximately a factor of 10, where the predicted difference was 2.8.

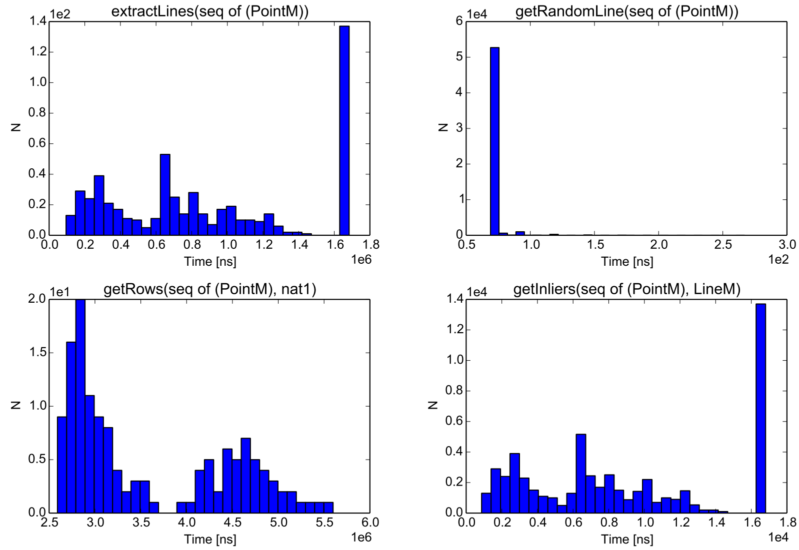

Fig. 5. Histogram of the main operations of the row detection algorithm, obtained from the 2.8GHz CPU default timing.

Operation	Mean	Median	Min	Max	stddev
getRows	45.9ms	45.8ms	37.4ms	58.8ms	3.9ms
extractLines	10.9ms	6.8ms	950.8μs	40.7ms	10.3ms
getInliers	103.2μs	56.9μs	7.2μs	2.3ms	147.0μs
getRandomLine	286.1ns	272.0ns	229.0ns	31.0μs	177.3ns
addNewBestFit	3.2μs	3.2μs	2.5μs	12.4μs	542.6ns

Table 4. Timing results for RT501 hardware platform executing the C++ generated model

Operation	Mean	Median	Min	Max	stddev
getRows	422.2ms	423.4ms	348.3ms	535.6ms	34.2ms
extractLines	100.5ms	64.6ms	9.5ms	360.7ms	91.5ms
getInliers	935.9μs	532.3μs	71.0μs	7.2ms	1.3ms
getRandomLine	3.7μs	3.7μs	3.0μs	82.0μs	692.7ns
addNewBestFit	28.6μs	28.0μs	20.7μs	47.3μs	3.1μs

Table 5. Timing results for RT101 hardware platform executing the C++ generated model

5.3 Timing from VDM-RT interpreter using target measurements

The result of backporting the mean values of the operations (`getRandomLine`, `getInliers` and `addNewBestFit`) to the VDM-RT model and executing the model using the VDM-RT interpreter is shown in table 6 and table 7 for the RT501 and RT101 respectively.

Operation	Mean	Median	Min	Max	stddev
getRows	$41.4ms$	$41.4ms$	$41.3ms$	$41.5ms$	$59.1\mu s$
extractLines	$10.4ms$	$10.4ms$	$10.3ms$	$10.4ms$	$15.5\mu s$
getInliers	$103.2\mu s$	$103.2\mu s$	$103.2\mu s$	$103.2\mu s$	na
getRandomLine	$143.0ns$	$143.0ns$	na	$286.0ns$	$143.0ns$
addNewBestFit	$3.2\mu s$	$3.2\mu s$	$3.2\mu s$	$3.2\mu s$	na

Table 6. VDM-RT interpreter timing results using the RT501 timing backported to the model

Operation	Mean	Median	Min	Max	stddev
getRows	$376.3ms$	$376.3ms$	$376.1ms$	$376.5ms$	$88.1\mu s$
extractLines	$94.1ms$	$94.1ms$	$94.0ms$	$94.2ms$	$46.0\mu s$
getInliers	$935.9\mu s$	$935.9\mu s$	$935.9\mu s$	$935.9\mu s$	na
getRandomLine	$3.7\mu s$	$3.7\mu s$	$3.7\mu s$	$3.7\mu s$	na
addNewBestFit	$28.6\mu s$	$28.6\mu s$	$28.6\mu s$	$28.6\mu s$	na

Table 7. VDM-RT interpreter timing results using the RT101 timing backported to the model

We have chosen to only use the measured mean values for the operations which are called by the `extractLines` operation. This allows us to explore how close the VDM-RT interpreter prediction is to the actual mean execution time of the algorithm, and will furthermore allow us to simulate with different algorithm parameters, such as number of iterations, without having to re-measure the execution time on the real platforms. Table 8 shows the execution time if we execute the algorithm with **max_iter**, from table 1, set to 50 instead of 100 for the RT501 experiments.

Operation	Mean	Median	Min	Max	stddev
getRows	$20.8ms$	$20.8ms$	$20.7ms$	$20.8ms$	$8.5\mu s$
extractLines	$5.2ms$	$5.2ms$	$5.2ms$	$5.2ms$	$4.8\mu s$
getInliers	$103.2\mu s$	$103.2\mu s$	$103.2\mu s$	$103.2\mu s$	na
getRandomLine	$286.0ns$	$286.0ns$	$286.0ns$	$286.0ns$	na
addNewBestFit	$3.2\mu s$	$3.2\mu s$	$3.2\mu s$	$3.2\mu s$	na

Table 8. VDM-RT interpreter timing results using the RT501 timing and **max_iter** set to 50

Operation	Mean	Median	Min	Max	stddev
getRows	$24.2ms$	$24.4ms$	$18.7ms$	$31.3ms$	$2.5ms$
extractLines	$5.5ms$	$3.4ms$	$444.8\mu s$	$22.1ms$	$5.3ms$
getInliers	$104.0\mu s$	$56.8\mu s$	$7.0\mu s$	$2.1ms$	$148.4\mu s$
getRandomLine	$278.2ns$	$263.0ns$	$221.0ns$	$12.0\mu s$	$146.8ns$
addNewBestFit	$3.1\mu s$	$3.1\mu s$	$2.4\mu s$	$12.1\mu s$	$455.9ns$

Table 9. Timing results for RT501 hardware platform executing the C++ generated model with **max_iter** set to 50

A comparison of the times for the `getRows` operation in table 8 and table 9 shows that the prediction made using the VDM-RT interpreter and the timing information in table 4 with **max_iter** set to 100, differs only by 16.5%. This implies that we in our case can use the obtained timing information to determine time related impact of changing some of the row detection parameters.

6 Summary of the case study

The timing information obtained in section 5 shows that the default prediction, using 2 cycles per instruction as timing input, is far from the measured results. One reason for this is the performance of the C++ code generator, which does not output optimised code. A second reason is that the default duration used for each statement by the VDM-RT interpreter does not reflect the actual cost, because the interpreter instructions are not a direct replicate of primitive CPU instructions. However the results obtained by automatically backporting the measured mean execution time, could be used to predict the execution time of the algorithm running with a different set of parameters. This prediction came close (within 16.5%) to the measured execution time. However the parameter changed was directly related to the number of times the measured operation was called. If we had changed a parameter which directly influenced the runtime of the algorithm, e.g the tolerance in table 1 we would not see any difference in the predicted execution time. However we believe that the automated approach makes it easy to make new tests which can provide a better set of execution times for various parameter configurations.

Even though the predicted execution time (based on time measurements) is not accurate in all cases, the difference between two platforms could still be predicted with 16.5% difference between the prediction and measurement. The prediction from the VDM-RT interpreter using the default timing, showed that the difference in runtime should be 2.8 but the measured execution times showed that there was a difference of approximately 10 between the RT501 and RT101 platforms.

The results obtained from executing the VDM-RT interpreter using timing information from the real platform allows us make more precise predictions on the required performance of the selected hardware platform, while allowing us to explore the consequence of changing the parameters of the algorithm and see both the effect on the performance, e.g does it find the rows, and the effect on the required execution time.

7 Discussion

We now discuss the two extensions developed for the Overture tool. The extension for the C++ generator, instrumenting the generated code with timing information measurement calls, automates the process of measuring the execution time of operations and functions. However care still has to be taken when defining the model to be executed, for example, the methods measured must be called a number of times such that a proper mean value can be calculated.

The extension made to the VDM-RT interpreter, where the timing measurements are injected into the AST before execution, automates the process of annotating a

model with execution time information. Operations or functions which varies in execution time, due to e.g for-loops are difficult to back port to the VDM-RT model. The backported timing of a operation is a single mean value of all the measurements, and therefore not useful for describing the execution time of an operation of function which varies based on state or input. Here a solution could be to manually specify a duration for the body of the for-loop.

We note that the two extensions are not coupled to each other, but instead solve two different problems. The extension to the code generator makes it easy to generate source code containing the timing measurement logic and the trace back to the model. The VDM-RT interpreter extension provides the means to inject duration statements into a given model, using a file containing the mappings of operations and functions to the mean execution time. This means that any tool can provide the execution time information, as long as the format is correct. This makes it possible to include models which, for example, are generated to Java or manually translated into source code.

8 Concluding remarks

The two prototype extensions to the Overture tool presented in this paper, provide a method for automatically obtaining timing estimates on real hardware platforms, and automatically include the measured execution times back into the VDM-RT model. We demonstrated the extensions on a row detection algorithm, where we obtained the mean execution time of each operation and automatically backported the measured execution times into the VDM-RT model, and compared the measured execution time with the VDM-RT interpreter prediction. The experiments conducted showed that we could predict the execution time of the algorithm with a 16.5% percent accuracy using the measured mean execution time.

The experiments performed are far from being conclusive and the extensions in their current form have a limited use case, since they only provide operation/function level inclusion of measured timing information. One improvement we want to investigate is to create a platform execution time benchmark, which would measure the execution time of each of the interpreter constructs such as statements and expressions on the hardware platform using the Overture code generator. The measured time of each code generated construct can then be used as the default value when executing a model with the VDM-RT interpreter.

Another important aspect of the proposed extensions is how they are best used in the design of Cyber-Physical Systems which typically include multiple hardware platforms. Furthermore Cyber-Physical Systems include multiple controllers which have to meet certain deadlines, here we believe that the proposed extensions can, together with the Overture platform, provide a powerful basis for exploring design alternatives including hardware platform selection.

References

1. Da Penha, D.O., Weiss, G.: Integrated timing analysis in the model-driven design of automotive systems. Proceedings of NiM-ALP p. 40 (2013)

2. Fischler, M.A., Bolles, R.C.: Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. Communications of the ACM 24(6), 381–395 (1981)
3. 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
4. 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
5. Gamma, E., Helm, R., Johnson, R., Vlissides, R.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
6. Jørgensen, P., Larsen, M., Couto, L.: A code generation platform for vdm. University of Newcastle-upon-Tyne. Computing Science. Technical Report Series CS-TR-1446 (2015)
7. 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
8. Rapitime (April 2015), http://www.rapitasystems.com/products/RapiTime
9. Ribeiro, A., Lausdahl, K., Larsen, P.G.: Run-Time Validation of Timing Constraints for VDM-RT Models. In: Wolff, S., Fitzgerald, J. (eds.) Proceedings of the 9th Overture Workshop. pp. 4–16. No. ECE-TT-2 in Technical Report Series (June 2011), http://eng.au.dk/fileadmin/DJF/ENG/PDF-filer/Tekniske_rapporter/Technical_Report_ECE-TT-2-SAMLET.pdf
10. Conpleks robotech 101 computer datasheet (April 2015), http://conpleks.com/wp-content/uploads/2015/04/DE-00-150-026-en-A1.pdf
11. Conpleks robotech 501 computer datasheet (April 2015), http://conpleks.com/wp-content/uploads/2014/04/DE-00-150-015-en-C1.pdf
12. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) 7(3), 36 (2008)

Case Studies on Combination of VDM and Test-Driven Approaches: Application, Model Finding and Refinement

Fuyuki Ishikawa

National Institute of Informatics, Japan
f-ishikawa@nii.ac.jp

Abstract. Testing has been considered as the key approach to validation in VDM, as its languages include executable syntax. One of the benefits of testing is that it allows engineers to obtain confidence by examining concrete scenarios. Meanwhile, many practitioners are nowadays attracted by test-driven development (TDD) and its extensions for constructing program code. These approaches also focus on testing, or concrete scenarios. However, the difference is that test cases are also considered as the key information that guides development (explained as "Specification by Example"). In this paper, matching between VDM and test-driven approaches is discussed through case studies with industry engineers. Three topics are discussed: application of TDD to VDM, model finding that supports TDD and VDM, and challenges for test-based refinement in VDM.

1 Introduction

VDM has attracted a wide range of researchers and practitioners as a representative of formal specification methods. One of the key characteristics of VDM is its executable syntax. It is supported well by tool functions, such as interpreters and coverage measurement, in VDMTools [6] and Overture [5]. Thus testing is the key approach for validation in VDM.

There are many advantages of testing as a means of validating formal specifications. For example, engineers can apply practices of testing that they are familiar with in their activities for testing program code (while they are often not so familiar with practices of theorem proving or model checking). Another notable point is that testing allows for looking at concrete scenarios that give much more confidence. In contrast, it is almost impossible to have real feeling of confidence by only writing declarative predicates or by looking at "OK" messages from theorem provers and model checkers. This power of concrete examples has been discussed in many different contexts, e.g., [16].

In communities for agile software development, practitioners have discussed principles and methods for test-driven development (TDD) [8]. Here, test cases (or examples) are considered as primary elements that guide the development activities. They play similar roles to those of specifications and are not only

for checking the constructed code. Recent extensions, such as behavior-driven development (BDD), have honed this direction with principles such as "tests as documents" and "specification by example" [3, 7]. Test cases can give more confidence and have less dependency on experience in programming or declarative descriptions. Thus they can be widely used for discussions, e.g., with customers.

It could be interesting to focus on possible combinations of VDM and test-driven approaches, both emerging from apparently different communities, since the key factors of test cases or examples are in common. In this paper, this matching is discussed through case studies with industry engineers, through the author's education and application experience with the industry e.g., in the Top SE program [11, 12]. Three topics are discussed: application of TDD to VDM, model finding that supports TDD and VDM, and ongoing challenges for test-based refinement in VDM.

The remainder of this paper is organized as follows. The principles of TDD and its extensions are introduced in Section 2. Then application of TDD to VDM is discussed in Section 3. A trial is discussed about incorporation of a model finder in the combination of TDD and VDM in Section 4. An ongoing challenge is described with defining test-based refinement in VDM in Section 5. Section 6 contains discussions and concluding remarks.

2 Test-driven Development and its Extensions

Assuming the readers of this paper are familiar with VDM and formal methods, this section briefly introduces test-driven development (TDD) and its extensions.

We use a popular sample for test design, that is, a function that judges the type of a triangle given three integer values for the lengths of the edges [14]. Below is the interface definition in the VDM language (VDM-SL or VDM++). The result type is defined by an enum type for equilateral, isosceles, scalene, and non-triangles.

In this section, the standard TDD for program code is introduced, however, using the VDM syntax. The interface of the target function is defined as follows.

```
types
  TType = <EQUI> | <ISO> | <SCA> | <NON>;

functions
  judgeTriangle : int * int * int -> TType
  judgeTriangle(a, b, c) == is not yet specified;
```

When an engineer uses test-driven development for this function, he/she will start with a check list (TODOs) and test cases. The (initial) check list probably includes items like "Can Judge Equilateral," "Can Judge Scalene," and so on. Suppose the engineer thinks the equilateral item is the easiest. Then he/she designs and writes a test case for it first, e.g., using VDMUnit:

```
TestEquilateral : () ==> ()
```

```
TestEquilateral() ==
  assertTrue( judgeTriangle(5,5,5) = <EQUI> );
```

The engineer sees that this test case fails, obviously as the executable part of the function is not given. He/she then concentrates on writing code to pass it. The code can even be fake, e.g., just write `return <EQUI>`, if he/she thinks the problem is too difficult to write clean code that works (though this is not the case in this simple problem for explanation).

After passing the first test case, the engineer chooses another item in the check list and writes a test case, e.g., for a scalene triangle, which fails. The executable part is then generalized a little more by this "triangulation."

These small cycles are repeated until the people involved have confidence about the realization of the function. Each cycle may include extraction of functions or operations found necessary as well as refactoring. Thus, the design evolves through the cycles rather than being fully predicted before coding.

The first point of this process is to have quick feedback with small cycles. In other words, one should avoid writing long code without any checks, as it can result in code that mysteriously works for some test cases, but not for others. Such code entails a lot of effort to debug and rollback.

The second point is to use and discuss concrete test cases or examples. Customers (non-programmers) can understand and discuss the examples with confidence if they are about UI-level functions. On the other hand, even experts may assign different meanings to general expressions or easily overlook deficiencies due to careless mistakes or the complexity of the problem. Human-readable representation of test cases is sometimes used to realize principles called "Tests as Documents" and "Specification by Example" in behavior-driven development (BDD) or acceptance test-driven development (ATDD) [3, 7].

This paper omits discussing the details of other essential aspects of TDD and its extensions, such as focus on value with high priority, avoidance of unnecessary generalization, as well as exploratory design through refactoring.

3 Application of TDD to VDM

There seem to be no reason that invalidates the rationales for TDD when they are applied to VDM specification, not program code. The author had a case study with an engineer who wanted to investigate this point. He originally had a feeling that TDD can be a good way to learn VDM, which enables to start with small steps while getting quick feedback through execution.

This paper does not give theoretical or empirical support for such a claim about the effectiveness of TDD, as in studies such as [9]. Below, the differences between program code and VDM specification are discussed.

3.1 TDD Process for VDM

TDD can be applied as it is for the executable part, i.e., the explicit description, in functions and operations in VDM. For the other part, i.e., the implicit

description, it is also possible to use TDD by focusing on the functions that represent pre- and post-conditions.

Below is an example of test cases for the post-condition function of the `judgeTriangle` function defined in Section 2.

```
TestJudgeCorrectIsoscelesResult : () ==> ()
TestJudgeCorrectIsoscelesResult() ==
  assertTrue( post_judgeTriangle(5,3,5,<ISO>) );
```

It is originally up to the modellers whether to write the explicit part first or the implicit part first. One can gain more confidence by starting with the more concrete explicit part. Or, one can start with writing what is required without worrying about “how”. In any case, one can follow the TDD principles, clarifying the next TODO, defining a test case for that, and write the corresponding part of VDM.

3.2 Notes on Testing Conditions

Even though TDD can be applied in the same way, it can be more effective on the declarative specification of constraints, i.e., pre- and post-conditions. It is possible to keep the same discussion for invariants, though concrete examples are omitted in this paper. Below, key points discussed in the case study are described, which do not essentially depend on TDD or test-first but are significant as principles for testing of conditions.

Value of Small Steps It is somewhat obvious that any part of description should receive feedback as soon as possible before being extended into a larger one. Below is an example of post-condition with a fault for the triangle sample.

```
post
  a <> b and b <> c <=> \result = <SCA>
```

This condition lacks `c <> a`, e.g., accepts the result `<SCA>` for `(a,b,c)=(5,3,5)`. This can be detected by testing the post-condition function, if test cases such as the following is used:

```
TestJudgeIncorrectScaleneResult : () ==> ()
TestJudgeIncorrectScaleneResult() ==
  assertFalse( post_judgeTriangle(5,3,5,<SCA>) );
```

There is a higher possibility of finding the fault if engineers try to define test cases for situations in which the post-condition denies wrong results.

When an error is detected and suggests something is wrong, debugging is much easier in smaller steps, compared with the full specification constructed without any validation, such as:

```
post
  a <> b and b <> c <=> \result = <SCA>
    and
  (a = b and b <> c) or (b = c and c <>)
      or ... <=> \result = <ISO>
```

This post-condition becomes false in the case of c=a. For example, it even rejects the correct result <ISO> for (a,b,c)=(5,3,5), which can be a kind of mystery especially for those who are not so familiar with logics.

The principle of small steps, in other words, unit testing on each component formula, can add more confidence and make the debugging easier.

Weak Post-conditions It is more likely to have weak post-conditions that accept wrong results, for example:

```
post
  \result = <SCA> => a <> b and b <> c
```

This condition does not reject, for example, the result <SCA> for (a,b,c)=(5,3,5).

If such a weak post-condition is used to check results from the explicit part, it does not cause an error or help detect a fault. This occurs silently, and it is likely no one would investigate the details of this "test pass." Thus, testing the pre- and post-conditions, not only testing the explicit part using the conditions, is significant for reliability.

This silent false-negative pass of test occurs only with weak conditions. For example, too strong conditions cause a false-positive test failure and are thus investigated. As weak conditions mean less fault-finding capability [15], it is worth recommending making test cases that check whether post-conditions can reject wrong results expectedly.

4 Model Finding for VDM and TDD

4.1 Motivation and Approach

TDD completely relies on the design of test cases. Although a check list can start with easy examples, it should eventually become complete enough for the people involved to have confidence in achieving the expected value. The fundamental methods for testing are helpful, e.g., equivalence partitioning and boundary analysis, but require context-specific applications and discussions, e.g., what are the partitions in our case? Test cases such as (a,b,c)=(5,3,5) are actually derived results. The test designs, or the intentions behind the cases, are a target of discussion and validation, e.g., try at least one case of c = a && a <> b.

The "test design " discussed above is a specification of the test suite, e.g., "include one test case for isosceles triangles with a = c." The specification of the function also matters as a test oracle that determines whether the expected output values are correctly defined. Thus it is valuable to derive or assert test cases with formal properties as needed.

Given the above motivation, the author created a prototype of the following.

- A language that add test cases and test designs in addition to the specification of functions and operations in VDM. The language design is independent of the target specification or programming language, and currently instantiated as extensions for VDM and Java.
- A tool based on the language that generates or validates test cases (examples). The current prototype was built quickly by using a model finder, or a constraint solver, Alloy Analyzer [1], with simple mechanisms of symbolic representation to mitigate the problem of state explosion.

4.2 Sample Scenario

Here, the language and tool are illustrated with the triangle sample. As the first step, suppose that non-positive values for the input `(a,b,c)` are considered invalid, rather than valid input that makes a non-triangle. In this case, the tool generates test cases that include only positive values for the three arguments:

```
pre
  a > 0 and b > 0 and c > 0
```

a	b	c
2	3	10
19	20	0 [LowerB]
...	...	...

The choices of the values are arbitrary. In the second line, the value for `c` accompanies the tag “LowerB” (for lower boundary). In this simple case, the tool can understand the constraints bound by constant values and can attach the tag. The tool may generate the output value (`TType`) as well, but does not do so by default as arbitrarily wrong output values are not so meaningful.

The tool also accepts a command to generate test cases with invalid input, attaching tags for invalid values:

a	b	c
-2 [Under]	10	3
-1 [UnderB]	5	-8 [Under]
...	...	...

Suppose we add part of the postcondition for equilateral triangles. The tool then starts to include the result in the output (for the valid input):

```
post
  a = b && b = c => \result = <EQUI>
```

a	b	c	\result
3	3	3	`<EQUI>`
1 [LowerB]	3	5	`<NON>`
10	9	3	`<EQUI>`
...	...	...	...

As expected, result values are arbitrary for all cases except for the first row that matches the left side of the implication (`a = b && b = c`). The tool applies a heuristic that automatically adds a test design that includes at least one case that matches the left side of the top-level implication formula.

Suppose an engineer had a test case already defined and agreed to by people involved, e.g., consumers of this function. He/she can add the test case with a tag and the case is then always included in the result of the tool with the tag:

```
example \"ex-equi" a=5 && b=5 && c=5 && \result=<EQUI>
```

a	b	c	\result	PROP
5	5	5	<EQUI>	[ex-equi]
3	3	3	<EQUI>	
1 [LowerB]	3	5	<NON>	
10	9	3	<EQUI>	
...	...	...	...	...

This is an extended syntax from the current one of VDM.

Confidence in the meaning of the postcondition can be increased by using a command to show counterexamples that show wrong execution results from valid input, i.e., cases that satisfy the pre-condition but not the post-condition:

a	b	c	\result
3	3	3	<ISO>
5	5	5	<NON>
2	2	2	<SCA>
...	...	...	...

From a different aspect, the language allows test design descriptions to be given, primarily in the form of partitions. The following is a sample of a test design description, that is not only partial but also naive.

```
partition
  \"p-equi" a = b && b = c, \"p-sca" a <> b && b <> c,
  \"p-iso1" a = b && b <> c, \"p-iso2" b = c && c <> a,
  \"p-iso3" c = a && a <> b
}
```

The scalene partition lacks `c <> a` and is thus not disjoint with the third isosceles partition. The following is a possible result table.

a	b	c	\result	PROP
3	4	5	<EQUI>	[p-sca]
5	5	4	<ISO>	[p-iso1]
9	3	3	<NON>	[p-iso2]
4	7	4	<NON>	[p-sca,p-iso3]
5	5	5	<EQUI>	[ex-equi,p-equi]
...	...	...	...	...

The tags help a lot to see which case is in which partition(s), and in this case there are unexpected multiple partition tags in the fourth row. In fact, the above partition also lacks the condition to compose a triangle, which may be found similarly when the partition for non-triangle is added.

It is notable that this language and tool have a different, complementing objective from the trace specification in VDM and its tool support. Rather than generating numerous test cases exhaustively, the tool supports understanding and human validation of the test design.

4.3 Current Status

The presented language and tool mix specification, test design, and test cases together with a model finder. This approach enables quick feedback in an iterative process (called Spec-Test-Go-Round).

The prototype implementation was done for the Java version, and tested with over 60 industry engineers in the form of a one-day seminar. The approach received positive feedback in terms of the direction it provides as well as the applicability for different tasks in TDD or formal specification. It is also notable that the seminar received very positive feedback as it mixed knowledge and principles from apparently different communities. The most critical issue is performance or scalability as the current implementation employs some simple heuristics with an existing model finder. More detail of the prototype and the seminar can be found in [10].

5 Test-based Refinement

In another case study with an engineer, a lightweight and practical way for refinement was discussed. As software development involves models of different abstraction levels, it is essential to consider refinement in some form, i.e., connecting models of different levels in a systematic, verifiable way.

5.1 Motivation

B-Method [2] and Event-B [4] are good targets of discussions on refinement with modelling paradigms similar to that of VDM. In these methods, a part of invariants declares relationships between variables in the abstract model and those in the refinement model (called "link invariants" or "gluing invariants"). Thus, correspondence between states in the two models is clarified and verified to be kept consistently through state changes. Correspondence between operations or methods in the two models is also clarified, too, and validated with specific constraints, e.g., guard conditions do not become weak in the refinement model and thus break invariants proved in the abstract model.

Below, three issues are discussed that are related to this kind of formal refinement rules.

First, it is difficult for beginners to plan or understand refinement steps that are designed to align with this kind of rules. For example, refinement basically eliminates possibilities of transitions, in other words, decreases non-deterministic behaviors. Engineers may want to just start with an intuitively abstract model that defines the system behavior only in the case of success, without talking about anything related to failures. However, in Event-B for example, the first model should declare whether the possible result is either a success or failure, defining transitions that include those equivalent to or have more possibilities than the actual transitions. Therefore easy explanations like "you can go from abstract to concrete" do not hold.

The above discussion does not mean to go against the classical view on refinement, which has been proved very useful with a return on investment in correctness by construction. Though, there is a gap from the current practices in which UML models with multiple abstraction levels are constructed. Even if the models follow some systematic rules, they may be different from the rules defined in formal specification methods. Therefore, in this study, enabling flexible, user-defined refinement rules is considered.

Second, the refinement rules sometimes define constraints over internal information (data structure, events, etc.). Because such internal information is often a "mock-up" in abstract models [13], investigation of constraints over internal information seems not essential in terms of insights into and impacts on the development process. Therefore, in this study, refinement that only define constraints over the observable part of the model is considered (how the disclosed functions and operations behave in response to requests or events).

Finally, another essential point is that refinement shares the same issue discussed in this paper: obtaining confidence with declarative link invariants is very difficult. Therefore, in this study, defining refinement constraints in terms of test cases is considered.

5.2 Approach

A case study using VDM is conducted to try a refinement method that deals with the three issues. The basic idea is to have explicit refinement constraints, which connect two models, only in terms of test cases. Internal data structures and behaviors are free from specific rules forced by a method and thus transformed in various ways, as long as the observable part is refined adequately.

Starting Example As a very simple example, consider the following login operation.

```
types
  Authenticator = token;

operations
  public login : token ==> ()
  login(authenticator) == ...;
```

Suppose this is the first model as the result of system analysis, in which operations provided by the system are first defined with conceptual vocabularies. In this case, design of the authenticator is abstracted away and the token type is used.

Some test cases contain a call of this disclosed operation.

```
operations
  public test1A : ...
  test1A() ==
     ( ...
     login(mk_token("tom"));
     ... )
```

In the next step of early design, a decision is made to use a user name and a password as the authenticator.

```
types
  Authenticator ::
    username : seq of char
    password : seq of char;

operations
  public login : Authenticator ==> ()
  login(authenticator) == ...;
```

Then, the corresponding test cases can be defined that call the refined form of the operation.

```
operations
  public test1A : ...
  test1A() ==
     ( ...
     login(mk_Authenticator("tom", "tompwd"));
     ... )
```

The observable operation is refined from a conceptual systematization level to an early design level. The refinement method only requires defining a rigorous mapping between the test cases. In the above example, the presented parts (the lines calling `login`) of the two test cases are intuitively the same. Specifically, the arguments of the calls are conceptually the same, referring to the same user "tom," though the refinement one has more detail (password) in a different type. We can formalize this rule to define correspondence between the arguments to the `login` operation in the two different models, as follows.

```
match : MODEL1`Authenticator * MODEL2`Authenticator -> bool
match(a1, a2) ==
  exists pwd : seq of char &
    a2 = MODEL2`mk_Authentication(a1, pwd);
```

If mappings for the omitted parts in the test cases are also defined, then there is a rigorous mapping between the two test cases, in a user-defined way. Thus, test-based refinement has the potential to allow for flexible refinement that requires the minimum level of consistency between models with different abstraction levels in a user-defined way.

Current Status We are still accumulating case studies that connect two models with test cases as in the example. For instance, sometimes we want to connect a precondition in a model and an input check behavior that throws exception in the refinement. This approach requires clarification into a generic, repeatable method and evaluation through experiments. We are now extracting rules, including the following.

- Each model has a list of functions or operations disclosed to and accessible from the outside.
- Each model has a list of test cases and they manipulate only the observable functions and operations.
- Each refinement model defines links between observable functions and operations in the refinement and those in the abstract model.
- Each refinement model defines links between test cases in the refinement and those in the abstract model.

6 Concluding Remarks

In this paper, further roles of test cases were discussed, not only verifying the explicit part of VDM specification after its construction. The author believes this direction will help with the use of VDM as well as with connecting with practitioners because of the essential role of testing in VDM as well as increasing interest in test-driven approaches by practitioners.

7 Acknowledgements

The author would like to thank all the engineers and researchers for their discussions on the topics presented in this paper, especially Jun Itoh and Yuma Mizutani.

References

1. Alloy. `http://alloy.mit.edu/alloy/`
2. B Method - Presentation of B Method, B Language, and formal methods. `http://www.bmethod.com/`
3. Cucumber - making bdd fun. `http://cukes.info/`
4. Event-b.org. `http://www.event-b.org/`
5. Overture - Open-source Tools for Formal Modelling. `http://www.overturetool.org/`

6. VDM information web site. `http://vdmtools.jp/en/`
7. Adzic, G.: Specification by Example: How Successful Teams Deliver the Right Software. Manning Pubns Co (2011)
8. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional (2002)
9. Erdogmus, H.: On the effectiveness of test-first approach to programming. IEEE Transactions on Software Engineering 31(3), 226–237 (January 2005)
10. Ishikawa, F., Doi, T., Sakamoto, K., Yoshioka, N., Tanabe, Y.: Enlightening test-driven with formal, formal with test-driven through spec-test-go-round. Tech. Rep. GRACE-TR 2015-05, GRACE Center, National Institute of Informatics (June 2015)
11. Ishikawa, F., Taguchi, K., Yoshioka, N., Honiden, S.: What Top-Level Software Engineers Tackles after Learning Formal Methods - Experiences from the Top SE Project. In: The 2nd International FME Conference on Teaching Formal Methods (TFM 2009). pp. 57–71 (November 2009)
12. Ishikawa, F., Yoshioka, N., Tanabe, Y.: Keys and roles of formal methods education for industry: 10 year experience with Top SE program. In: 1st Formal Methods in SW Engineering Education and Training Workshop (FMSEET 2015) (June 2015)
13. Kurita, T., Ishikawa, F., Araki, K.: Practices for formal models as documents: Evolution of VDM application to "Mobile FeliCa" IC chip firmware. In: 20th International Symposium on Formal Methods (FM 2015) (June 2015)
14. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, 3rd edn. (2011)
15. Polikarpova, N., Furia, C.A., Pei, Y., Wei, Y., Meyer, B.: What good are strong specifications? In: The 2013 International Conference on Software Engineering (ICSE 2013). pp. 262–271 (2013)
16. Zayan, D., Antkiewicz, M., Czarnecki, K.: Effects of using examples on structural model comprehension - a controlled experiment. In: The 36th International Conference on Software Engineering (ICSE 2014). pp. 955–966 (May 2014)

Pacemaker Parameter Tuning using Crescendo

Carl Gamble[1], Martin Mansfield[1], John Fitzgerald[1], and Peter Gorm Larsen[2]

[1] School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
[2] Department of Engineering - Software Engineering, Aarhus University, Aarhus, Denmark

Abstract. We apply collaborative model-based techniques to an aspect of the design of an artificial cardiac pacemaker. Our co-model consists of a pacemaker controller represented in the VDM-RT notation, linked with a model of the heart electrical activity represented using signal models in the 20-sim tool. We extend and improve the architecture of a previous co-model to allow tuning of design parameters for a noise detection function that controls a mode switch. Our experience leads to recommendations for design space exploration support in tool chains for cyber-physical systems, including enhanced post-processing of co-simulation outcomes, and the ability to state and evaluate cross-simulation predicates.

Keywords: pacemaker, co-modelling, co-simuation, design space exploration, VDM, 20-sim, overture

1 Introduction

In order to gain confidence in the trustworthiness of Cyber-Physical Systems (CPSs), it is necessary to combine heterogeneous models in a disciplined way. For example, ICT and software engineers who naturally use rich discrete-event (DE) models must collaborate with engineers who use continuous-time (CT) techniques suited to the description of the controlled plant and environment. A goal of the Cyber-Physical Lab at Newcastle University[3] is to facilitate collaborative modelling and simulation, in which engineers to work in the most expressive notations for the cyber and physical elements of the particular problem, and allowing trade-offs across the DE/CT divide.

The DESTECS project[4] developed a framework for collaborative designs (called *co-models*) composed of models of DE (typically loop and supervisory controller) and CT (typically plant/environment) elements expressed in different formalisms. Reconciled operational semantics for the two formalisms allows the constituent models to run in their own simulators, while a *co-simulation* engine manages the synchronisation of time and data between them under the control of an external script that can represent environment or user choices. Co-models are created and simulated using the Crescendo tool[5]. Methods have been developed for the construction of co-models [1], patterns for fault tolerance co-modelling have been created [2], and there is support for Design Space Exploration (DSE) by multiple co-simulation and ranking of co-models [3]. The

[3] http://research.ncl.ac.uk/cplab/
[4] http://www.destecs.org
[5] http://crescendotool.org

approach and tools are being validated in industry on applications including transportation, heavy machinery and high-speed paper processing [4].

Medical applications represent an important and challenging domain for CPS design technology [5, 6]. In order to provide a common basis for evaluating emerging methods and tools, McMaster University and Boston Scientific released a natural language specification for a previous generation artificial cardiac pacemaker [7], inspiring several studies on the use of formal DE-only modelling [8–11]. However, it is appropriate to consider co-modelling in this context, since confidence in such a medical CPS requires models of diverse aspects such as software, electrophysiology and even fluid flow.

An artificial pacemaker is intended to replace or compensate for incorrect functioning of the natural heart. Our concern is therefore to ensure that “faulty heart plus pacemaker” approximates the behaviour of a “healthy” heart. The system of interest is therefore *not* the pacemaker alone, but encompasses the pacemaker and the heart. This combined system therefore has cyber and physical elements. The purpose of the co-model presented here is to enable exploration of the effects of pacemaker designs on the performance of this system of interest. The co-model therefore consists of a VDM model of the pacemaker controller and a 20-sim model of heart electrophysiology as this structure reflects the cyber and physical characters of these two elements. The work presented here builds on a previous co-model [12] which served as a proof-of-concept for linking an abstract CT model of heart electrophysiology in 20-sim with a VDM model of a pacemaker controller. This model has been revised in order to exercise the application of modelling patterns developed in DESTECS [4], and to explore the capabilities of the Crescendo tool for DSE across the DE/CT boundary.

Section 2 presents background in the pacemaker challenge problem and co-modelling. Section 3 presents the co-model in terms of its constituents; the heart (Section 3.1), the pacemaker (Section 3.2), and the co-model interface between them (Section 3.3). The design space exploration and its results are described in Section 4. Finally, Section 5 describes the limitations and of the modelling performed and identify future research directions.

2 Background

In this section we present a simplified view of heart functioning (Section 2.1), and briefly recall the cardiac pacemaker challenge (Section 2.2). We then describe the solution technology that we explore in our work, specifically co-modelling and design space exploration (Section 2.3).

2.1 The Heart

To pump blood around a body the heart must rhythmically contract and relax its upper and lower chambers (atria and ventricles respectively). The atria contract first, pumping blood into the ventricles which contract shortly afterwards, pumping blood around the lungs and the rest of the body. The contraction timing is controlled by electro-chemical nodes, the Sinoatrial Node (*SA node*) and the Atrioventricular Node (*AV node*). An electrical discharge from the SA node causes a contraction of the atria and, after a short

delay while traversing an internodal pathway, leads to a discharge of the AV which causes contraction of the ventricles. When the system of natural regulation of heartbeat is impaired, an artificial pacemaker may be used (we will use the term "pacemaker" to refer to the artificial device unless stated otherwise).

2.2 Cardiac Pacemakers and the Pacemaker Challenge

An artificial cardiac pacemaker is an implantable medical device that can deliver electrical impulses to a heart via electrodes, stimulating heart muscles in order to regulate the rhythm with which the heart beats. A pacemaker comprises a pulse generator and a number of electrodes; electrodes monitor intrinsic electromagnetic activity across the heart, and based on some computation, deliver electrical impulses when necessary. A pacemaker is prescribed to a patient to maintain a healthy heart rate when the natural rhythm of the heart is inadequate, which can be caused by a number of medical conditions.

Modern pacemakers are highly programmable, and can be tuned by a cardiologist to ensure the optimum parameters for a given patient. A pacemaker can operate according to a number of pacing modes, and can switch between modes based on the circumstances of the patient at any given time. Each mode can restrict which parts of the heart are monitored for intrinsic activity, how the pacemaker should respond to intrinsic activity, and which parts of the heart can be artificially stimulated. Each mode can be categorized according to its response to intrinsic activity: synchronous pacing modes detect when the heart beats sufficiently without assistance and do not initiate stimulation unless the heart fails to do so, whilst asynchronous pacing modes deliver stimulation at a predetermined interval, regardless of any intrinsic activity. For example the simulation output in Fig. 3 contains an over-pacing event witnessed by two close pulses on the first and third traces after the pacemaker had been in an asynchronous mode for a short period. Since synchronous pacing modes only stimulate when strictly necessary, it is preferable to operate in this way whenever possible.

A natural language specification for a previous generation artificial cardiac pacemaker has been made available from the Software Quality Research Lab at McMaster University and Boston Scientific, in an effort to provide a common basis for evaluating emerging methods and tools. The specification is released as part of The Pacemaker Formal Methods Challenge, the first challenge issued by the North American Software Certification Consortium. We use the specification throughout our work as a guide to design decisions, and in particular we focus on the description given to how a pacemaker should respond when the signals it monitors are subject to noise.

A co-model provides a highly detailed description of the CT environment with which the pacemaker must interact. The fidelity of the CT model makes co-simulation suitable for observing the behaviour of the pacemaker over short periods of time (the order of tens of beats) in close detail, rather than for observing the long-term behaviour of the device. For this reason, a co-modelling environment is well suited to demonstrating the reaction of a pacemaker through the onset of noise. In addition, noise detection algorithms can be tuned for optimization, and exploration of parameter values requires exercising the DSE capabilities within the Crescendo tool.

2.3 Co-modelling and Design Space Exploration

A co-model consists of a DE model of a controller and a CT model of the plant, with a *contract* describing controlled and monitored variables, named events (raised in the CT model and handled by the DE model), and shared design parameters. During co-simulation, user and environmental interactions are governed by a script. We use the Vienna Development Method (VDM) formalism for DE-side, and bond graphs for CT-side modelling. VDM is a a rich language with features for modelling object-orientation and concurrency, and real-time distributed embedded systems [13]. VDM models define state variables over abstract data types, and functionality via state-changing operations. VDM simulation is supported by the Overture tool[6]. CT models are built, simulated and visualised using the 20-sim[7] tool [14]. It allows the dynamics of the plant to be modelled in several ways, including signal type connections between equation blocks, and the powerful domain-independent bond graph [15] notation.

Previous work in embedded systems design, such as BODERC [16] and Modelica [17] provides modelling environments and libraries for simulating physical and computing components. Approaches to DE/CT co-simulation are defined by Nicolescu et al. [18], and there are several co-simulation architectures including Cosimate[8] and HLA [19]. Ptolemy II [20] offers DE and CT simulation within a single tool, though lacking the object-orientation offered by VDM and the component libraries offered by 20-sim. Work on time synchronisation between DE and CT models is described in hybrid systems literature, e.g. [21]. The DESTECS approach using the Crescendo tool is distinctive in including a rich but abstract DE-side modelling language, and in managing co-simulation of heterogeneous models in their "native" tools. Simulation is a vital tool in exploring design spaces and discovering key properties; there is also a significant and growing body of work on hybrid systems verification using approaches such as that of KeYmaera [22], which holds out the promise of symbolic verification of known correctness properties (for example, related to safety).

A key feature of the Crescendo tool is its support for Design Space Exploration (DSE) through its Automated Co-model Analysis (ACA) functionality [23]. When developing a model in Crescendo it is possible to define a set of shared design parameters (SDPs), each SDP has its value defined in the simulation launch configuration and can be used in both the DE and CT models. This makes it simple for the modeller to experiment with different parameter values without having to edit the models themselves. The ACA feature of the crescendo tool allows the modeller to define a range of values for each SDP, the tool will then proceed to sweep over all combinations of parameters performing a simulation for each. The details of each simulation including its launch configuration and any results in the form of log files are saved in separate subdirectories for later analysis.

[6] http://www.overturetool.org
[7] http://www.20sim.com
[8] http://www.chiastek.com

3 Co-model

As indicated in Section 1, the purpose of the co-model is to allow the exploration of alternative pacemaker designs on the overall performance of a cyber-physical system of interest composed of the heart and the pacemaker. Our co-model therefore has two main constituents - a heart model and a pacemaker controller model – linked by an interface contract. These elements are described below.

3.1 Heart

In our previous work we modelled the upper and lower chambers of the heart as if they were a pair of charging and discharging capacitors [12]. The capacity of the capacitors and charging rates effectively determined the rhythm of the chambers and the current flowing in and out of each capacitor provided the signals for the ECG output. The resulting model was both complex, containing feedback signals from the lower to the upper chamber that had no analogue in a real heart, and was also hard to tune, specifically determining the correct pacing current and period to trigger the expected response in the chamber.

This new model adopted a different approach, attempting to produce an ECG like signal by mimicking more closely the cells (myocardia) and fibres of the heart muscles. The model is predominantly composed of replicated simple models of myocardia. In reality, the surface voltage of each myocardia derives from the relative concentrations of a number of ions that flow into and out of the cell, controlled by gates. High fidelity models of these ionic flows have been produced [24], but this level of detail and computational cost is not necessary for our purpose as the profile of the surface voltage (action potential) of a myocardium over time and in response of external stimulation is well known. Thus a myocardium may be modelled by replaying the voltage profile from a lookup table or, in the case of this model, a simplified equation.

For this work the myocardia model consists of two submodels, an ‘actionPotential’ block and a trigger block, Figure 1. The purpose of a cell’s trigger block is to observe the action potential of the myocardia immediately adjacent to it and to determine if two conditions are true. The first condition is that this cell is out of its refractory period, this being a time during which stimulation from the outside will not cause it to depolarise which results in contraction and stimulation of the surrounding cells. If this is true then the trigger considers the action potential of the surrounding cells, if this is above a set value then the trigger sends signals to the action potential block that initiate it to simulate depolaristion. The action potential block produces a simplified action potential profile during depolarisation that is based upon a sine wave, positive half only. There are two refractory periods during the depolarisation of a cell, the Effective Refractory Period (ERP), during which the cell can not be re-stimulated, and the Relative Refractory Period (RRP) which follows ERP and during which the cell may only depolarise under certain conditions [25]. The action potential block uses the signals from the trigger block to determine whether it is between ERP and RRP or if it has passed through both of these periods, in the former case it simulates an action potential that is reduced in both duration and amplitude, in the latter case the simulated action potential achieves both its maximum amplitude and duration. Both the trigger and action potential blocks

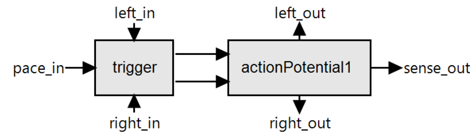

Fig. 1. Non-specialised myocardium model

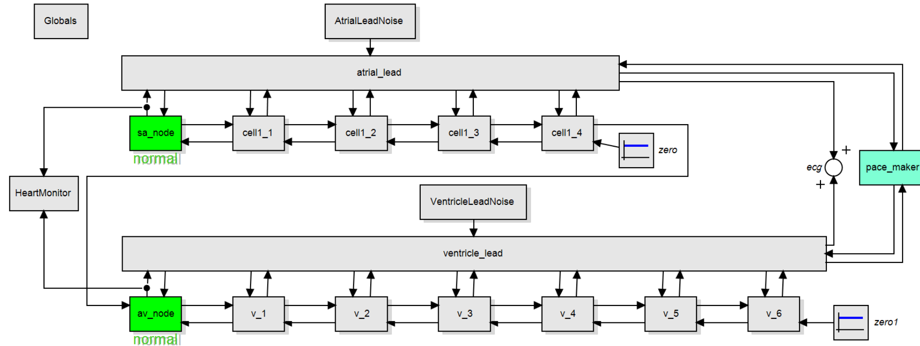

Fig. 2. Overall CT model of the heart

are modelled using signal type communications as there is no energy flow between the cells and so it is not necessary to use the bond graph capability of 20-sim.

The heart model consists of two strings of the myocardia models, five at the top of the model representing the atria, which are connected to the start of seven sub-models below representing the ventricles (Figure 2). At the start of each of these strings of models, there is a specialised myocardia model, labelled the `sa_node` and the `av_node`. These are identical to the standard myocardia except that they have a modified trigger model that contains a timer, which, in the absence of an external stimulation from another cell, triggers the cell to depolarise at the intrinsic rhythm for that chamber of the heart. During normal heart operation, the timer in the `av_node` triggers that cell to depolarise, this triggers each of the cells in the atria in turn to depolarise. `cell1_4` depolarising is sensed by the trigger in the `av_node`, which then itself depolarises after the a period of time called the "AV delay". The cells of the ventricle then depolarise in turn, completing one cycle of the heart.

The heart CT model is able to simulate three types of faulty behaviour: two heart arrhythmia, sinus bradycardia and heart block, and electrical noise injected onto the pacemaker leads. Sinus bradycardia is implemented via a modified trigger sub-model in the `sa_node` so the node has a slower than normal intrinsic rhythm, heart block is implemented via a modified trigger in the `av_node` such that it ignores the action potential from `cell1_4` in the atria. The electrical noise, which may have multiple sources such as the patient's environment or their chest muscle activity[26], is injected directly into the two pacemaker lead models, `atrial_lead` and `ventricle_lead` in Figure 2. A noise profile may be selected in the debug config, this profile determines when and how much noise is added to the signal detected by each of the pace maker

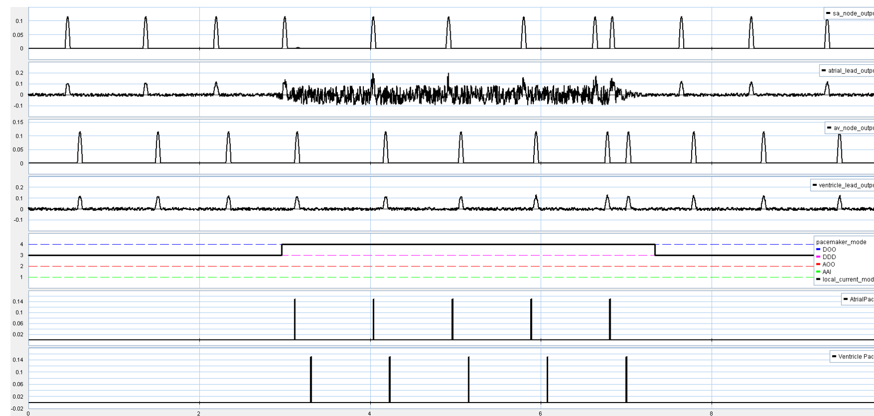

Fig. 3. Heart model output with noise injected onto the atrial lead

leads. Figure 3 shows the output of the heart model, the second and third plots on the graph show the signals returned to the pacemaker via the atrial and ventricle leads, a strong noise signal, with amplitude greater than the signals from the heart, may be seen on atrial lead between three and five seconds. These noise signals that will be used to test the mode changing accuracy of the pacemaker model during the DSE experiment.

3.2 Pacemaker

Specification The pacemaker specification outlines various constraints on the behaviour of an artificial cardiac pacemaker. Included in the specification is a description of how a pacemaker should respond when the electromagnetic signals being monitored are subjected to interference (noise). It is defined that:

> "In the presence of continuous noise the device response shall be asynchronous pacing" [7]

In order to model this behaviour, the modelled controller must be able to determine when the signals it is monitoring are corrupted by electromagnetic interference, and be able to acknowledge when this noise extends over some period of time. In the event that a monitored signal is subjected to noise for an extended period, the pacemaker should transition to an asynchronous pacing mode. To satisfy this, the modelled controller must be able to transition safely between a prescribed synchronous mode and an appropriate asynchronous alternative.

Model The DE model of the pacemaker controller has several layers, and the overall structure is illustrated using a class diagram in Figure 4. Whilst the class diagram includes some examples of high level operations, it does not list the entire implementation.

A low-level layer directly monitors and controls the CT environment, reporting any sensed activity in the heart, and actuating the delivery of artificial paces. Labeled as `IOManager` in Figure 4, this layer acts as an interface for the pacing algorithms to access the pacing leads. An intermediate layer manages the computation of significant events to be used by the pacing algorithms, identifying intrinsic activity from sensing data, calculating when necessary intrinsic behaviour is absent, and distinguishing between noisy and reliable data. Shown in Figure 4 as `ActivityTracker`, this layer is used to generate events necessary for the pacing algorithms from sensing data. A high-

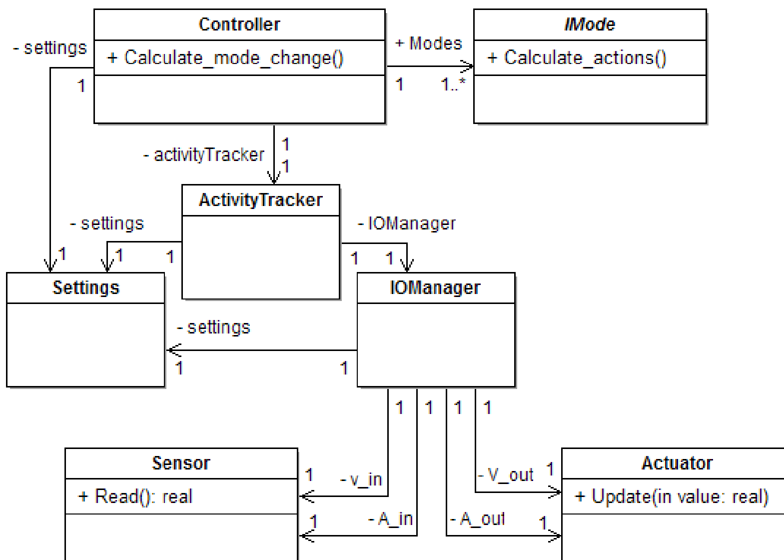

Fig. 4. Pacemaker model class diagram

level layer computes the necessity for artificial pacing, based on the events observed by the pacemaker. The controller can be instructed to operate according to any of the available modes, and decisions for pacing are directed by the active mode of operation. If a synchronous mode is selected and a continuously noisy signal is suspected, then the controller calculates an appropriate asynchronous alternative and switches into it. Figure 4 shows this layer labeled as `Controller`, and it maintains a suitable mode of operation based on events observed from sensing. Listing 1.1 shows the steps taken by the controller, including the sampling of sensing data, the calculation of an appropriate mode, and the delegation of the pacing activity to the appropriate mode.

Each available mode is modelled according to an interface (`IMode`), and the controller delegates mode-specific actions to the appropriate implementation. Listing 1.2 shows part of implementation of the 'DDD' mode.

An auxiliary class, labeled `Settings` in Figure 4, is used to store operational parameters that would typically be selected by a cardiologist. These include a desired

mode of operation, as well as timing constraints and tolerances. It also stores parameters to specify sensitivity thresholds for noise detection.

```
public Mode = <AAI> | <AOO> | <DDD> | <DOO>;

instance variables
  activity : ActivityTracker;
  settings : Settings;
  current_mode : Mode;

operations
  Step : () ==> ()
  Step() == (
   -- update timers
    activity.Step();

  -- Calculate if mode change is required
    Calculate_Mode_Change();

  -- delegate control to current mode
   modes(current_mode).Step();
  );

thread periodic(1E6,0,0,0)(Step);
```

Listing 1.1. Sample VDM-RT code with mode change

```
  -- Atrial sensing
  if activity.Last_A_Activity() > settings.ARP() and
     activity.Last_V_Activity() > settings.PVARP()
  then activity.Sense_A();

  -- Atrial pacing
  if activity.Last_A_Activity() >= settings.LRL()
  then activity.Pace_A();

  -- Ventricular sensing
  if activity.Last_V_Activity() > settings.VRP()
  then activity.Sense_V();

  -- Ventricular pacing (P-wave tracking)
  if activity.Last_V_Activity() > activity.Last_A_Activity() and
     activity.Last_A_Activity() >= settings.AV_Delay()
  then activity.Pace_V();
```

Type	*Name*	*Description*
monitored	atrial_sensed	atrial lead output
	ventricular_sensed	ventricle lead output
controlled	atrial_pace	pacing output to atrial lead
	ventricular_pace	pacing output to ventricle lead
	current_mode	pacing mode ID number for graphical output
shared	noise_window_length	length of noise window (ms)
	noise_roughness_threshold	numerical value indicating roughness (no units)
	quiet_period_length	length of time before we consider noise to have ended
	noise_profile_number	which noise profile to use in a simulation

Table 1. Parameters defined in the co-model contract.

Listing 1.2. DDD mode calculating pacing activity

3.3 Contract

The contract for this model (Table 1) closely mimics the interface between the cyber and physical elements in the real heart and pacemaker system. Specifically it represents the connection between the pacing leads and the pacemaker itself. The pacing leads both monitor and stimulate the heart through a single physical connection, the connections described in the contract are uni-directional so it is not possible to model a single pacing lead, but instead for each lead we must model the sensing action as a monitored variable and the pacing action as a separate controlled variable. There are five other items defined in the contract. The current_mode controlled variable is used to export the pacemaker operating mode, as a number, to the CT model such that it could be stored in the CT model logs files during simulation. This is not required for the modelling of the heart or the pacemaker but is instead to support the post simulation analysis of the pacemaker mode changing behaviour, described further in Section 4. The final four SDPs in the contract represent the parameters the model was developed to explore, with the effects of noise_window_length and noise_roughness_threshold presented in this paper.

4 Design Space Exploration

To function correctly the pacemaker must have an accurate view of the recent activity of the heart it is monitoring and there are a range of issues that may inhibit it having this view. The view is derived directly from the signals it receives from its pacing lead(s), these may break or become detached from the wall of the heart or experience electrical noise due to either the wearer's environment or their own body. The DSE experiment considers the tuning of a hypothetical software method to detect lead noise and allow the pacemaker to operate in the correct mode.

The noise detection and mode change algorithm has three main parameters that affect its behaviour. The first is the `noise roughness threshold` which comes

Parameter	*Values*
`noise roughness threshold`	0.0005; 0.001; 0.002; 0.003; 0.004; 0.005
`noise window length`	5;10;20;30;40;50

Table 2. Parameter values uses in the DSE sweep

from the second derivative of the lead signal and assumes that the frequencies within a noise signal are higher than those of the normal heart signals. The second parameter is the `noise window length` which defines the number of samples used to determine if noise currently exists on a lead. Too low a value can make the noise detection over-sensitive, while too high a value could make the pacemaker unresponsive to noise. The final parameter is the `quiet period length`, which defines how many samples must pass with a noise value below the threshold before the pacemaker considers the noise to have passed. This is to reduce the sensitivity of the noise detection to temporary drops in the measured noise value.

To shorten the length of simulations a set of predefined noise profiles was used to determine when noise will be injected into the signal captured by each lead. The profile is selected by setting the `noise profile number`. There were five profiles, which variously exercise the leads, for example, profile 1 applies noise to both leads for 4s, 3s after the simulation starts, profile 2 applies 1s of noise into each lead individually before applying two more noise periods into both at them at the same time. None of the profiles ever reduce the noise to a non-existent level, there is always a small ripple on both lead signals, so in a sense, the periods discussed previously could be termed 'elevated noise'. Profile 5 does not include any elevated noise.

Table 2 shows the actual parameter values that were fed into the experiment. There are six of each, making a total of 36 simulations, which in this case can be performed in a little over an hour on a modest laptop computer. This design space is not of the order that makes a complete exploration infeasible, but it does allow us to investigate the use of Crescendo DSE in this domain.

While the graphical output from 20-sim (Figure 3) is useful for observing the behaviour of the modelled heart for a single simulation run, it does not lend itself to the comparison of multiple simulations. To achieve our goal of comparing the effectiveness of multiple sets of noise detection parameters we needed a numerical output that we could post-process. Crescendo allows variables from both the CT and DE models to be logged in CSV (comma separated value) files, and so this was used to keep a record of, amongst other things, the noise magnitude and pacemaker operating mode for each simulated time slice.

Crescendo records the CSV logs for each simulation in its own file in its own directory and this is where its automated support for co-model analysis ends. To determine which noise detection parameters are best, we must determine which result in the pacemaker being in the correct mode for the greatest proportion of the simulated time. The first step in calculating this is to determine what mode the pacemaker should have been in. This is done by comparing the noise magnitude for each simulated time step from the CT model with a threshold. If the magnitude is below the threshold, the pacemaker mode should be synchronous (AAI/DDD), and if it is equal to or greater than the thresh-

		noise roughness threshold					
		0.0005	0.001	0.002	0.003	0.004	0.005
noise window	5	53.3	53.3	53.3	53.9	64.7	70.8
	10	53.3	53.3	53.3	64.3	76.0	83.1
	20	53.3	53.3	62.5	74.6	85.7	92.5
	30	53.3	53.3	69.0	83.5	81.3	81.0
	40	53.3	53.6	68.9	82.3	88.0	77.2
	50	53.3	53.3	72.2	83.8	82.6	73.5

Table 3. Percentage of time the controller was in the correct mode (availability) for a range of noise window lengths (rows, 5–50ms) and noise roughness thresholds (columns, 0.0005–0.005) for a pacemaker initially in DDD mode

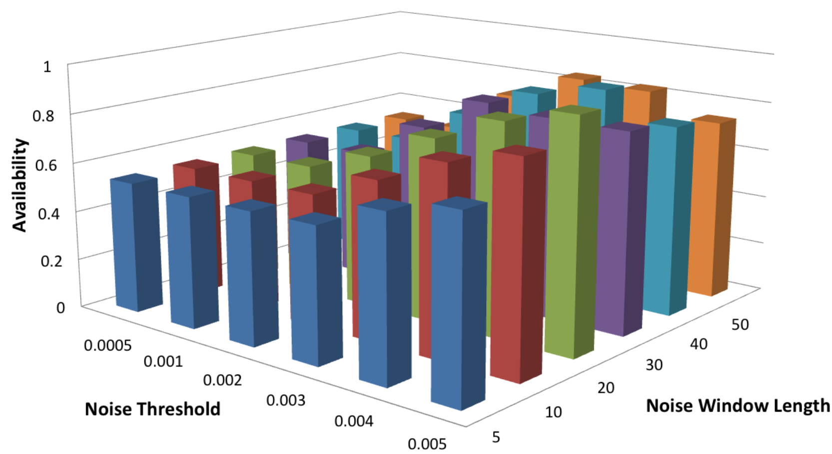

Fig. 5. The simulation results as a graph showing the best result is with a noise window of 20ms and a roughness value of 0.005

old, the mode should be asynchronous (AOO/DOO). The next step is to compare what mode the pacemaker was in, as reported by the DE model, with the expected mode calculated in the previous step resulting in one of the following four conditions: *True Synchronous*/*True Asynchronous* when the pacemaker mode was appropriate for the current noise level, and *False Synchronous*/*False Asynchronous* when the pacemaker mode was incorrect for the current noise amplitude. Finally the ratio of "True" modes to "False" modes is calculated to determine the score for each design. This data was fed into a spreadsheet to allow tabular (Table 3) or graphical (Figure 5) presentation.

It was necessary to create a Java application to analyse the raw simulation output data, and this highlights a gap in Crescendo's current support for DSE. This gap, which prevents Crescendo from being able to read and process simulation results, also prevents it from autonomously using more efficient forms of design space exploration such as simulated annealing, space culling [27] and genetic methods [28]. From our experience here we can start to identify functionality that would be required to "close the loop" and

inform the development of more advanced tool chains such as those being developed in INTO-CPS[9]. As a minimum we suggest:

- Post-simulation support to read in CSV log files for each simulation;
- definition of functions that act on values in each time slice and return a result (e.g., is noise greater than a threshold?);
- definition of functions that act across the whole simulation (e.g., minimum, maximum, mean);
- counters that persist for a single simulation (e.g., count of times mode was correct for the noise level);
- functions that are performed after reading the results of a simulation (e.g., proportion of time pacemaker mode was correct);
- simple tabulation of results for human consumption.

5 Conclusions and Challenges

DE Model In addition to addressing noise acknowledgment and response, we have demonstrated a dramatically different approach to modelling the DE controller from our previous work [12]. The previous version directed pacing functionality via a complex series of nested conditionals. Rather than model pacing modes explicitly, it achieved the constraints prescribed by each mode by setting boolean flags associated with each possible function. This resulted in a single control algorithm for all modes, which was difficult to maintain and debug.

The model presented here exercises design pattern guidance developed throughout DESTECS. The DE controller employs the Modal Controller Pattern described in [4] to simplify the description of pacemaker behaviour by separating out the pacing behaviour of each mode into an independent class. This enables modification of a single mode without risk of inadvertent side effects on the behaviour of other modes. The overall structure of the model is influenced by other patterns, such as the IO Synchronisation Pattern (exercised by the IOManager class).

CT Model Our new CT model with its string of identical myocardia models is an improvement over of the previous two capacitor model in several ways. First, the structure requires no artificial feedback connections in the model that are not present in a real heart and thus the two heart faults, Sinus Bradycardia and Heart Block, are implemented without the need to adjust any other part of the heart model, increasing confidence that this is a good base for modelling further cardiac faults. The structure could also be extended to model multiple, longer and parallel heart fibres opening the possibility of modelling more complex arrhythmia such as Atrial Fibrillation.

The pacemaker leads and their connections to the heart fibres are explicitly represented in the heart model rather than just being a voltage signal read from the capacitor in our earlier model. This presents two avenues of exploration: the effects of placement and area of contact of the pacing leads, and modelling the results on pacing and sensing of a lead becoming detached from the heart wall.

[9] http://into-cps.au.dk

Monitoring The CSV log output feature of Crescendo allowed us to monitor the state of both the heart model and the pacemaker for the purpose of our analysis. Where it failed to meet our needs was in providing support in itself to read in and analyse the content of the logs. In Section 4 we outlined some functions that could provide this support and in doing so permit use of more efficient DSE techniques. We hope that this experience will influence the design of tool chains in the INTO-CPS project.

Future Work We have shown that Crescendo is suitable for co-modelling a heart and pacemaker but that, in its current form, the model is only suitable for studying short-timescale effects, the simulation speed on a 2.4 GHz dual core CPU being on the order of 1/10th real time. Thus the model is suitable for our goals of studying the effects of mode changing, noise detection, pacing lead faults, etc. We would also like to consider effects over longer simulated times, such as the correctness of pacing data gathered by the pacemaker, total energy consumed or even whole life cycle of pacemaker implantation to removal), in which case it will be necessary to pair the current DE controller with either a simplified CT model and IOManager to reduce the co-modelling overhead, or with a completely DE heart model and simulate entirely within Overture.

References

1. J. Fitzgerald, P. G. Larsen, K. Pierce, and M. Verhoef, "A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems," *Mathematical Structures in Computer Science*, vol. 23, no. 4, pp. 726–750, 2013.
2. K. Pierce, J. Fitzgerald, and C. Gamble, "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.
3. J. Fitzgerald, K. Pierce, and C. Gamble, "A Rigorous Approach to the Design of Cyber-physical Systems through Co-simulation," in *Workshop on Open Resilient human-aware Cyber-physical Systems (WORCS-2012) – Supplement to Proc. Dependable Systems and Networks (DSN) 2012*, M. Kaâniche, M. Harrison, H. Kopetz, and D. Siewiorek, Eds. IEEE, 2012.
4. J. Fitzgerald, P. G. Larsen, and M. Verhoef, Eds., *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014. [Online]. Available: http://link.springer.com/book/10.1007/978-3-642-54118-6
5. D. Cofer, J. Hatcliff, M. Huhn, and M. Lawford, "Software Certification: Methods and Tools (Dagstuhl Seminar 13051)," *Dagstuhl Reports*, vol. 3, no. 1, pp. 111–148, 2013.
6. D. Méry, B. Schätz, and A. Wassyng, "The Pacemaker Challenge: Developing Certifiable Medical Devices (Dagstuhl Seminar 14062)," *Dagstuhl Reports*, vol. 4, no. 2, pp. 17–37, 2014.
7. Boston Scientific, "PACEMAKER System Specification," Boston Scientific, Tech. Rep., January 2007, http://www.cas.mcmaster.ca/sqrl/_SQRLDocuments/PACEMAKER.pdf.
8. H. D. Macedo, P. G. Larsen, and J. Fitzgerald, "Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM," in *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, ser. Lecture Notes in Computer Science, J. Cuellar, T. Maibaum, and K. Sere, Eds., vol. 5014. Springer-Verlag, 2008, pp. 181–197.
9. D. Méry and N. Singh, "Trustable formal specification for software certification," in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Lecture Notes in

Computer Science, T. Margaria and B. Steffen, Eds. Springer, 2010, vol. 6416, pp. 312–326.

10. A. O. Gomes and M. V. Oliveira, “Formal Development of a Cardiac Pacemaker: From Specification to Code,” in *Formal Methods: Foundations and Applications*, ser. Lecture Notes in Computer Science, J. Davies, L. Silva, and A. Simao, Eds., vol. 6527. Springer, 2011, pp. 210–225.
11. J.-R. Abrial, W. Su, and H. Zhu, “Formalizing Hybrid Systems with Event-B,” in *Abstract State Machines, Alloy, B, VDM, and Z*, ser. Lecture Notes in Computer Science, J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, Eds., vol. 7316, 2012, pp. 178–193.
12. C. Gamble, M. Mansfield, and J. Fitzgerald, “The Co-Simulation of a Cardiac Pacemaker using VDM and 20-sim,” in *Procs. Workshop on Trustworthy Cyber-Physical Systems*, ser. Technical Report Series, J. S. Fitzgerald, T. Mak, A. Romanovsky, and A. Yakovlev, Eds., vol. CS-TR-1347. School of Computing Science, Newcastle University, UK, 2012.
13. M. Verhoef, P. G. Larsen, and J. Hooman, “Modeling and Validating Distributed Embedded Real-Time Systems with VDM++,” in *FM 2006: Formal Methods*, ser. Lecture Notes in Computer Science 4085, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Springer-Verlag, 2006, pp. 147–162.
14. J. F. Broenink, “Modelling, Simulation and Analysis with 20-Sim,” *Journal A Special Issue CACSD*, vol. 38, no. 3, pp. 22–25, 1997.
15. V. Duindam, A. Macchelli, S. Stramigioli, and H. Bruyninckx, *Modeling and Control of Complex Physical Systems*. Springer, 2009.
16. M. Heemels and G. Muller, *BODERC: Model-Based Design of High-tech Systems*, 2nd ed. Den Dolech 2, Eindhoven, The Netherlands: Embedded Systems Institute, March 2007.
17. P. Fritzson and V. Engelson, “Modelica - A Unified Object-Oriented Language for System Modelling and Simulation,” in *ECCOP ’98: Proceedings of the 12th European Conference on Object-Oriented Programming*. Springer-Verlag, 1998, pp. 67–90. [Online]. Available: http://www.modelica.org/documents/ModelicaSpec32.pdf
18. G. Nicolescu, H. Boucheneb, L. Gheorghe, and F. Bouchhima, “Methodology for Efficient Design of Continuous/Discrete-Events Co-Simulation Tools,” in *High Level Simulation Languages and Applications*, J. Anderson and R. Huntsinger, Eds. San Diego, CA: SCS, 2007, pp. 172–179.
19. “IEEE Standard for Modeling and Simulation: High Level Architecture (HLA)– Framework and Rules,” *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pp. 1 –38, August 2010.
20. J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, “Taming Heterogeneity – the Ptolemy Approach,” *Proc. of the IEEE*, vol. 91, no. 1, pp. 127–144, January 2003.
21. C. G. Cassandras, *Analysis and Design of Hybrid Systems: a Proceedings Volume from the 2nd IFAC Conference*. Elsevier, Jun. 2006.
22. A. Platzer, *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer, 2010.
23. C. Gamble and K. Pierce, “Design space exploration for embedded systems using co-simulation,” in *Collaborative Design for Embedded Systems*, J. Fitzgerald, P. G. Larsen, and M. Verhoef, Eds. Springer Berlin Heidelberg, 2014, pp. 199–222.
24. A. Murthy, M. A. Islam, E. Bartocci, E. M. Cherry, F. H. Fenton, J. Glimm, S. A. Smolka, and R. Grosu, “Approximate bisimulations for sodium channel dynamics,” in *Computational Methods in Systems Biology*, ser. Lecture Notes in Computer Science, D. Gilbert and M. Heiner, Eds., vol. 7605. Springer, 2012, pp. 267–287.
25. T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre, “Quantitative verification of implantable cardiac pacemakers over hybrid heart models,” *Information and Computation*, 2014, http://dx.doi.org/10.1016/j.ic.2014.01.014.

26. J. Wranicz, M. Chudzik, and I. Cygankiewicz, "From unipolar to bipolar leads: Fewer problems, more advantages?" *PROGRESS IN BIOMEDICAL RESEARCH*, vol. 8, pp. 201–205, 2003.
27. M. P. Christiansen, M. Larsen, and R. N. Jørgensen, "Collaborative model based development of adaptive controller settings for a load-carrying vehicle with changing loads," in *CIOSTA XXXV Conference*, D. D. Bochtis and C. A. G. Sørensen, Eds., July 2013.
28. G. B. Leyland, "Multi-objective optimisation applied to industrial energy problems," Ph.D. dissertation, École Polytechnique Fédérale de Lausanne, 2002.

TASTE for Overture to keep SLIM

Marcel Verhoef and Maxime Perrotin

European Space Agency, ESTEC, P.O. Box 299, 2200 AG Noordwijk, NL
`first.last@esa.int`

Abstract. Both the TASTE and Overture toolsets have successfully demonstrated that method integration is a very promising strategy to create robust and effective tool suites. We explore the path of integrating both open source tool suites, to leverage their effectiveness even further, with the aim to provide a potential platform for the end-to-end design and engineering of dependable embedded systems, with a focus on spacecraft avionics.

1 Introduction

The European Space Agency (ESA) designs, commissions and operates launchers and spacecraft for a multitude of mission profiles, such as science and robotic exploration, earth observation, navigation, telecommunications and human spaceflight. It is paramount that software plays a significant role in this domain, as all of these applications rely on the use of computers in order to reach their mission objective. Computers are used, amongst others, to provide attitude and orbit control, payload data handling, thermal control and health assessment and management. In this paper, we focus on the spacecraft itself, or rather the avionics subsystem that provides the aforementioned services, even though the Earth bound ground segment and operations account for a significant part of the entire mission eco-system. We take this focal point because after lauch, access to the spacecraft is restricted to telecommand and telemetry only, which limits our options to monitor and influence the spacecraft behavior.

The complexity of spacecraft design is driven by the need to operate under extreme environmental conditions for very long periods of time, typically several years upto decades. Moreover, the launch into space as well as execution of the mission itself poses engineering challenges in its own right. Even though spacecraft are continuously operated under ground control, some level of autonomy is always required in order to ensure mission success. Communication delays may take several minutes due to the distance between ground station and the spacecraft, or may not be possible at all for some time, for example during the launch phase or due to partial visibility of the spacecraft orbit in relation to the position of the ground station(s) or when another celestial object is directly in the line of sight between the spacecraft and Earth. Sometimes, communication is relayed as direct communication to Earth is not possible. An example of this is the Philae probe that recently landed on the 67P/Churyumov-Gerasimenko

comet, which communicates to Earth through the Rosetta mothership. This implies that action needs to be taken by the spacecraft itself if unforeseen events occur in between ground contact opportunities, at the very minimum to bring the spacecraft back into a known "safe" state.

Moreover, maintenance of the spacecraft is typically not possible, or very expensive and complex,[1] which implies that robustness to "wear and tear" needs to be addressed upfront as equipment performance will deteriorate over time or may even break down unexpectantly. In contrast, it is interesting to note that software maintenance is technically feasible and relatively low cost, by updating over the air. Nevertheless, it remains complex because it directly affects system operability and the overall health state.

It is clear that resilience to faults is an important design driver in order to demonstrate that the level of dependability is sufficient to support the mission objectives during its entire life-time. This focal point propagates throughout all system design artifacts and all supporting life-cycle processes [8] and is probably the single most dominant contributor to the cost of the mission. Therefore, improvement of our capability to support design and analysis of dependability has a high potential gain in terms of cost and risk reduction. This is one of the reasons why ESA is actively supporting research and development in this area, for example through funding mechanisms such as TRP and GSTP [2].

Model based software and systems engineering (MBSSE) is a focal point in these activities, with the aim to alleviate the increasing project schedule pressure by improving the quality of the early design artifacts such that less effort is required in the later engineering stages and supporting processes, in particular for on-board software. Rigorous development approaches are proposed to support a wide range of engineering activities, such as:

- the use of ontologies and formal verification techniques to create a catalogue of system and software properties, which can form the basis for correct-by-construction software synthesis and re-use of requirements across missions.
- the use of architecture description languages to explore system resilience by analysing explicit fault models using model checking [4,9].
- improve the production of flight software by integrating well-founded formal technologies in those parts of the engineering chain where their benefit is clear and the gain is significant.
- the use of time and space partitioning kernels to implement mixed criticality applications on multi-core processors, supported by formal analysis techniques to study deterministic schedulability in the presence of uncertainty.

We first focus on the third item and we will revisit the other areas of interest in Section 4. Our starting point is The ASSERT Set of Tools for Engineering (TASTE) [20], which was originally developed in the EU-FP6 ASSERT project

[1] Such as the International Space Station and the Hubble Space Telescope.
[2] Technology Research Program, aimed at technology readiness level (TRL) 2-4 and General Support Technology Program, aimed at TRL 4-6/8. See `http://emits.sso.esa.int` and `http://sci.esa.int/sre-ft/37710-strategic-readiness-level/`

[3] and is still being actively maintained today. We will introduce this technology in Section 2, and discuss the possible aligment with Overture in Section 3. It is assumed that the reader is already familiar with Overture and Crescendo, we refer the reader to [14,15] and [11,7,6] respectively, for further details.

2 The TASTE toolset

TASTE is a robust and open-source tool-chain for software development. It targets heterogeneous embedded systems using a model-centric development approach. Likewise to Overture, it is also meant as a laboratory for experimenting with new software related technologies, based on free and open-source solutions. It supports a rigourous process using formal models and automatic code generation. It is focused on the development of high-reliability applications, but not necessarily restricted to aerospace. But of course, due to its heritage, many of the technologies used have their roots in this domain.

The main philosophy has been to create a consistent set of interoperable tools, based on mature languages with long-term support. Interoperability is achieved by providing translators between the notations used, such that applications can be composed of elements coming from a rich set of source languages. In essence, a domain specific language approach has been adopted, whereby a notation is only applied to that part of the problem domain where its matches best. Productivity is improved because the power of each individual language and supporting tool is leveraged by the ability to seamlessly integrate artifacts derived from those languages into the target application. The main elements of TASTE are:

- An interface definition language; **ASN.1** [2]
  The Abstract Syntax Notation One (ASN.1, an ITU-T standard, endorsed by ISO) is used to describe all datatypes and their constraints. This allows high-level specification of data in a language neutral format, for example to describe all telecommands (TC) and telemetry (TM). The set of standards also provides methods for the physical representation of data. TASTE provides a compiler not only to verify ASN.1 specifications, but also to generate application code in C and Ada, in order to read and write values based on the chosen physical representation. Moreover, it can also generate interface specification documents and test data sets fully automatically.

- A language to describe the system architecture; **AADL** [1,10]
  The Architecture Analysis and Design Language (AADL, approved by SAE International) is used to model the logical and physical architecture of the system. It provides both a textual and graphical format to denote the system composition. It is used in TASTE to capture the system structure: the hardware artifacts, their physical interfaces and the deployment of software over these elements. This allows for example early schedulability analysis, with tools such as MAST [3] and Cheddar [4]. AADL is designed as an extensi-

[3] `http://mast.unican.es`
[4] `http://beru.univ-brest.fr/~singhoff/cheddar/`

ble language; user defined properties can be attached to any language artifact and so-called annexes allow semantically consistent language extensions. The System-Level Integrated Modeling language (SLIM) was developed in the COMPASS [4] and FAME [9] projects to introduce explicit fault models [18,17].

– A language to specify system behavior; **SDL** [19]
  The Specification and Description Language (SDL, an ITU-T standard) is a formal language for describing state machines, both in a graphical and textual format. It is easy to use yet very powerful, with a precise and complete semantics. SDL natively supports ASN.1 data types. TASTE comes with an integrated SDL editor and Ada code generator called OpenGEODE. Execution (or simulation) traces can be visualised using the well-known message sequence chart (MSC) notation. TASTE also provides features to record and playback these traces for analysis and testing respectively.

Note that several other techniques are supported to specify system behavior (i.e. Simulink, SCADE, VHDL, Ada and C) but in this paper we focus on SDL. An full overview of TASTE is provided in [16,12], all tools can be found at [20]. The Ocarina tool [5] is used to generate high-integrity compliant Ada code from the AADL models. This code can be combined with the generated Ada code from the SDL and ASN.1 models to form an application, which can be deployed on the PolyORB-HI middleware. TASTE also provides the functionality to manage the build process. A range of target environments are supported:

– Linux (for non-realtime analysis of the application behavior);
– simulation environments supporting the SMP2 standard;
– real-time operating systems such as RTEMS and Xenomai, using virtualisation technologies such as QEMU and TSIM;
– RTEMS or Ada-Ravenscar run-times on target hardware, usually part of an avionics test bench in order to emulate all external sensors and actuators.

The TASTE development process consist of the following steps:

1. describe the system logical architecture and interfaces (ASN.1, AADL)
2. describe the system behavior (SDL)
3. describe the deployment of functionality on the avionics hardware
4. verify the models (i.e. schedulability, fault resilience)
5. generate code, build the system and download on simulator or target
6. monitor and interact with the system at run-time

The development process above is used in an iterative style, whereby results obtained in any step can lead to changes made in previous steps. The high degree of automation and the seamless integration of the tools and techniques enables short turn-around times.

3 Potential alignment with Overture

A birdseye overview of TASTE was presented in Section 2. We base the potential alignment of Overture and TASTE on a short quantitative comparison of both toolsets and associated processes.

- Overture and Crescendo are both aimed at early design validation, whereas TASTE is aimed at improving the production of high-quality software artifacts. Both are driven from formal models and rely on the integration of robust and specialist tools. Therefore, they naturally complement each other, as is also demonstrated by the goals of the INTO-CPS Horizon 2020 project [13], from which the future development of Crescendo is now actively supported. The aim of this project is to extend Crescendo to upstream modeling technologies such as SysML [5] and downstream with code generation to support hardware in the loop (HIL) simulations.

- Both Overture (through Crescendo with 20-sim [6]) and TASTE (through embedding code generated from Matlab/Simulink models) support the ability to perform co-simulation, albeit at different levels of abstraction. The need for this facility is clearly recognized in both approaches, in order to validate the design in the former and in order to test the generated application in the latter.

- The family of VDM languages, that is at the core of Overture and Crescendo, provide an extremely powerful set of model oriented specification languages that can be used for a wide range of applications. However, this versatility comes at a price. Creating a VDM model for a specific application domain usually requires development of a framework or a set of libraries in order to improve productivity and maintainability. In particular in the area of embedded systems, which are reactive systems by nature, the lacking built-in notion of state machines is an example of such a weakness.

- The reverse seems to be true for SDL, a language that is naturally suited to describe state machines with events described using ASN.1. SDL has built-in features to describe the actions taken during state transitions, but the expressive power of the notation is limited and does not allow complex algorithmic specifications. It has to rely on so-called “external calls” to model these complex transactions, but this also limits the ability to simulate SDL models as a synthesis step is required to get the external call (implemented in some other programming language) into to loop. Even though this process is automated in TASTE, it remains cumbersome, in particular if debugging is required across this interface to find the root cause of some test failure.

[5] `http://sysml.org`
[6] `http://www.20sim.com`

- VDM, SDL and ASN.1 support powerful techniques to support testing, to complement more heavy-weight verification technologies such as model checking and proof.

From this short assessment, incomplete as it may be, we can already conclude that a lot of common ground is available, despite several distinct differences. We argue that both tool environments can benefit from each other, by adressing the weakness of one by the strength of the other and vice versa. We believe that this will leverage the potential impact of both tool environments at lesser cost than implementing new features to resolve or address some of the weaknesses in either toolset individually. Of course, an investment must be made to achieve this goal, and we make a few suggestions here, ordered in increasing perceived impact and complexity:

- To convert ASN.1 definitions into VDM data types and values; this would enable the independent specification of functionality over those data types in VDM, with the ability to perform validation, for example using combinatorial testing in Overture.

- To convert VDM data types into ASN.1 definitions, with the obvious benefit that robust code generators are already available to read and write instances according to an independently selected physical format. This alleviates the need to write error prone "glue code" that is required when VDM models are coupled, either to simulators or real hardware.

- To couple OpenGEODE with Overture, such that SDL "external calls" can be realised by executing specific operation calls inside a VDM model, using the built-in interpreter already available in Overture. This would allow animation of SDL models without the need to perform a synthesis step, while re-using the powerful debugging features already available in both tool sets. The abstraction mechanisms in VDM can be fully exploited to write concise state machine transition specifications, rather than including hand-written code. It provides the VDM world with a strong and well-defined notion of state machines, almost for free. Note that this approach relies on the availability of the ASN.1 translators mentioned earlier, as this is the technique used to specify data types in SDL.

- Assuming the previous step is feasible and successful, a similar approach can be taken with Crescendo, which then also allows simulation of the physical world without the need to instantiate a full avionics test bench. This will provide the opportunity to use TASTE earlier in the design process.

- Analogous, implementing a code generator for MISRA-C or high-dependability ADA (SPARK-ADA) directly from the VDM specification, would allow to use these artifacts also for downstream engineering processes, whereby TASTE already provides the infrastructure for managing the build process and for creating and deploying the target image.

4 Summary and conclusions

We consider the suggestions from the previous section as the "low-hanging fruit" even though we recognise that the amount of effort to realise each step can be significant. We have not quantified this effort on purpose, as our primary aim is to motivate the community to consider these options as interesting and viable, both from an academic as well as a practical viewpoint. For this, we require further dialogue with the Overture community at the workshop.

On a more fundamental note, we see other opportunities on the horizon that pose a greater challenge and investment, but also with likewise higher potential benefits and rewards. Over the past decade, the VDM-RT dialect has matured into the modelling technology now used as the key asset in the Crescendo tool. The VDM++ notation was extended with asynchronous operations and specific language elements such as `BUS`, `CPU` and `system` to construct explicit architectures onto which other software artifacts specified in VDM can be deployed [21]. This allows the possibility to analyse the timing and performance behavior of real-time applications. This also proved to be a great step forward in the early life-cycle analysis of system resilience to faults, as was demonstrated in the DESTECS project. However, the fidelity of those models is restricted (i.e. only a single bus can connect between any two `CPUs`) and defining (and changing) properties of the model is cumbersome (i.e. message length is implictly based on the "size" of the VDM datatype). Instead of improving the situation by extending the built-in notation, we believe that adopting AADL as the mechanism to describe the system architecture is a better, more future proof, solution.

The richness of the AADL notation allows to describe a far wider range of embedded applications, keeping the existing notions of deployment already available in VDM-RT intact, but then described in AADL. Eclipse based tool support for AADL is readily available, for example OSATE2 [7]. Moreover, AADL was defined with extensions in mind, which may provide a very convenient mechanism to change model properties without affecting the structure of the model. And last but not least, other modeling extensions, such as the SLIM language can be used orthogonal to the same model, allowing the application of other verification and schedulability analysis tools. It even opens up the possibility to describe and analyse mixed criticality applications on multi-core or time and space partioning kernels, for example by using the ARINC653 AADL extension.

In summary, we offer our brief viewpoint on the future of Overture in this paper, as input to the on-going discussion on the strategic research agenda of the open source project. Of course this is driven from our business perspective and background. Nevertheless, we believe that these suggestions are sufficiently generic to be of interest to a wider community and we hope that they will trigger the appetite to consider coupling TASTE with Overture. Furthermore, we also hope that it brings the different formal methods communities closer together, as we believe that gaining critical mass is paramount not only to gain momentum

[7] `https://wiki.sei.cmu.edu/aadl/index.php/Osate_2`

in terms of tool development and cool stuff we can do with it, but also to increase our user community.

References

1. SAE AS 5506B: Architecture Analysis & Design Language (AADL) (2012), http://standards.sae.org/as5506b/
2. ITU X.680-X.693 : Information Technology - Abstract Syntax Notation One (ASN.1) & ASN.1 encoding rules, http://www.itu.int/rec/T-REC-X.680-X.693-200811-I/en
3. ASSERT : Automated proof based system and software engineering for real-time applications, http://cordis.europa.eu/project/rcn/71564_en.html
4. COMPASS project home page, http://compass.informatik.rwth-aachen.de/
5. Conquet, E., Perrotin, M., Dissaux, P., Tsiodras, T., Hugues, J.: The TASTE toolset: turning human designed heterogeneous systems into computer build homogeneous software. In: European Congress on Embedded Real-Time Software (ERTS) (May 2010), http://oatao.univ-toulouse.fr/3357/1/Hugues_3357.pdf
6. CRESCENDO tool home page, http://crescendotool.org
7. DESTECS project home page, http://www.destecs.org
8. ECSS: European Coorporation for Space Standardization, http://www.ecss.nl
9. FAME project home page, https://es-static.fbk.eu/projects/fame/
10. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. SEI Series in Software Engineering, Addison-Wesley Professional (September 2012)
11. Fitzgerald, J., Larsen, P.G., Verhoef, M.: Collaborative Design for Embedded Systems: Co-modelling and Co-simulation. Springer (2014)
12. Hughes, J., Zalila, B., Pautet, L., Kordon, F.: From the prototype to the final embedded system using the Ocarina AADL tool suite. Transactions on Embedded Computing Systems (TECS) 7(4) (2008), http://dl.acm.org/citation.cfm?id=1376810
13. INTO-CPS project home page, http://into-cps.au.dk
14. 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
15. Overturetool project home page, http://www.overturetool.org
16. Perrotin, M.: What is TASTE?, http://download.tuxfamily.org/taste/misc/what_is_taste.pdf
17. RWTH Aachen University (editors): Semantics of the COMPASS system-level integrated modeling (SLIM) language. Tech. rep. (May 2013)
18. RWTH Aachen University (editors): Specification of the COMPASS system-level integrated modeling (SLIM) language. Tech. rep. (May 2013)
19. ITU-T Rec. Z.100 (12/2011) Specification and Description Language - Overview of SDL-2010, http://www.itu.int/rec/T-REC-Z.100/en
20. TASTE community home page, http://taste.tuxfamily.org/
21. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and validating distributed embedded real-time systems with vdm++. In: FM2006: Formal Methods. Lecture Notes in Computer Science, vol. 4085, pp. 147–162 (2006)